Mining Cohesive Patterns in Sequences and Extreme Multi-label Classification

Proefschrift

voorgelegd tot het behalen van de graad van
Doctor in de Wetenschappen: Informatica
aan de Universiteit Antwerpen
te verdedigen door

Len Feremans

Promotor: prof. dr. Bart Goethals

Antwerpen, 2020

Mining Cohesive Patterns in Sequences and Extreme Multi-label Classification
Nederlandse titel: *Cohesieve Patronen ontdekken in Sequentiële Data en Extreme Multi-label Classificatie*

Samenvatting

In de huidige maatschappij genereert ieder van ons een enorme hoeveelheid data. Datawetenschappers zijn actief met het ontwikkelen van algoritmen die data verwerken, patronen herkennen en leren uit deze gegevens. Deze algoritmen hebben een enorm effect op ons dagelijks leven en de economie: mensen chatten, posten, liken, vinden, zoeken, winkelen en consumeren informatie en diensten online, en de impact van kunstmatige intelligente op deze activiteiten is enorm.

In **hoofstuk 1** situeren we ons onderzoek in deze brede context. Een eerste toepassing van ons onderzoek is om automatisch patronen te identificeren en te gebruiken om bijvoorbeeld het onderhoud en de levensduur van windturbines op zee te optimaliseren. In het algemeen, is het *vinden van patronen in sequentiële of chronologische gegevens een belangrijk uitdaging* binnen data science. Dat wil zeggen, het is uiterst nuttig om een algoritme te hebben dat relevante informatie of interessante patronen automatisch kan vinden. Er zijn echter verrassend veel onopgeloste uitdagingen voor algoritmen om interessante patronen te vinden in een stroom van gegevens. Allereerst verwerken weinig algoritmen rechtstreeks een enkele lange reeks, en vereisen ze dat de reeks gesplitst wordt in kleinere sequenties, wat nadelen heeft. Ten tweede, is het ook een voortdurende uitdaging om te definiëren wat “interessant” is voor een computer. Vaak vinden algoritmen enkel frequente patronen, maar dit zorgt vooral voor veel patronen die vaak minder interessant zijn. Tenslotte moeten we nadenken over de verschillende soorten van patronen en de beperking dat een computer alle interessante patronen kan berekenen binnen een redelijke tijd. Een tweede probleem dat we in dit proefschrift bestuderen, is hoe we *gegevens automatisch kunnen classificeren* door de logica of patronen geassocieerd met elk onderwerp te leren. Wikipedia heeft bijvoorbeeld miljoenen artikelen en aan elk artikel zijn één of meer onderwerpen gekoppeld. Een classificatie algoritme leert de logica achter elk onderwerp en voorspelt onderwerpen voor nieuwe documenten. Hierbij zijn we geïnteresseerd in algoritmen die nieuwe voorspellingen maken in milliseconden voor datasets met miljoenen labels.

In **hoofdstuk 2** beschrijven we FCI$_{seq}$. Dit algoritme vindt alle *cohesieve itemsets* in een sequentie op een efficiënte manier. Hierbij, definiëren we cohesieve itemsets als een verzameling van symbolen, of items, die vaak samen voorkomen in een sequentie. Bestaande algoritmen zoeken meestal frequente patronen, maar daar komen de symbolen in elk patroon vaak samen voor door toeval. Omdat cohesie geen anti-monotone maat is, vertrouwen we op een bovengrens om kandidaat patronen en verzamelingen waarvan het kandidaat patroon een deelverzamelingen is, te snoeien. We bewijzen theoretisch dat deze bovengrens correct is. We breiden dit algoritme uit en zoeken daarbij naar sequentiële patronen, waarbij de volgorde van elk symbool belangrijk is. We zoeken ook naar een tussenvorm van cohesieve itemsets en sequentiële patronen, namelijk dominante episodes. Voor episodes is de volgorde tussen sommige symbolen belangrijk en tussen anderen niet. Op basis van de gevonden cohesieve itemsets vinden we ook associatie regels. Deze regels kunnen gebruikt worden om symbolen te voorspellen in een sequentie. Met behulp van teksten, zoals de tweets van Donald Trump

en het boek over "de oorsprong der soorten" van Charles Darwin, tonen we aan dat patronen met een hoge waarde van cohesie, kwalitatief interessant zijn. We voeren ook verschillende vergelijkende proeven uit waarbij we kunstmatig ruis toevoegen, vergelijkbaar met mogelijke scenario's in de echte wereld. Onze resultaten suggereren dat cohesie robuuster is voor ruis dan andere bestaande methoden en dat FCI$_{seq}$ superieur is in het ontdekken van patronen verborgen in sequenties.

In **hoofdstuk 3** definiëren we een maat van cohesie gebaseerd op kwantielen. Deze maat is gemakkelijk te interpreteren en meldt zowel frequente als minder frequente, maar altijd sterk samenhangende, sequentiële patronen, die andere methoden vaak niet vinden. Omdat op kwantiel gebaseerde cohesie ook geen anti-monotone maat is, vertrouwen we ook hier op een bovengrens om kandidaat sequentiële patronen en alle mogelijke supersequenties te snoeien. We bewijzen theoretisch dat deze bovengrens correct is. Het voorgestelde algoritme, QCSP, maakt verder ook gebruik van projecties op een databank. Hiervoor wordt telkens de prefix van een sequentieel patroon gebruikt waardoor de geprojecteerde databank kleiner wordt en we cohesie sneller kunnen berekenen voor langere patronen. We tonen empirisch aan dat QCSP zeer efficiënt is, dat wil zeggen ongeveer een orde van grootte sneller dan FCI$_{seq}$. Bovendien is cohesie gebaseerd op kwantielen robuuster voor ruis, of willekeurig voorkomende uitschieters in de data, dan cohesie. We vergelijken QCSP kwalitatief met twee recente algoritmen die interessante sequentiële patronen vinden in meerdere sequenties. In vergelijking met de eerste methode, SKOPUS, vinden we langere patronen en ook patronen die minder frequent voorkomen maar wel samenhangend zijn. In vergelijking met GOKRIMP vinden we dezelfde patronen, maar het omgekeerde is niet waar.

In **hoofdstuk 4** beschrijven we INSTANCEKNNFAST, een algoritme dat, gegeven een nieuwe vraag of *query*, heel snel zoekt naar de dichtstbijzijnde voorbeelden, of buren, in een grote databank. We doen dit efficiënt met behulp van twee strategieën uit information retrieval: term-at-a-time en document-at-a-time. Dankzij een strakke bovengrens op basis van een gepartitioneerde index, vergelijken we alleen met voorbeelden die mogelijk gelijkaardig zijn aan de query. Onze resultaten suggereren dat dit algoritme tot 25% sneller is dan vergelijkbare algorithmen, onder de assumptie dat de query bestaat uit een groot aantal termen (of attributen), en dat de databank veel lege waarden bevat. Vervolgens beschrijven we LCIF, een extreem multi-label classificatie algoritme. In deze methode beschouwen we elk voorbeeld als een rij in een matrix en elk attribuut als een kolom. Door gebruik te maken van INSTANCEKNNFAST, vinden we de dichtstbijzijnde rijen. Gebaseerd op een algoritme van slimme aanbevelingssystemen, berekenen we efficiënt de labels met de hoogste correlatie met elk attribuut. Tenslotte maken we een voorspelling voor nieuwe labels door beide scores te combineren. Op tien verschillende datasets uit verschillende domeinen, vergelijken we LCIF met soortgelijke algoritmen op verschillende evaluatiecriteria. Onze resultaten suggereren dat LCIF beter presteert en vooral veel sneller is, waarbij we voorspellingen maken in seconden of minuten waar andere methoden uren of dagen nodig hebben. Bovendien geeft LCIF uitstekende resultaten op precision@k op extreem grote datasets. LCIF laat toe dat we nauwkeurige voorspellingen maken voor labels, in minder dan 20 milliseconden per voorbeeld, op extreme grote datasets zoals Wikipedia en dit op een gewone computer.

Tenslotte, vatten we in **hoofdstuk 5** de belangrijkste bijdragen van dit proefschrift en het potentieel voor toekomstig onderzoek samen.

Acknowledgements

In the past years, I enjoyed my time at the University of Antwerp. In the role of assistant, I helped bachelor students with their exercises and projects for programming, databases and artificial intelligence. In the role of researcher, we developed novel algorithms and open-source software for pattern mining and extreme multi-label classification. What is not included in this thesis, is that we also developed algorithms for anomaly detection (Feremans et al. 2019a), proposed a generic framework and tool for pattern mining and anomaly detection in heterogeneous time series (Feremans et al. 2019b), and applied our research towards solving real-world problems and applications, such as, conditional monitoring of fleets of offshore wind turbines (Feremans et al. 2017a; Daems et al. 2019). I also worked on real-world projects for predicting anomalies in the water consumption for Colruyt, combining different enterprise databases using entity resolution for Cropland, and automated labelling of criminal records for the Federal police. It has been a fascinating and enriching period, where I enjoyed working together with my colleagues from Adrem Data Lab and researchers from other universities.

First, I am grateful towards all jury members, namely Kris Laukens, Floris Geerts, Celine Vens, Wannes Meert, Albrecht Zimmermann, Bart Goethals and Boris Cule, for their effort in thoroughly reviewing this thesis and their insightful feedback.

From the university of Antwerp, I wish to thank my promotor Bart Goethals for allowing me this chance. Bart is an exceptional person and an authority in pattern mining, recommender systems and data science in general. Your down-to-earth and enthusiastic manner of guidance was refreshing, and I cannot thank you enough for your guidance (and patience) the past years. Second, I want to thank Boris Cule, who was my main co-author and also a mentor for the past 6 years. Boris is always relaxed, humble, thorough, listens well and thinks hard before he says something. This makes him completely opposite from me, which is probably why we worked together so well.

From the DTAI Lab at the Katholieke Universiteit Leuven, I want to thank Vincent Vercruyssen and Wannes Meert. Our collaboration resulted in a novel pattern-based anomaly detection algorithm that will be part of Vincent's thesis. Vincent is ambitious, smart and, not unimportantly, very nice and fun to work together. From the mechanical engineering department at the VUB, I want to thank prof. Jan Helsen. Jan spent many days next to me at the university, resulting in two publications on pattern mining of offshore wind turbine fleets. Thank you for your guidance. From ITEC, an imec research group at the Katholieke Universiteit Leuven, I want to thank prof. Celine Vens for her valuable feedback on the multi-label classification method presented in the fourth chapter.

My daily workday at the Middelheim campus, in particular the lunch discussions and team building events were an absolute pleasure. I want to thank prof. Toon Calders and prof. Floris Geerts for there valuable advice and guidance. Stephen, sorry I made fun of you for about a million times. If teasing is indicative of affection, we should get married! Joeri, thank you for the incomplete PhD template. Sandy and Koen, thanks for all your advice. Also many thanks to my current and former fellow (post)doctoral researchers: Sam, Maarten, Joey, Olivier, Jan,

Mozghan, Lien, Koen, Elyne, Emin, Tayena, Cheng, Hassaan, Robin and others. Also special thanks to Kris Laukens and the biomina group, in particular, Aida, Charlie, Bart, Danh, Nicolas, Pieter Meysman, Pieter Moris, Sofie and Wout. You are all wonderful people and I hope we keep in touch for decades to come.

I also want to thank my family, my mother Yolande, father Eric, sister Kim, plus brothers Merijn and Lenz, plus mothers Linda and Claire, Ivo, Maaike, Sanne, godchild Manon, nieces and nephews Margot, Jente, Iza, Lynke, Mathias and Mats, the bomma, Luc and Erna! Also, special thanks to my closest friends Filip, Arianne, Seppe, Wouter, Leen, Susan, Maarten, Sven, Brit, Niels, Isabelle, Joeri, Evelyn, Gert, Eva, Frederik, Eva Sels, Erwin and many others.

Finally, in the past years, there have been periods where I have felt anxious, insecure or frustrated with my progress in restarting my career and pursuing science. However, when I come home to Pascale and my two sons, Leon and Jules, there is such a tsunami of happiness, love and support, that I feel I can handle everything. This thesis is by me, but it is also because of you. You are the best! Leon en Jules, iedereen, jong en oud, vindt graag nieuwe dingen uit. Het is belangrijk om veel te lezen en voor moeilijke onderwerpen, zoals wiskunde, altijd je tijd te nemen en je best te doen. Je programmeert nu al lego robots, maar - voor je het weet - gebruik je al je creativiteit en maak je zelf leuke computerspelletjes en computers. Jullie zijn de beste en papa heeft dit boekje kunnen schrijven omdat jullie en mama elke dag weer zo'n schatjes zijn.

You all supported me and enabled this work. *Len Feremans*
Thank you! *Antwerpen, 2020*

Contents

Samenvatting i

Acknowledgements v

Contents vii

Publications xiii

1 Introduction 1
- 1.1 Mining Patterns in an Event Sequence . . . 1
  - 1.1.1 Event Sequence . . . 2
  - 1.1.2 Pattern Mining . . . 3
  - 1.1.3 Itemsets and Episodes . . . 5
  - 1.1.4 Association Rules . . . 5
- 1.2 Multi-label Classification . . . 6
  - 1.2.1 Nearest Neighbours Classification . . . 6
  - 1.2.2 Top k-queries . . . 7
  - 1.2.3 Item-based Collaborative Filtering . . . 8
- 1.3 Overview . . . 8
- 1.4 Open-source Code . . . 9

2 Efficiently Mining Cohesive Patterns and Rules in Sequences 13
- 2.1 Introduction . . . 14
- 2.2 Problem Setting . . . 16
  - 2.2.1 Frequent Cohesive Itemsets . . . 16
  - 2.2.2 Representative Sequential Patterns . . . 18
  - 2.2.3 Dominant Episodes . . . 19
  - 2.2.4 Association Rules . . . 21
- 2.3 Mining Cohesive Itemsets . . . 22
  - 2.3.1 Depth First Search . . . 22
  - 2.3.2 Pruning . . . 24
  - 2.3.3 Computing the Sum of Minimal Windows . . . 28
- 2.4 Mining Representative Sequential Patterns . . . 30
  - 2.4.1 Computing Minimal Windows for Sequential Patterns . . . 30
  - 2.4.2 Algorithm . . . 31
- 2.5 Mining Dominant Episodes . . . 33
- 2.6 Mining Association Rules . . . 34
  - 2.6.1 Efficiently Computing Confidence . . . 34
  - 2.6.2 Algorithm . . . 36
- 2.7 Setting Parameters and Top-k Mining . . . 36

2.8 Experiments . . . 38
2.8.1 Comparison on Synthetic Benchmark . . . 38
2.8.2 Quality Comparison on Text Datasets . . . 43
2.8.3 Association Rules . . . 48
2.8.4 Performance Analysis . . . 50
2.9 Related Work . . . 53
2.10 Conclusion . . . 56

3 Mining Quantile-based Cohesive Patterns in Sequences 59
3.1 Introduction . . . 60
3.2 Problem Setting . . . 62
3.3 Mining Quantile-based Cohesive Sequential Patterns . . . 63
3.3.1 Prefix-projected Pattern Growth . . . 64
3.3.2 Incremental Computation of Prefix-projections . . . 66
3.3.3 Pruning . . . 68
3.4 Experiments . . . 72
3.4.1 Datasets . . . 73
3.4.2 Performance Comparison . . . 73
3.4.3 Quality Comparison . . . 75
3.5 Related Work . . . 77
3.6 Conclusion . . . 78

4 Extreme Multi-label Classification using Instance and Feature Neighbours 81
4.1 Introduction . . . 82
4.2 Problem Setting . . . 83
4.3 Linear Combination of Instance- and Feature-based kNN . . . 84
4.3.1 Instance-based kNN . . . 84
4.3.2 Feature-based kNN . . . 87
4.3.3 Linear Combination . . . 89
4.3.4 Thresholding . . . 89
4.4 Fast kNN Search . . . 90
4.4.1 Indexing . . . 90
4.4.2 TAAT and DAAT Traversal with Weak-And Pruning . . . 92
4.5 Experiments . . . 94
4.5.1 Experimental Setup . . . 94
4.5.2 Classification Performance LCIF . . . 97
4.5.3 Runtime Performance INSTANCEKNNFAST . . . 100
4.5.4 Runtime Performance LCIF . . . 102
4.6 Related Work . . . 103
4.7 Conclusion . . . 106

5 Conclusion and Outlook 111
5.1 Main Contributions . . . 111
5.2 Outlook . . . 112
5.2.1 Future of Pattern Mining In Sequences . . . 113
5.2.2 Improving Extreme Multi-label Classification . . . 116

A Additional Material for FCI_{seq} 119
A.1 Computing Minimal Windows for Sequential Patterns . . . 119

A.2 Compute Support of an Episode 120
A.3 Top 25 Patterns Discovered by FCI_{seq} on Species 121
A.4 Top 25 Patterns Discovered by FCI_{seq} on Trump 125

B Additional Material for QCSP 129
B.1 Weighted Quantile-based Cohesion 129
B.2 Top 20 Sequential Patterns Discovered by QCSP 130

C Additional Material for LCIF 133
C.1 Second Order Instance-based kNN 133

List of Figures 135

List of Tables 138

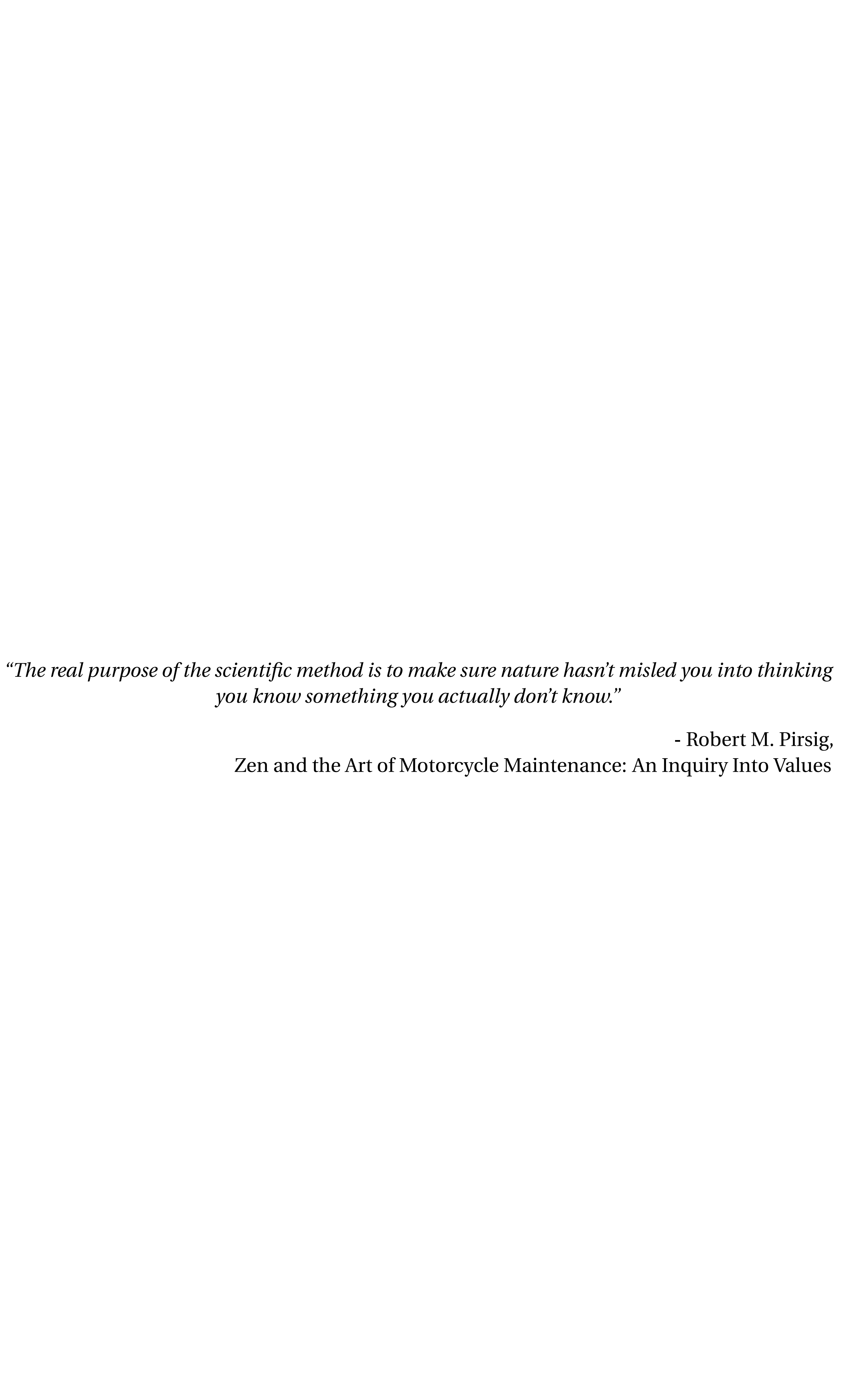
List of Algorithms 140

List of Definitions, Problems and Theorems 142

Bibliography 147

Publications

- Boris Cule, **Len Feremans** and Bart Goethals. Efficient Discovery of Sets of Co-occurring Items in Event Sequences. In *Proceeding of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2016)*, pages 361-377, 2016.
- **Len Feremans**, Boris Cule, Christof Devriendt, Bart Goethals and Jan Helsen. Pattern Mining for Learning Typical Turbine Response during Dynamic Wind Turbine Events. In *Proceeding of the ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC-CIE 2017)*, American Society of Mechanical Engineers, 2017.
- **Len Feremans**, Boris Cule, Celine Vens, and Bart Goethals. Combining Instance and Feature neighbors for Efficient Multi-label Classification. In *Proceedings of the IEEE International Conference on Data Science and Advanced Analytics (DSAA 2017)*, pages 109-118, IEEE, 2017.
- **Len Feremans**, Boris Cule and Bart Goethals. Mining Top-k Quantile-based Cohesive Sequential Patterns. In *Proceedings of the SIAM International Conference on Data Mining (SDM 2018)*, pages 90-98, 2018.
- Boris Cule, **Len Feremans** and Bart Goethals. Efficiently Mining Cohesion-based Patterns and Rules in Event Sequences. In *Data Mining and Knowledge Discovery (DAMI 2019)*, 33(4), pages 1125-1182, 2019.
- **Len Feremans**, Vincent Vercruyssen, Wannes Meert, Boris Cule, and Bart Goethals. A Framework for Pattern Mining and Anomaly Detection in Multi-dimensional Time Series and Event Logs. In *Post-Proceedings of the International Workshop on New Frontiers in Mining Complex Patterns (NFMCP 2019)*, 2019.
- **Len Feremans**, Vincent Vercruyssen, Boris Cule, Wannes Meert and Bart Goethals. Pattern-based Anomaly Detection in Mixed-type Time Series. In *Proceeding of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2019)*, 2019.
- Pieter-Jan Daems, **Len Feremans**, Timothy Verstraeten, Boris Cule, Bart Goethals and Jan Helsen. Fleet-oriented Pattern Mining Combined with Time Series Signature Extraction for Understanding of Wind Farm Response to Storm Conditions. In the *Second World Congress on Condition Monitoring (WCCM 2019)*, 2019.
- **Len Feremans**, Boris Cule, Celine Vens, and Bart Goethals. Combining Instance and Feature Neighbours for Extreme Multi-label Classification. *International Journal of Data Science and Analytics (JDSA 2020)*, 2020.



“The real purpose of the scientific method is to make sure nature hasn’t misled you into thinking you know something you actually don’t know.”

- Robert M. Pirsig,
Zen and the Art of Motorcycle Maintenance: An Inquiry Into Values

CHAPTER 1

Introduction

In the current world, each of us is generating an enormous amount of data. Households and companies hold various devices that continuously log all kinds of data in a streaming way. We, data scientists, are active with developing algorithms, that is, step-by-step instructions, that lead to an automatic way of processing and learning from data. These algorithms have an enormous effect on everyday life and the economy: People chat, post, like, search, shop and consume information online, and the impact of artificial intelligence on these activities is considerable. The largest companies in the world are technology-driven and compete in inventing algorithms that retrieve the most relevant information and suggestions, thereby creating increasingly smarter services and products for end-users. While commercial applications motivate much progress in data science, there are many applications where algorithms also help with challenges in society. In life sciences, energy and public services, advancement in data science can mean improvements in the medical or pharmaceutical domain or carbon-neutral electricity production using wind turbines. For example, recent improvements in deep-learning improve the diagnosis of patients by using software that analyses medical images and can highlight patterns that might elude human experts. Also, by applying algorithms to the analysis of offshore fleets of wind turbines, we can identify patterns. We can then use these patterns to optimise maintenance and lifetime for wind turbines.

1.1 Mining Patterns in an Event Sequence

An important data mining task is to find interesting patterns in sequential, or chronological, data. That is, having an algorithm that mimics a human to filter relevant information or patterns is extremely useful. However, for finding interesting *patterns* in a *single event sequence* there are, surprisingly, many unresolved challenges in algorithm design. First of all, few algorithms work on a single long sequence and require that we split the single sequence in many short sequences, which has disadvantages. Second, few algorithms work on event sequences consisting of *discrete* or *mixed-type* data. Third, it is hard to define what is "interesting" for a computer. A decade ago, most researchers used *frequency* to define what was interesting, but this leads to unsatisfactory patterns. Fourth, we can use *machine learning* to learn what is interesting and what is not. However, this requires labelled data, and if this is not available, or we do not know what is interesting, this is a problem. Fifth, there are many different types of patterns, and each type of pattern has its advantages and disadvantages. Finally, we have to design algorithms that are efficient to handle *big datasets* in a reasonable time.

1.1.1 Event Sequence

We are interested in analysing sequences of events. Figure 1.1 shows a fragment of the novel Moby Dick written by Herman Melville from Chapter 3. We create a single sequence of this fragment by removing all punctuation, transforming words to lower case and splitting words on spaces resulting in (*all, this, while, tashtego, ..., him*). Logs produced by computers, smart devices and machines are also sequences. Also, in industry there are numerous devices and machines that log data at each time point. For instance, wind turbines typically log sensor values, i.e., wind speed, actions of operators, and possibly warning or failure codes at each time point. Figure 1.2 shows an example of such a *heterogeneous* sequence (Feremans et al. 2019b). Here we show continuous variables, i.e., wind speed and power output, and discrete events logged by the embedded computer. Remark that from a data science perspective, algorithms make abstraction of the specific semantics underlying each kind of sequence and require a single sequence of ⟨*timestamp*, *value*⟩ pairs.

All this while Tashtego, Daggoo, and Queequeg had looked on with even more intense interest and surprise than the rest, and at the mention of the wrinkled brow and crooked jaw they had started as if each was separately touched by some specific recollection.

"Captain Ahab," said Tashtego, "that white whale must be the same that some call Moby Dick."

"Moby Dick?" shouted Ahab. "Do ye know the white whale then, Tash?"

"Does he fan-tail a little curious, sir, before he goes down?" said the Gay-Header deliberately.

"And has he a curious spout, too," said Daggoo, "very bushy, even for a parmacetty, and mighty quick, Captain Ahab?"

"And he have one, two, tree - oh! good many iron in him hide, too, Captain," cried Queequeg disjointedly, "all twiske-tee betwisk, like him - him - "

Figure 1.1: Fragment of the novel Moby Dick written by Herman Melville. We highlight 4 sequential patterns

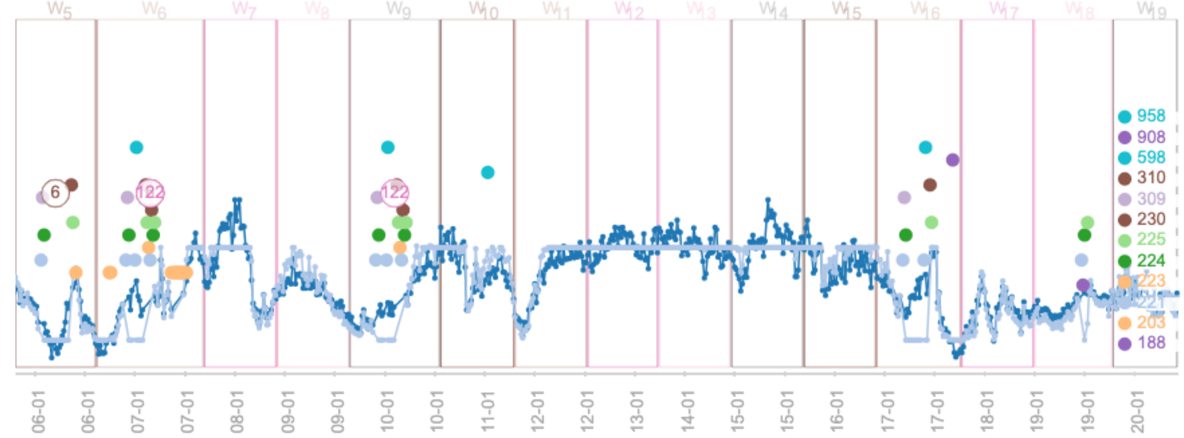


Figure 1.2: Data for 15 days of activity of a wind turbine. On the first day (W_5) pattern 6 occurs, meaning that events 203, 221, 224, 225, 309 and 310 co-occur and the wind turbine has stopped. On the second and fifth day (W_6 and W_9) pattern 122 occurs, meaning events 221, 224, 225, 230 and 310 co-occur and the turbine is remotely paused and re-started

1.1.2 Pattern Mining

The simplest type of pattern is a single event. Some events might occur more frequently, while some are rare. However, things get more interesting if we consider relations between pairs, triples or higher-order *patterns* of events. We illustrate this in Figure 1.1, where we highlight 4 *sequential patterns*. A sequential pattern is a sequence of events in a particular order. Recognising (*Moby*, *Dick*) automatically in this small fragment might seem silly. However, we must realise that for textual datasets and vision the human brain, with its 10^{10} neurons, is a sophisticated pattern mining machine that had a lifetime of continuous training. Moreover, for other datasets for which the human brain is not trained, such as in Figure 1.2, detecting patterns is hard. For example, pattern 122, consists of the co-occurring events 221=error, 224=pause, 225=run, 230 =auto-restart and 310=remote run, meaning a routine maintenance activity where the wind turbine is remotely paused and re-started.

To further illustrate the complexity of finding, or *mining*, patterns see Figure 1.3. Here we show a small fragment of the possible *frequent* sequential patterns found in Figure 1.1.

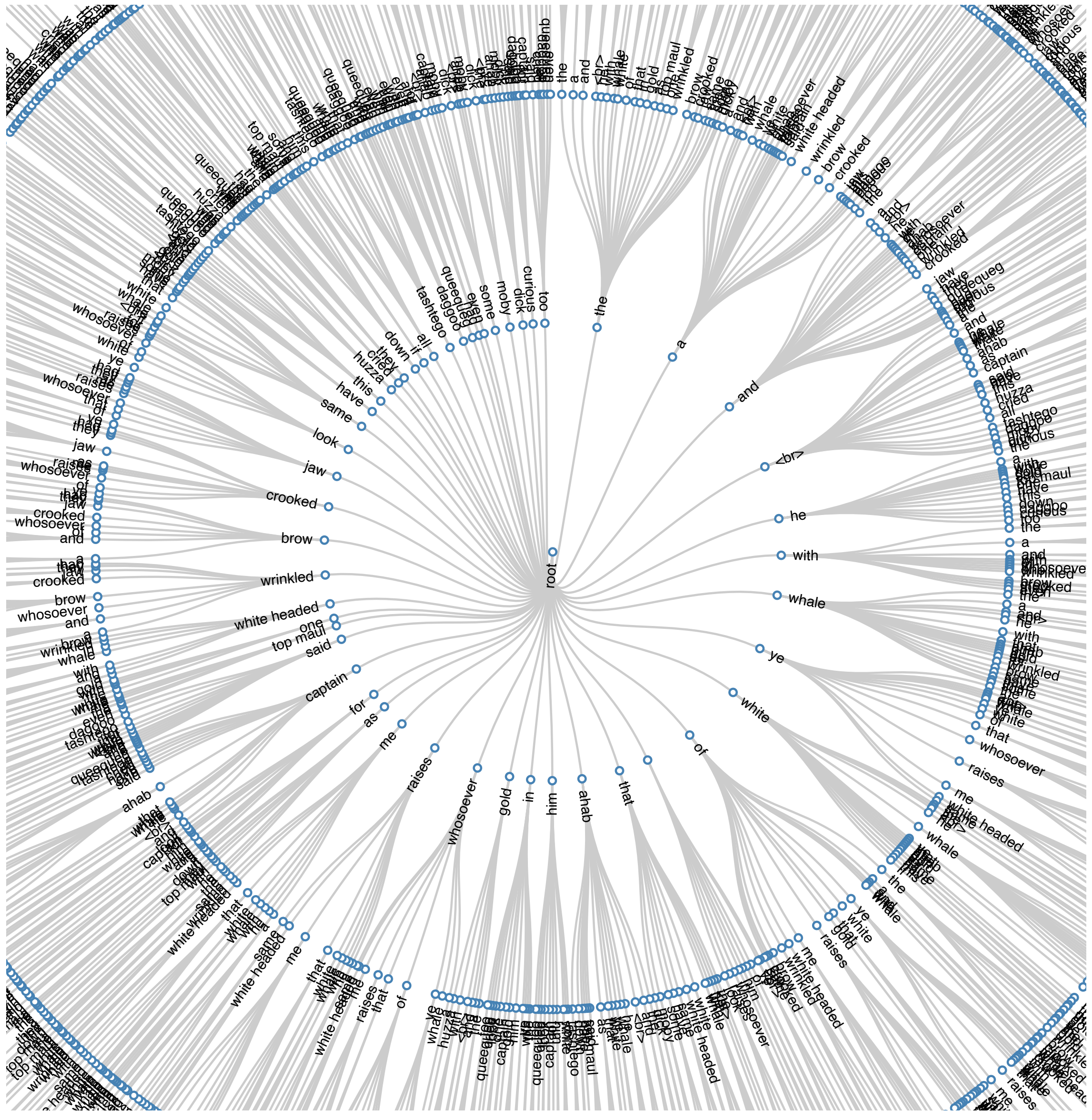


Figure 1.3: Frequent sequential patterns mined in the fragment of Moby Dick

That is, all sequences of at most 4 words that co-occur at least twice and where there are most 4 unrelated words, or gaps, between the first and the last word of the pattern. If we report the most frequent patterns, we find that (*white*, *whale*), but also (*the*, *and*) and (*the*, *the*) are top-ranked patterns.

In Chapters 2 and 3 we will focus on mining *cohesive* patterns. The algorithm that finds cohesive patterns and ranks (*Moby*, *Dick*) and (*wrinkled*, *brow*, *crooked*, *jaw*) on top, reports only a fraction of patterns, shown in Figure 1.4. We remark that (*wrinkled*, *brow*, *crooked*, *jaw*) is not frequent since it only occurs two times in the complete fragment. However, it is highly cohesive, meaning that if any of these 4 words occur, they occur together. In contrast, the pattern (*Captain*, *Ahab*) has a relatively low value for cohesion since *Captain* and *Ahab* also occur individually. However, we will see that we can still find patterns where words are often very near each other, using *quantile-based cohesion*. We find that by only evaluating different types of *cohesive* patterns and *pruning* the number of patterns using cohesion, we ignore many unsatisfactory patterns.

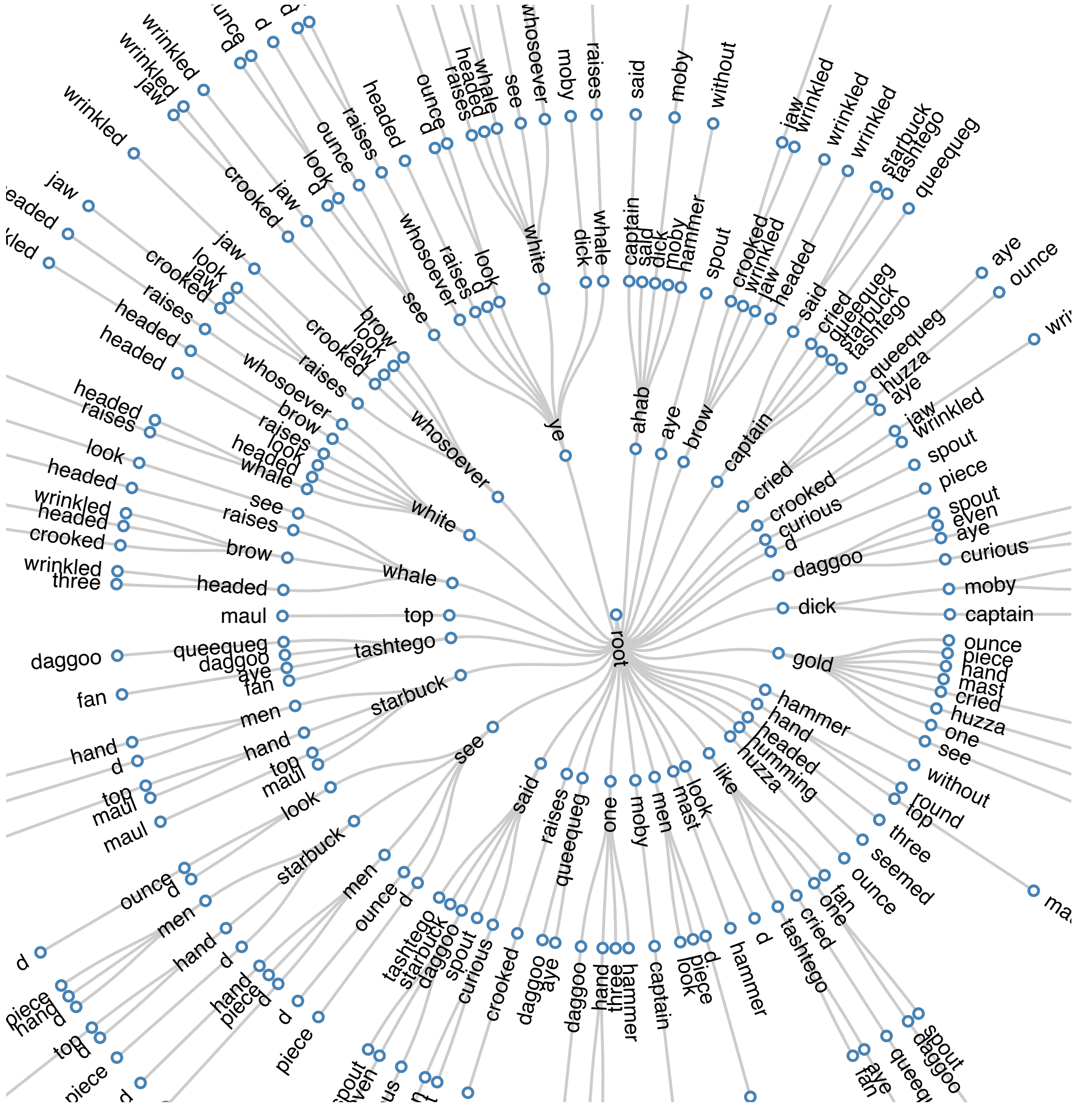


Figure 1.4: Cohesive sequential patterns mined in the fragment of Moby Dick

1.1.3 Itemsets and Episodes

Besides *sequential patterns*, researchers working on sequential data typically consider two other types of patterns: *itemsets* and general *episodes*. These three types of patterns differ in how strict we are about the order between pattern events. For an itemset, the order between pattern events, or items, is ignored. That is, if we match the itemset {*Moby*, *Dick*} to the input sequence, we ignore whether *Moby* or *Dick* comes first. A partially ordered episode is something between an itemset and a sequential pattern, where the order between events is only partially constrained. For example, suppose we have the pattern events *natural*, *selection* and *species*. These three terms often co-occur; however, if they occur, *natural* is always before *selection*, while *species* is sometimes before and sometimes after (*natural*, *selection*). In this scenario, we can define an episode with events {*natural*, *selection*, *species*} and a partial order *natural* → *selection*, instead of an itemset. An example of all three types of patterns is shown in Figure 1.5. We consider these patterns in more detail in Chapter 2.

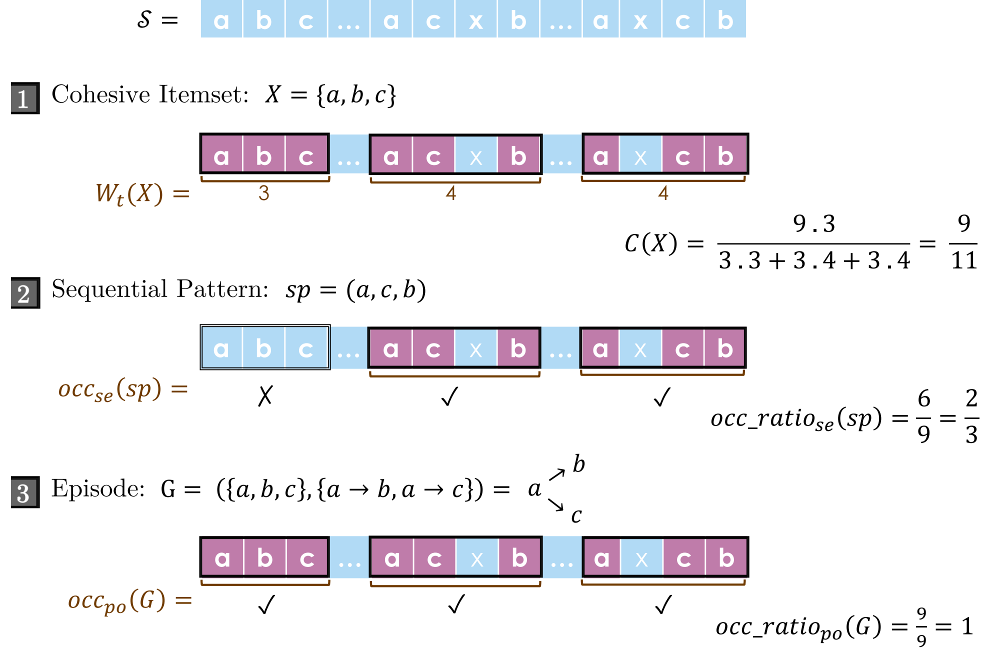


Figure 1.5: Example of a cohesive itemset, a sequential pattern and an episode

1.1.4 Association Rules

An important application of pattern mining is learning *association rules*. For example, based on the itemset {*Moby*, *Dick*} we can derive a rule *Moby* ⇒ *Dick*, meaning that if *Moby* occurs, *Dick* will occur with high *confidence*. For many applications, learning rules like this is of high importance. For instance, in the management of a wind turbine fleet, *predicting* the next value in a sequence using rules is of importance, for example, *high_windspeed*, *error_917* ⇒ *stop_turbine*. Likewise, we can also learn association rules and apply them for *anomaly detection*[1], i.e., if *high_windspeed*, *code_917* ⇒ *code_955* holds with high confidence, but if then *code_955* does not follow the occurrence of *high_windspeed* and *code_917* we report an anomaly. We consider association rules in more detail in Chapter 2.

[1] A chapter on our research on pattern-based anomaly detection is part of the PhD thesis of Vincent Vercruyssen (Feremans et al. 2019a).

1.2 Multi-label Classification

A second problem we study is how to automatically *label* or *classify* data by *learning* the logic or patterns associated with each label. For example, Wikipedia has millions of articles, and each article has one or more topics associated with it. A *classification algorithm* learns the logic behind each topic, or label, and automatically *predicts* topics for new documents. Likewise, commercial companies want to predict which set of advertisements is the most relevant for each user. Here each ad is a label, and each user has many features such as demographic attributes, sites visited etc. Many algorithms have been developed in the past decades to solve the task of classification. However, these algorithms require to learn a *model* for each label. The *time needed* to learn a model for each label can prevent many applications where we have extremely large data streams with millions of labels. Additionally, for online applications, new predictions are required within milliseconds. Only recently, researchers have begun developing algorithms tackling these stringent requirements.

1.2.1 Nearest Neighbours Classification

The *k-nearest neighbour algorithm* is a straightforward classification algorithm. In this type of instance-based learning or lazy learning, we assign a label to a new instance, based on the labels of the instances that are the most *similar*. We show an illustrative example in Figure 1.6. For example, using $k = 2$, it would make sense to predict that the label for the star-shaped example is the triangle. Using $k = 8$, we could predict two likely labels, the triangle since $\frac{5}{8}$ of neighbours have this label, and the circle since $\frac{3}{8}$ of neighbours have this label.

In Chapter 4 we study recent algorithms from information retrieval and recommender systems and developed a new algorithm for *multi-label classification* for datasets with an *extremely large number of labels*. We also use the labels of the *nearest neighbours* to make predictions. However, different from the illustrative example, each instance might have millions of labels and millions of features or dimensions. Like in Figure 1.6 we search the most similar documents (or users) in the labelled database and use their labels to make predictions. For instance, we can compare a new document to all other documents in the database and use the

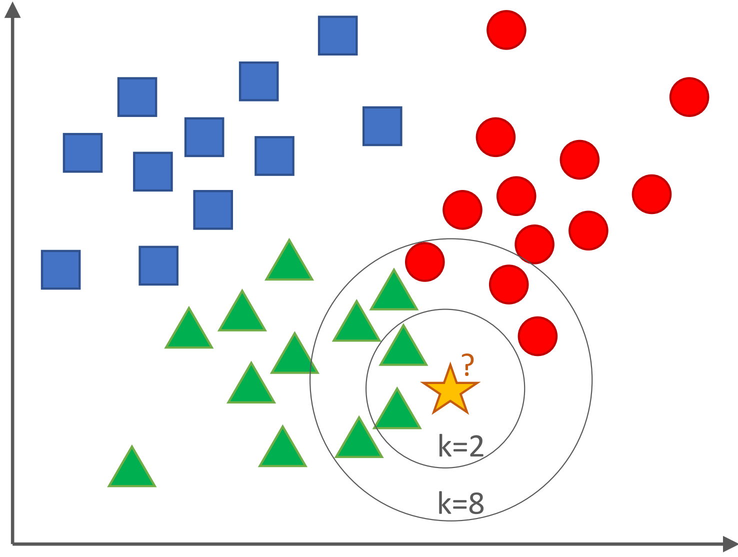


Figure 1.6: Illustrative example of k-nearest neighbours classification

known labels of the top-100 most *similar* documents. We measure the similarity using *cosine similarity*. In a large dataset, there are millions of attributes and labels possible. However, each individual instance typically has only a few hundred of nonzero features or labels. Therefore, measuring the cosine similarity is better suited than euclidean distance.

1.2.2 Top k-queries

A problem with the nearest neighbour algorithm is that we have to compare each new instance against possibly millions of instances in the labelled database. Therefore, until a decade ago, most researchers only focused on *approximate* algorithms on sparse datasets. However, recently, scientists have created algorithms that return the *exact* set of k-nearest neighbours and are as fast as high-quality approximate algorithms, by pruning or discarding instances that are known to be far from similar (Bayardo et al. 2007; Fontoura et al. 2011). We illustrate two common strategies for top-k query algorithms in Figure 1.7. Using *term-at-a-time*, we loop over each query term at a time. First, we retrieve all documents from an *inverted index*, and update the partial similarities between documents matching each term and the query. For example, if we search for *Soccer*, *Goal*, *Red* we first find documents 1 and 3 sharing *Soccer* and update the similarity between the query and these documents. Next, we do the same for documents matching *Goal*, and finally for documents matching *Red*. The main advantage of this strategy is that non-zero terms are not computed. Given that extreme datasets sometimes have only 0.0001% of non-zero values, this is efficient. In *document-at-a-time*, we consider each document sharing at least one term, and process them one by one. In our example, we first find document 1, and compute a similarity of 1.0. Next, we visit document 2, however, since its value for *Red* is 0.5, we can ignore other columns and *prune* this document from further comparison, as we can infer that document 1 is more similar. We have developed an algorithm

Training database. Query is: Soccer, Goal, Red

Doc ID	Soccer	Goal	Car	Road	...	Red	Football	Racing
1	1.0	1.0				1.0	✓	
2			1.0	1.0		0.5		✓
3	1.0	1.0					✓	
4			1.0	1.0				✓

Term-at-a-time traversal

Doc ID	Soccer	Goal	Car	Road	...	Red	Football	Racing
1	1.0	1.0				1.0	✓	
2			1.0	1.0		0.5		✓
3	1.0	1.0					✓	
4	1	2	1.0	1.0		3		✓

Document-at-a-time traversal for query with pruning

	Soccer	Goal	Red		
1	<Doc 1, 1.0>	<Doc 1, 1.0>	<Doc 1, 1.0>		Compute 1.0 similarity with Doc 1
3	<Doc 3, 1.0>	<Doc 3, 1.0>	<Doc 2, 0.5>	2	Prune Doc 2. Similarity is at most 0.5

Figure 1.7: Illustrative example of finding the nearest neighbours using term-at-a-time and document-at-a-time with pruning

for computing the top-k nearest neighbours that fuses both approaches, which we present in more detail in Chapter 4.

1.2.3 Item-based Collaborative Filtering

Collaborative filtering algorithms are used by popular information services to make suggestions for movies, music or products based on the history of liked (or clicked) items of each user. One of the most popular techniques is *item-based collaborative filtering*. The technique works by computing the similarities, column-wise, between all item pairs. If a user then prefers an item, for example, movie X, the recommendation system suggests similar items and produces well-known *"because you like X we recommend Y"* types of explanations. In multi-label classification, a similar approach does not exist. However, intuitively adopting this idea makes sense. For example, the overall correlation column-wise between *Car* and *Racing*, shown in Figure 1.7, can be used for recommending this label, i.e., *"because you have Car we recommend Racing"*. We have adapted item-based collaborative filtering and combined it with the k-nearest neighbour algorithm for multi-label classification of datasets with an extreme number of labels in Chapter 4.

1.3 Overview

This thesis contains the following chapters.

- In **Chapter 2**, we present FCI$_{seq}$. This algorithm finds all cohesive itemsets in a single event sequence efficiently. We extend this algorithm and find representative sequential patterns, dominant episodes and association rules for each cohesive itemset. We experimentally validate that, on an external synthetic benchmark, FCI$_{seq}$ is superior to state-of-the-art methods in discovering patterns hidden in a single data sequence. We also provide human-readable, qualitative examples of patterns using textual data sources, such as the tweets from Donald Trump.

- In **Chapter 3**, we define quantile-based cohesion. This measure is more robust to noise and suitable for ranking sequential patterns, with possible repeating items. We revisit the algorithm from Chapter 2, and propose the QCSP algorithm. QCSP is even faster than FCI$_{seq}$, and mines all quantile-base cohesive sequential patterns in both single, and multiple, event sequences.

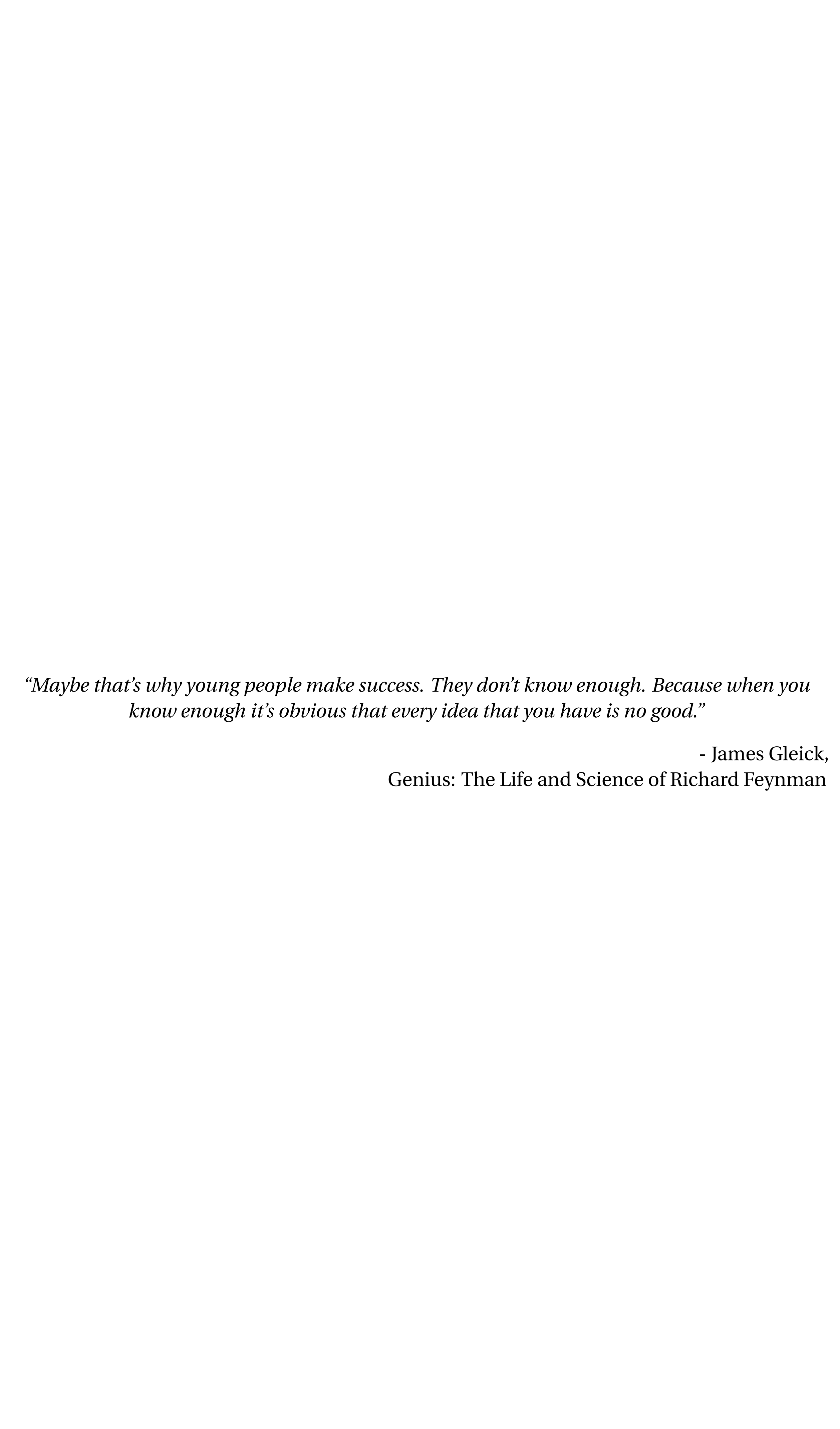
- In **Chapter 4**, we propose INSTANCEKNNFAST, an algorithm for fast k-nearest neighbour search. Given a test instance, we search for the nearest neighbours using term-at-a-time and document-at-a-time with a tight upper bound for pruning based on a partitioned index. Next, we compute all similarities between each feature and label column-wise and find the nearest labels for each feature, similar to item-based collaborative filtering. Finally, we introduce LCIF an extreme multi-label classification algorithm that combines label predictions based on the instance- and feature-based neighbours. We experimentally validate the accuracy and efficiency compared to state-of-the-art methods on real-world datasets with millions of examples, features and labels.

- In **Chapter 5** we summarise the main contributions of this thesis and the potential for future research.

1.4 Open-source Code

We have provided public open-source repositories for each algorithm. This facilitates future researchers in comparing with our methods and enables applications.

- The source code in Java for FCI$_{seq}$, including experimental scripts, datasets and an online demo, is available at https://bitbucket.org/len_feremans/fci_public. FCI$_{seq}$ is also part of the SPMF library available at https://www.philippe-fournier-viger.com/spmf.
- The source code in Java for QCSP, including experimental scripts and datasets, is available at https://bitbucket.org/len_feremans/qcsp_public. QCSP is also part of the SPMF library.
- The source code in C++ for LCIF, including large datasets and experimental scripts, is available at https://bitbucket.org/len_feremans/lcif. Extreme datasets and benchmark results for state-of-the-art methods are made publicly available by Bhatia et al. (2016).
- A chapter on PBAD, an algorithm for pattern-based anomaly detection, will be available in the doctoral thesis of Vincent Vercruyssen (Feremans et al. 2019a). The source code in Python is available at https://bitbucket.org/len_feremans/pbad.
- We have not included a chapter on the framework for pattern mining and anomaly detection in multi-dimensional time series and event logs (Feremans et al. 2019b). However, the source code in Java for the corresponding tool, TIPM, is available at https://bitbucket.org/len_feremans/tipm_pub.



“Maybe that’s why young people make success. They don’t know enough. Because when you know enough it’s obvious that every idea that you have is no good.”

- James Gleick,
Genius: The Life and Science of Richard Feynman

CHAPTER 2

Efficiently Mining Cohesive Patterns and Rules in Sequences

Discovering patterns in long event sequences is an important data mining task. Traditionally, research focused on frequency-based quality measures that allow algorithms to use the anti-monotonicity property to prune the search space and efficiently discover the most frequent patterns.

In this Chapter[1], we step away from such measures and evaluate patterns using cohesion — a measure of how close to each other the items making up the pattern appear in the sequence on average. We tackle the fact that cohesion is not an anti-monotonic measure by developing an upper bound on cohesion in order to prune the search space. By doing so, we are able to efficiently unearth rare, but strongly cohesive, patterns that existing methods often fail to discover. Furthermore, having found the occurrences of cohesive itemsets in the input sequence, we use them to discover the most dominant sequential patterns and partially ordered episodes, without going through the computationally expensive candidate generation procedures typically associated with sequence and episode mining.

Experiments show that our method efficiently discovers important patterns that existing state-of-the-art methods fail to discover.

[1]This chapter is based on work published in the Data Mining and Knowledge Discovery journal as "Efficiently mining cohesion-based patterns and rules in event sequences" by Boris Cule, Len Feremans and Bart Goethals (Cule et al. 2019).

2.1 Introduction

Pattern discovery in sequential data is a well-established field in data mining. The earliest attempts focused on the setting where data consisted of many (typically short) sequences, where a pattern was defined as a (sub)sequence that re-occurred in a high enough number of such input sequences (Srikant and Agrawal 1996). The first attempt to identify patterns in a single long sequence of data was proposed by Mannila et al. (1997). The presented WINEPI method uses a sliding window of a fixed length to traverse the sequence, and a pattern is then considered frequent if it occurs in a high enough number of these sliding windows. Mannila et al. (1997) describes algorithms for mining various pattern types — parallel episodes, which are essentially itemsets (with the possibility of some items re-occurring), serial episodes, which are equivalent to sequential patterns, and general episodes, which are partially ordered patterns. Here, we use the term *pattern* when we talk of any pattern type and use more specific terms when appropriate.

An often-encountered critique of this method is that the obtained frequency is not an intuitive measure since it does not correspond to the actual number of occurrences of the pattern in the sequence. For example, given sequence $axbcdbya$, and a sliding window length of 3, the frequency of itemset $\{a,b\}$ will be equal to 2, as will the frequency of itemset $\{c,d\}$. However, itemset $\{a,b\}$ occurs twice in the sequence, and itemset $\{c,d\}$ just once, and while the method is motivated by the need to reward c and d for occurring right next to each other, the reported frequency values remain difficult to interpret.

Laxman et al. (2007) attempted to tackle this issue by defining the frequency as the maximal number of non-overlapping minimal windows of the pattern in the sequence. In this context, a minimal window of the pattern in the sequence is defined as a contiguous subsequence of the input sequence that contains the pattern, such that none of its smaller contiguous subsequences also contains the pattern. However, while the method uses a relevance window of a fixed length, and disregards all minimal windows that are longer than the relevance window, the length of the minimal windows that do fit into the relevance window is not taken into account at all. For example, given sequence $axyzbcd$, with a relevance window larger than 4, the frequency of both itemset $\{a,b\}$ and itemset $\{c,d\}$ would be equal to 1, which would not reflect that the items making up the second pattern occur much closer to each other than those making up the first pattern.

Cule et al. (2014) proposed an amalgam of the two approaches (MARBLES$_W$), defining the frequency of a pattern as the maximal sum of weights of a set of non-overlapping minimal windows of the pattern, where the weight of a window is defined as the inverse of its length. However, this method, too, struggles with the interpretability of the proposed measure. For example, given sequence $axbcdbya$ and a relevance window larger than 3, the frequency of $\{a,b\}$ would be $2/3$, while the frequency of $\{c,d\}$ would be $1/2$. Additionally, as the input sequence grows longer, the sum of these weights will grow, and the defined frequency can take any real positive value, giving the user no idea how to set a sensible frequency threshold.

All the techniques mentioned above use frequency measures that satisfy the so-called APRIORI property (Agrawal and Srikant 1994). This property implies that the frequency of a pattern is never smaller than the frequency of any of its superpatterns (in other words, frequency is an *anti-monotonic* measure). While this property is computationally very desirable, since large candidate patterns can be generated from smaller frequent patterns, the undesirable side effect is that larger patterns, which are often more useful to the end-users, will never be ranked higher than any of their subpatterns. On top of this, all these methods focus solely on how often certain items occur near each other and do not take occurrences of these items far away

from each other into account. Consequently, if two items occur frequently, and through pure randomness often occur near each other, they will form a frequent itemset, even though they are, in fact, in no way correlated.

In another work, Cule et al. (2009) proposed a method that steps away from anti-monotonic quality measures, and introduce a new interestingness measure that combines the coverage of an itemset with its cohesion. Cohesion is defined as a measure of how near each other the items making up an interesting itemset occur on average. However, the authors defined the coverage of an itemset as the sum of frequencies of all items making up the itemset, which results in a massive bias towards larger patterns instead. Furthermore, this allows for a very infrequent item making its way into an interesting itemset, as long as all other items in the itemset are very frequent and often occur near the infrequent item. As a result, the method is not scalable for any sequence with a large alphabet of items, which makes it unusable in most realistic data sets.

In this work, we use the cohesion introduced by Cule et al. (2009) as a single measure to evaluate cohesive itemsets. We consider itemsets as potential candidates only if each individual item contained in the itemset is frequent in the dataset. This allows us to filter out the infrequent items at the very start of our algorithm, without missing out on any cohesive itemsets. However, using cohesion as a single measure brings its own computational problems. First of all, cohesion is not an anti-monotonic measure, which means that a superset of a non-cohesive itemset could still prove to be cohesive. However, since the search space is exponential in the number of frequent items, it is impossible to evaluate all possible itemsets. We solve this by developing an upper bound on the maximal possible cohesion of all itemsets that can still be generated in a particular branch of the depth-first-search tree. This bound allows us to prune large numbers of potential candidate itemsets, without having to evaluate them. Furthermore, we present an efficient method to identify the minimal windows that contain a particular itemset, which is necessary to evaluate its cohesion.

Having discovered the cohesive itemsets, we move on to the problem of finding cohesive sequential patterns and partially ordered episodes. Due to the combinatorial explosion, the number of possible candidate patterns that potentially must be generated and evaluated quickly becomes prohibitive for typical sequential pattern or episode mining algorithms. We avoid this problem by taking the already discovered cohesive itemsets as a starting point, and then only evaluating those total and partial orders that actually occur in the data. More concretely, each discovered minimal window of an itemset represents one or more possible sequential patterns, so all we need to do is go through the list of such windows and update the frequency of each encountered sequential pattern. Once that is done, we report the representative sequential patterns and the dominant episodes (which we obtain simply by intersecting the discovered sequential patterns).

Finally, we show that cohesive itemsets can also form a basis for mining association rules between items. A cohesive itemset tells us that the items forming the itemset occur close to each other on average, but in some cases, it may happen that an occurrence of a particular item implies, with a high probability, that some other items will occur nearby, but the implication may not hold in the other direction. For example, it is possible that item a always occurs near item b, but not vice versa (i.e., there may be many occurrences of item b far from any occurrence of item a). In this case, itemset $\{a, b\}$ would not be very cohesive, but an association rule $\{a\} \Rightarrow \{b\}$ would still be informative. We present an efficient algorithm that generates such rules starting from the discovered cohesive itemsets, using a cohesion-based confidence measure. Unlike the traditional frequency-based approaches, which need all the frequent itemsets to be generated before the generation of association rules can begin, we are able to generate rules in parallel with the interesting itemsets. Furthermore, we present an important mathematical

property that allows us to very quickly compute the confidence of most association rules, without having to revisit the data at all.

Our experiments show that our method discovers important patterns that existing methods struggle to rank highly while dismissing obvious patterns consisting of items that co-occur frequently but are not at all correlated. We further show that we achieve these results quickly, thus demonstrating the efficiency of our algorithm.

The remainder of this chapter is organised as follows. In Section 2.2 we formally describe the problem setting and define the patterns we aim to discover. In Section 2.3 we present how to mine cohesive itemsets efficiently. In Section 2.4 we present how we mine representative sequential patterns based on cohesive itemsets, in Section 2.5 how to mine dominant episodes and in Section 2.6 how to mine association rules. We discuss how to set parameters for our method in Section 2.7. In Section 2.8 we present a thorough experimental evaluation of our method, in comparison with a number of existing state-of-the-art methods. Finally, we present an overview of the most relevant related work in Section 2.9, before summarising our main conclusions in Section 2.10.

2.2 Problem Setting

Definition 2.1 (Event sequence)**.** *The dataset consists of a single event sequence* $\mathcal{S} = (e_1, \ldots, e_n)$*. Each event* e_k *is represented by a pair* (i_k, t_k)*, with* i_k *an event type (coming from the domain of all possible event types) and* t_k *a (integer) timestamp. For any* $1 < k \le n$*, it holds that* $t_k > t_{k-1}$*.*

We use $\mathcal{S}_{[j,l]}$, with $j < l$, to denote subsequence $(e_j, e_{j+1}, \ldots, e_{l-1}, e_l)$. The *length* (or duration) of sequence $\mathcal{S}$, denoted $|\mathcal{S}|$, is equal to $t_n - t_1 + 1$, and the length of a subsequence $\mathcal{S}_{[j,l]}$, denoted $|S_{[j,l]}|$, is equal to $t_l - t_j + 1$. For simplicity, we omit the timestamps from our examples, and write sequence $(e_1, \ldots, e_n)$ as $i_1 \ldots i_n$, implicitly assuming that the timestamps are consecutive integers starting with 1. In what follows, we refer to event types as *items*, and sets of event types as *itemsets*.

2.2.1 Frequent Cohesive Itemsets

Definition 2.2 (Itemset support)**.** *For an itemset* $X = \{i_1, \ldots, i_m\}$*, we denote the set of occurrences of items making up* X *in a sequence* $\mathcal{S}$ *by*

$$N(X) = \{t \mid (i,t) \in \mathcal{S}, i \in X\}.$$

We define the support of X *in an input sequence* $\mathcal{S}$ *as the number of occurrences of* $i \in X$ *in* $\mathcal{S}$*,*

$$support(X) = |N(X)|$$

.

Given a user-defined support threshold θ, we say that an itemset X is *frequent* in a sequence $\mathcal{S}$ if for each $i \in X$ it holds that $support(i) \ge \theta$. We remark that this definition of support is quite different from the traditional definition using sliding windows, that is, the support will monotonically increase for longer patterns.

To evaluate the cohesiveness of an itemset X in a sequence $\mathcal{S}$, we must first identify minimal occurrences of the itemset in the sequence. More specifically, for each occurrence of an item in X, we will look for the minimal window within $\mathcal{S}$ that contains that occurrence and the entire itemset X.

Definition 2.3 (Minimal window). *Given a timestamp t, such that $(i, t) \in \mathcal{S}$ and $i \in X$, we define the size of a minimal occurrence of X around t as*

$$W_t(X) = \min\{|\mathcal{S}_{[s,e]}| \mid t_s \leq t \leq t_e \ \wedge \ \forall i \in X \ \exists (i, t') \in \mathcal{S} : t_s \leq t' \leq t_e\}.$$

We define the size of the average minimal occurrence of X in $\mathcal{S}$ as

$$\overline{W}(X) = \frac{\sum_{t \in N(X)} W_t(X)}{|N(X)|}.$$

If $|N(X)| = 0$, we define $\overline{W}(X) = 0$.

Remark that we compute the average for all minimal windows, unlike frequency-based methods that define a maximal window size (or relevance window). For example, consider itemset $\{a, b, c\}$ and sequence $\mathcal{S} = abc \ldots abc \ldots abc \ldots a \ldots b \ldots c$. If we have a limit on the window size, we would only consider nearby occurrences. However, for the last 3 items, the minimal window is large and we want to reflect this is in our ranking of patterns. That is, we want to distinguish between pattern occurrences that are always nearby, pattern occurrences that are often nearby and pattern occurrences that are sometimes nearby but, more often than not, far away.

Definition 2.4 (Cohesion). *We define the cohesion of itemset X in a sequence $\mathcal{S}$ as*

$$C(X) = \frac{|X|}{\overline{W}(X)}.$$

If $|X| = 0$, we define $C(X) = 1$.

Given a user-defined cohesion threshold *min_coh*, we say that an itemset X is *cohesive* in a sequence $\mathcal{S}$ if it holds that $C(X) \geq min_coh$. Note that the cohesion is higher if the minimal occurrences are smaller. Furthermore, a minimal occurrence of itemset X can never be smaller than the size of X, so it holds that $C(X) \leq 1$. If $C(X) = 1$, then every single minimal occurrence of X in $\mathcal{S}$ is of length $|X|$. The cohesion of a single item is always equal to 1. For singletons, therefore, our approach is equivalent to discovering frequent items. Since we are interested in mining frequent cohesive itemsets, we will from now on consider only itemsets consisting of 2 or more items.

A limitation of the current definition of an event sequence $\mathcal{S}$ is that we do not allow multiple items at the same timestamp. If we allowed multiple items, the minimal window could be smaller than the size of the pattern, i.e., if all items of X occur at the same timestamp and as a consequence the cohesion would not be within $[0, 1]$. We could bypass this limitation by adjusting the definition of a minimal window and add the constraint that all items should occur at different timestamps. However, for simplicity, we will assume that items are not overlapping in the remainder of this chapter.

Problem 2.1 (Frequent cohesive itemset). *Given a frequency threshold θ, a threshold min_coh and an optional parameter max_size, find all frequent cohesive itemsets X in a sequence $\mathcal{S}$, where*

1. $1 < |X| < max_size$,
2. $\forall i \in X : support(i) \geq \theta$,
3. $C(X) \geq min_coh$.

Cohesion is not an *anti-monotonic measure.* In other words, a superset of a non-cohesive itemset could turn out to be cohesive. While this allows us to eliminate bias towards smaller patterns, it also brings additional computational challenges which will be addressed in Section 2.3.

Example 2.1. For example, given sequence $\mathcal{S} = abc\ldots acb\ldots bac$ the minimal window size for itemsets $\{a,b\}$ are $2,2,3,3,2,2$ and $C(a,b) = 12/14 = 6/7$. Likewise, for other itemsets of size 2, $C(\{a,c\}) = C(\{b,c\}) = 6/7$. However, for itemset $\{a,b,c\}$ the cohesion is 1, since all minimal window sizes are 3, that is $C(\{a,b,c\}) = 9\cdot 3/9\cdot 3$. □

Finally, note that the definition of cohesion makes it potentially sensitive to *outliers.* For example, if two items a and b are strongly correlated, just one random occurrence of an a far from any b could very negatively affect the cohesion of itemset $\{a,b\}$ since we take the average of all minimal window sizes, which is not robust to outliers. In data known to contain outliers, a user could avoid this risk by dividing the input sequence into segments (overlapping or not), and then mining cohesive itemsets in each segment. A single outlier would then affect the value of a pattern in one segment, but the pattern would still be ranked highly in all other segments where it is present. We remark that an interestingness measure robust to outliers is introduced in Chapter 3.

2.2.2 Representative Sequential Patterns

In this section, we show how frequent cohesive itemsets can be used as a basis for discovering representative sequential patterns in the data while avoiding computationally expensive candidate generation steps typical for sequential pattern mining. Since we use cohesive itemsets as a starting point, we are only able to find sequential patterns in which no event can re-occur. This is an inherent cost when choosing for itemset mining instead of tackling the additional complexity of mining sequential pattern directly. Naturally, this does mean our method is not suitable for data where important patterns often contain multiple instances of the same item (such as, for example, DNA sequences, where the number of distinct items is limited). We define the necessary concepts below and present our mining algorithm in Section 2.4.

Definition 2.5 (Sequential pattern)**.** *Given a frequent cohesive itemset $X = \{i_1,\ldots,i_m\}$, we can generate a sequential pattern $sp = s_1 s_2 \ldots s_m$ from X if $s_k \in X$, for $k \in \{1,\ldots,m\}$ and $\forall i,j \in \{1,\ldots,m\} : i \neq j \Rightarrow s_i \neq s_j$. In this case, we call X the* underlying itemset *of sp, denoted X_{sp}.*

Having found the minimal occurrences of a frequent cohesive itemset, our goal here is to find in which order the items most often occur in those minimal occurrences. Any such frequently occurring order uniquely defines a sequential pattern. Given a sequence $\mathcal{S} = (e_1,\ldots,e_n)$, we say sequential pattern $sp = s_1 s_2 \ldots s_m$ *occurs* in $\mathcal{S}$, denoted $sp \sqsubseteq \mathcal{S}$, if there exist integers $1 \leq k_1 < \ldots < k_m \leq n$, such that $s_j = e_{k_j}$ for $j \in \{1,\ldots,m\}$.

Definition 2.6 (Sequential pattern occurrence ratio)**.** *We define the set of occurrences of a Sequential Pattern sp in an input sequence $\mathcal{S}$ as*

$$occ_{se}(sp) = \{t \in N(X_{sp}) \mid \exists j,k : j \leq t \leq k, sp \sqsubseteq \mathcal{S}_{[j,k]}, |\mathcal{S}_{[j,k]}| = W_t(X_{sp})\}.$$

We define the occurrence ratio of sp within the occurrences of X_{sp} as

$$occ_ratio_{se}(sp) = \frac{|occ_{se}(sp)|}{|N(X_{sp})|}.$$

Note that this formal definition corresponds to the above intuition, and says that a sequential pattern *sp* is considered to occur at timestamp t if its occurrence *around* timestamp t is as long as the minimal occurrence of its underlying itemset X_{sp} around timestamp t. Intuitively, the occurrence ratio of a sequential pattern *sp* measures the likelihood that the items making up its underlying itemset X_{sp} appear in a minimal occurrence of X_{sp} in the order defined by *sp*.

Definition 2.7 (Representative sequential pattern)**.** *Given a user-defined minimal occurrence ratio threshold min_or, we say a sequential pattern sp is representative for X_{sp} in sequence $\mathcal{S}$ if $occ_ratio_{se}(sp) \geq min_or$.*

It is interesting to note that multiple sequential patterns can be contained within the same minimal occurrence of their underlying itemset X_{sp}.

Example 2.2. For example, given itemset $X = \{a, b, c, d\}$ and input sequence $\mathcal{S} = abcbd$, we can see that the entire sequence is a minimal occurrence of X, and it contains occurrences of sequential patterns $abcd$ and $acbd$. In this case, both $abcd$ and $acbd$ would have an occurrence ratio of 1. Moreover, a minimal occurrence of an itemset at a given timestamp is not necessarily unique. For example, in an input sequence $abcda$, at timestamp 2, the minimal occurrence of itemset $\{a, b, c, d\}$ could be both $S_{[1,4]}$ and $S_{[2,5]}$. In this case, sequential patterns $abcd$ and $bcda$ both occur at timestamp 2, and at timestamps 3 and 4, but only one of them occurs at timestamps 1 and 5. As a result, in this example, both $abcd$ and $bcda$ would have an occurrence ratio of 4/5, or 0.8. □

2.2.3 Dominant Episodes

The approach described above can also be extended to finding dominant partial orders, or episodes, within the occurrences of frequent cohesive itemsets. In episode mining literature, an episode is typically represented by a directed acyclic graph $G = (V(G), E(G))$. Here $V(G) = (v_1, \ldots, v_m)$ is the set of nodes, where each node v_i corresponds to an item, and $E(G)$ is the set of directed edges between items, where an edge (v_i, v_j) means that item v_i occurs before item v_j in any occurrence of G. We say an episode G is *transitive closed* if for any $v_i, v_j, v_k \in V(G)$ it holds that if $(v_i, v_j) \in E(G)$ and $(v_j, v_k) \in E(G)$ then $(v_i, v_k) \in E(G)$. To avoid redundancy, we consider episodes and their transitive closure as equivalent. In our examples and illustrations we omit the edges whose presence is implied by other edges.

Formally, given a sequence $\mathcal{S} = (e_1, \ldots, e_n)$ and an episode G we say that G *occurs* in $\mathcal{S}$, denoted $G \preceq \mathcal{S}$, if there exists an injective function f mapping each node $v_i \in V(G)$ to an index in $\{1, \ldots n\}$, such that $v_i = e_{f(v_i)}$ for $i = \{1, \ldots, m\}$, and that if there is an edge (v_i, v_j) in $E(G)$, then we must have $t_{f(v_i)} < t_{f(v_j)}$. As with sequential patterns, we use frequent cohesive itemsets as a starting point (as a result, no item can re-occur in any episode we discover).

Definition 2.8 (Episode)**.** *Given an itemset X, we can generate episode G from X if $V(G) = X$. Again, in such a case, we call X the underlying itemset of G, denoted X_G. For two episodes, G_1 and G_2, with $X_{G_1} = X_{G_2}$, we say G_1 is a* subepisode *of G_2, denoted $G_1 \subseteq G_2$, if $E(G_1) \subseteq E(G_2)$. Given a sequential pattern sp, we denote its equivalent episode G_{sp}, whereby $V(G_{sp}) = X_{sp}$, and $E(G_{sp})$ imposes a total order on the items as defined by sp.*

Definition 2.9 (Episode occurrence ratio)**.** *We define the set of occurrences of G in an input sequence $\mathcal{S}$ as*

$$occ_{po}(G) = \{t \in N(X) \mid \exists j, k : j \leq t \leq k, G \preceq \mathcal{S}_{[j,k]}, |\mathcal{S}_{[j,k]}| = W_t(X_G)\}.$$

We define the occurrence ratio of G within the occurrences of X_G as

$$occ_ratio_{po}(G) = \frac{|occ_{po}(G)|}{|N(X_G)|}.$$

Intuitively, $occ_ratio_{po}(G)$ measures the likelihood that the items making up the underlying itemset X_G will appear in a minimal occurrence of X_G in the order defined by G. Naturally, given an itemset X, the episode with the highest occurrence ratio will always be the itemset itself (i.e., the episode with $V(G) = X$ and $E(G) = \emptyset$), since it will, per definition, occur in every minimal occurrence of X. In fact, the occurrence ratio is an anti-monotonic measure, in the sense that if $G_1 \subseteq G_2$ then $occ_ratio_{po}(G_1) \geq occ_ratio_{po}(G_2)$. As a result, the episodes with the highest occurrence ratio will arguably be the least interesting ones (those with no edges will score the highest, followed by those with a single edge, etc.). Therefore, we choose to search for interesting episodes differently from our approach outlined for sequential patterns before. We propose to discover *exactly one* episode for each frequent cohesive itemset, namely, the most specific episode (i.e., as many edges as possible) that describes a sufficient number of occurrences of the itemset itself. We conclude this section by formalising the above concepts.

Definition 2.10 (Intersecting episode)**.** *Given a user-defined minimal occurrence ratio threshold min_por and a set of episodes $\{G_1,\ldots,G_n\}$, with $V(G_i) = X$ for $i \in \{1,\ldots,n\}$, we define the intersecting episode of $\{G_1,\ldots,G_n\}$ as the episode with nodes $V(\bigcap_{i\in\{1,\ldots,n\}} G_i) = X$ and edges $E(\bigcap_{i\in\{1,\ldots,n\}} G_i) = \bigcap_{i\in\{1,\ldots,n\}} E_i$.*

We can generate an intersection episode of an itemset X by taking the intersection of the top-ranked sequential patterns (with respect to the occurrence ratio) generated from X. For example, assuming $sp_1 = abc$ and $sp_2 = acb$ have the highest occurrence ratio, we compute the intersecting episode of G_{sp_1} and G_{sp_2} as $G(\{a,b,c\},\{a \to b, a \to c\})$.

Definition 2.11 (Dominant episode)**.** *Given a frequent cohesive itemset X, and a set of sequential patterns $\{sp_1,\ldots,sp_n\}$ where $X_{sp_k} = X$, we define the intersecting episode, w.r.t. a threshold t as*

$$dpo_{X,t} = \bigcap_{\substack{X_{sp}=X \\ occ_ratio_{se}(sp)\geq t}} G_{sp} = G(X, \{\bigcap_{i\in\{1,\ldots,k\}} E(G_{sp_k}) \mid occ_ratio_{se}(sp_k) \geq t\})$$

where t is a threshold on the the occurrence ratio of the sequential patterns. The dominant episode w.r.t. min_por is created by taking the highest value of this threshold, denoted t_{max}, such that $occ_ratio(dpo_{X,t_{max}}) \geq min_por$ while minimising the number of sequential patterns k.

$$t_{max} = \max_{t\in[0,1]} \{occ_ratio_{po}(dpo_{X,t}) \geq min_por\}.$$

In other words, we compute the dominant episode of an itemset X by taking the intersection of the minimal necessary number (as guaranteed by the value t_{max} above) of the sequential patterns generated from X. In practice, as will be discussed in detail in Section 2.5, we keep on adding the top sequential patterns until the occurrence ratio of the resulting intersection becomes high enough to satisfy the *min_por* threshold. Here, too, our main goal is to provide a straightforward, quick, method, based on cohesive itemsets and avoiding the high computational cost of generating exponentially many partial orders and then comparing them all to each other using some interestingness measure, that can still produce very satisfactory results. However, due to the simplicity of the approach, there is an inherent risk that some interesting partial orders may be missed.

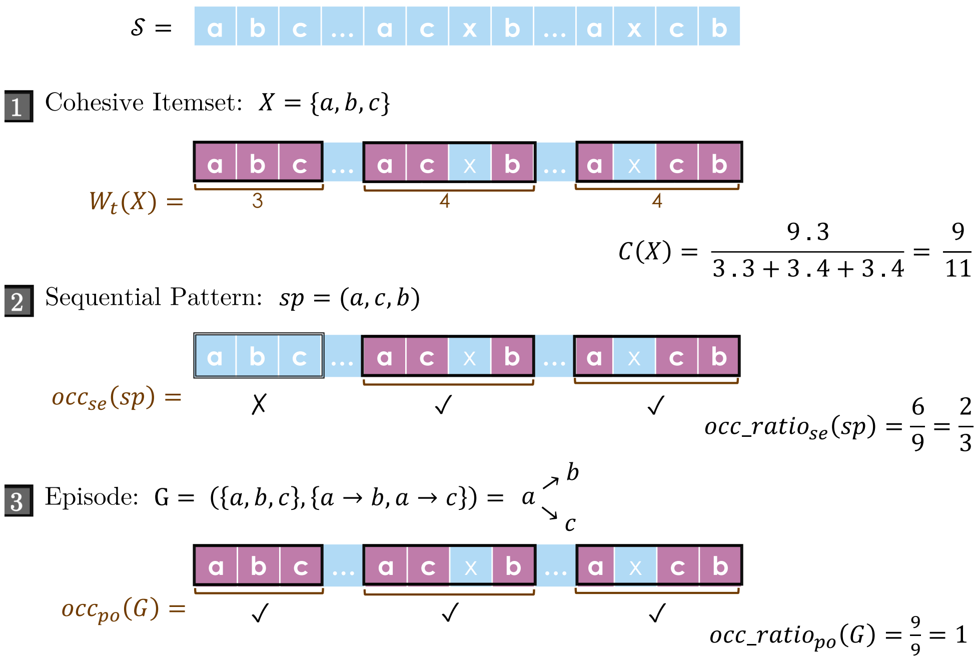


Figure 2.1: Example of a cohesive itemset, a representative sequential pattern and a dominant episode

Example 2.3. We illustrate the above concepts with a simple example, shown in Figure 2.1. Consider input sequence $\mathcal{S} = abc \dots acxb \dots axcb$. In this case, $X = \{a, b, c\}$ is a cohesive itemset with cohesion $C(X) = 3 \cdot 9/33 = 9/11$. Sequential pattern $sp = acb$ has an occurrence ratio $occ_ratio_{se}(acb) = \frac{6}{9} = \frac{2}{3}$. For generation of the dominant episode G we would start with sequential patterns $sp_1 = acb$, with $occ_ratio_{se}(sp_1) = \frac{2}{3}$, and $sp_2 = abc$, with $occ_ratio_{se}(sp_2) = \frac{1}{3}$. Then we convert both sequential patterns to their equivalent episodes $E(G_{sp_1}) = \{a \to c, c \to b, a \to b\}$ and $E(G_{sp_2}) = \{a \to b, b \to c, a \to c\}$ where we added one edge to both episodes by computing the transitive closure. With a value of $t_{max} \geq 1/3$, we generate the intersecting episode $E(G_{sp_1}) \cap E(G_{sp_2}) = \{a \to b, a \to c\}$, which has an occurrence ratio of 1. If the min_por threshold was set higher than 2/3, G would be the dominant episode of itemset X, otherwise G_{sp_1} would be the dominant episode with $t_{max} = 2/3$. Remark that the notation shown in Figure 2.1 indicates that event a occurs first, followed by events b and c in unspecified order. □

2.2.4 Association Rules

Finally, we are also able to use the frequent cohesive itemsets as a basis for discovering association rules in this setting. The aim is to generate rules of the form *if X occurs, Y occurs nearby*, where $X \cap Y = \emptyset$ and $X \cup Y$ is a frequent cohesive itemset. We denote such a rule by $X \Rightarrow Y$, and we call X the antecedent of the rule and Y the consequent of the rule. Intuitively, the closer Y occurs to X on average, the higher the value of the rule.

Definition 2.12 (Extended average window size)**.** *To compute the confidence of the rule $X \Rightarrow Y$, we must use the average length of minimal windows containing $X \cup Y$, but only from the point of view of items making up itemset X. We therefore define this new average as*

$$\overline{W}(X, Y) = \frac{\sum_{t \in N(X)} W(X \cup Y, t)}{|N(X)|}.$$

Definition 2.13 (Rule confidence)**.** *We define the confidence of the association rule as*

$$c(X \Rightarrow Y) = \frac{|X \cup Y|}{\overline{W}(X,Y)}.$$

A rule $X \Rightarrow Y$ is considered confident if its confidence exceeds a given threshold, min_conf, i.e., if $c(X \Rightarrow Y) \geq min_conf$. Note that if Y always occurs right next to a fully cohesive itemset X, the average minimal window will be exactly equal to the size of itemset $X \cup Y$, and the confidence of the rule $X \Rightarrow Y$ will be equal to 1. Intuitively, the inverse of the confidence tells us how large the average minimal window containing $X \cup Y$ for each occurrence of X is, compared to the minimal possible window (the size of $X \cup Y$), as illustrated in the example below.

Like the cohesion of an itemset, the confidence of an association rule can also be sensitive to *outliers* in the data. Again, if two items a and b are strongly correlated, just one random occurrence of an a far from any b could negatively affect the confidence of rule $\{a\} \Rightarrow \{b\}$. However, in this case, the confidence of rule $\{b\} \Rightarrow \{a\}$ would not be affected. Once again, to reduce the effect of outliers, data could be divided into segments and rules discovered to hold in many segments could be considered to be the most reliable. An interestingness measure robust to outliers is introduced in Chapter 3.

Example 2.4. Consider, for example, input sequence $\mathcal{S} = abcabdbxyzb$. We can see that every a has a b right next to it, but not the other way around. As a result, itemset $X = \{a, b\}$ has a cohesion of $C(X) = 2/3$, which is high, but far from perfect. However, if we look at the two possible association rules between the two items, we can gain more insight into the data. For the two occurrences of item a, the minimal occurrence of itemset X nearby is always of size 2. Therefore, $c(\{a\} \Rightarrow \{b\}) = 1$. However, for the four occurrences of item b, the minimal occurrences of X are of size 2, 2, 4 and 8, respectively, with an average of 4. Therefore, $c(\{b\} \Rightarrow \{a\}) = 0.5$. From this information, we can conclude that if we encounter item a, we can be quite certain that item b will occur nearby, while the inverse implication is less likely. □

2.3 Mining Cohesive Itemsets

In this section, we present a detailed description of our algorithms. We first show how we generate candidates in a depth-first manner, before explaining how we can prune large numbers of potential candidates by computing an upper bound of the cohesion of all itemsets that can be generated within a branch of the search tree. Next, we provide an efficient method to compute the sum of minimal windows of a particular itemset in the input sequence. Finally, we discuss algorithms for finding representative sequential patterns and dominant episodes, as well as confident association rules based on cohesive itemsets.

2.3.1 Depth First Search

The main routine of our FCI_{seq} algorithm is given in Algorithm 2.1. We begin by scanning the input sequence, identifying the frequent items, and storing their occurrence lists in an inverted index for later use. We then sort the set of frequent items on support in ascending order (line 2). The main reason for sorting items is that we need an absolute ordering to efficiently compute the upper bound. We sort on ascending support because patterns composed of less frequent items are faster to compute. Next, we initialise the set of frequent cohesive itemsets *FC* as an empty set (line 3), and start the depth-first-search process (line 4). Once the search is finished, we output the set of frequent cohesive itemsets *FC*, representative sequential patterns FC_{seq},

Algorithm 2.1: FCI$_{seq}$($\mathcal{S}$, θ, *max_size*, *min_coh*, *min_or*, *min_por*, *min_conf*) Mine cohesive itemsets, representative sequential patterns, dominant episodes and association rules in a single sequence

Input: An event sequence $\mathcal{S}$, θ, *max_size* and *min_coh* thresholds for cohesive itemset mining and optional *min_or*, *min_por*, *min_conf* thresholds for sequential patterns, dominant episodes and association rules mining

Result: Sets of frequent cohesive itemsets *FC*, (optional) set of representative sequential patterns FC_{seq} and dominant episodes FC_{epi}, (optional) set of confident association rules FC_{rules}

1 FI = all items i where $N(\{i\}) \geq \theta$;
2 sort *FI* on support in ascending order;
3 $FC = FC_{seq} = FC_{epi} = FC_{rules} = \emptyset$;
4 DFS($\mathcal{S}, \emptyset, FI, max_size, min_coh, min_or, min_por, min_conf$);
5 **return** $\langle FC, FC_{seq}, FC_{epi}, FC_{rules}\rangle$

dominant episodes FC_{epi} and association rules FC_{rules} (line 5), that are computed depending on the status of the parameters *min_or*, *min_por* and *min_conf*.

The recursive DFS procedure is shown in Algorithm 2.2. In each call, X contains the candidate itemset, while Y contains items that are yet to be enumerated. In line 1, we evaluate

Algorithm 2.2: DFS($\mathcal{S}$, X, Y, *max_size*, *min_coh*, *min_or*, *min_por*, *min_conf*) Pruned depth-first search to find cohesive itemsets and post-processing for other types of patterns and rules

Input: An event sequence $\mathcal{S}$, candidate itemset X, set of items Y, other parameter as in Algorithm 2.1

Result: Update patterns in *FC*, FC_{seq}, FC_{epi} and rules in FC_{rules}

```
1  if C_max(X, Y) < min_coh then
2  |   return;
3  else if Y ≠ ∅ then
4  |   a = FIRST(Y);
5  |   if |X ∪ {a}| ≤ max_size then
6  |   |   DFS(S, X ∪ {a}, Y \ {a}, max_size, min_coh, min_or, min_por,
   |   |     min_conf);
7  |   DFS(S, X, Y \ {a}, max_size, min_coh, min_or, min_por, min_conf);
8  else
9  |   if |X| > 1 then
10 |   |   FC = FC ∪ {X};
11 |   |   if min_or ≠ 0 or min_por ≠ 0 then
12 |   |   |   sps ← FIND_SEQUENTIAL_PATTERNS(S, X);
13 |   |   |   if min_or ≠ 0 then
   |   |   |     FC_seq ← FC_seq ∪ {sp | sp ∈ sps ∧ occ_ratio_se(sp) ≥ min_or} ;
14 |   |   |   if min_por ≠ 0 then
   |   |   |     G ← FIND_DOMINANT_EPISODE(S, X, sps, min_por);
15 |   |   |     FC_epi ← FC_epi ∪ G ;
16 |   |   if min_conf ≠ 0 then
   |   |     FC_rules ← FC_rules ∪ FIND_RULES(S, X, min_conf) ;
```

the pruning function $C_{max}(X,Y)$ to decide whether to search deeper in the tree or not. This function will be described in detail in Section 2.3.2. If the branch is not pruned, there are two possibilities. If we have reached a non-leaf node (line 3) and there are more items to be enumerated, we pick the first such item a (line 4) and make two recursive calls to the DFS function — the first with a added to X (this is only executed if $X \cup \{a\}$ satisfies the *max_size* constraint), and the second with a discarded. Alternatively, if a leaf node is encountered (line 8), we add the discovered cohesive itemset to the output (provided its size is greater than 1). We then call FIND_SEQUENTIAL_PATTERNS (discussed in Section 2.4) to find sequential patterns based on X, if either *min_or* or *min_por* is not 0. We report the representative sequential patterns if *min_or* is not 0 and call FIND_DOMINANT_EPISODE (discussed in Section 2.5) to find the dominant episode based on X if *min_por* is not 0. Finally, we also find and report association rules based on X by calling FIND_RULES (discussed in Section 2.6) if *min_conf* is not 0.

2.3.2 Pruning

At any node in the search tree, X denotes all items currently making up the candidate itemset, while Y denotes all items that are yet to be enumerated. Starting from such a node, we can still generate any itemset Z, such that $X \subseteq Z \subseteq X \cup Y$ and $|Z| \leq max_size$. In order to be able to prune the entire branch of the search tree, we must therefore be certain that for every such Z, the cohesion of Z cannot satisfy the minimum cohesion threshold. In this section, we first define an upper bound for the cohesion of all itemsets that can be generated in a particular branch of the search tree, before providing a detailed proof of its soundness.

Definition 2.14 (Upper bound cohesion)**.** *Given itemsets X and Y, with $|X| > 0$ and $X \cap Y = \emptyset$,*

$$C_{max}(X,Y) = \frac{|X \cup Y_{max}||N(X \cup Y_{max})|}{\sum_{t \in N(X)} W_t(X) + |X \cup Y_{max}||N(Y_{max})|},$$

where

$$Y_{max} = \{Y_i | \max_{\substack{Y_i \subseteq Y, \\ |Y_i| \leq max_size - |X|}} |N(Y_i)|\}. \tag{2.1}$$

For $|X| = 0$, we define $C_{max}(X,Y) = 1$.

Note that if $Y = \emptyset$, $C_{max}(X,Y) = C(X)$, which is why we do not need to evaluate $C(X)$ before outputting X in line 4 of Algorithm 2.2. Before proving that the above upper bound holds, we will first explain the intuition behind it. When we find ourselves at node $\langle X, Y \rangle$ of the search tree, we will first evaluate the cohesion of itemset X. If X is cohesive, we need to search deeper in the tree, as supersets of X could also be cohesive. However, if X is not cohesive, we need to evaluate how much the cohesion can still grow if we go deeper into this branch of the search tree. Logically, starting from $C(X) = \frac{|X|}{\overline{W}(X)} = \frac{|X||N(X)|}{\sum_{t \in N(X)} W_t(X)}$, the value of this fraction will grow maximally if the numerator is maximised, and the denominator minimised. Clearly, as we add items to X, the numerator will grow, and it will grow maximally if we add as many items to X as possible. However, as we add items to X, the denominator must grow, too, so the question is how it can grow minimally. In the worst case, each new window added to the sum in the denominator will be minimal (i.e., its length will be equal to the size of the new itemset), and the more such windows we add to the sum, the higher the overall cohesion will grow. Note that *the worst case* from the point of view of our pruning capability actually refers to the highest-scoring candidate that could yet be produced, and therefore corresponds to *the best case* in terms of pattern discovery.

Example 2.5. For example, given sequence $\mathcal{S} = acb$ and a cohesion threshold of 0.8, assume we find ourselves in node $\langle\{a,b\},\{c\}\rangle$ of the search tree. We will then first find the smallest windows containing $\{a,b\}$ for each occurrence of a and b, i.e., $W_1(\{a,b\}) = W_3(\{a,b\}) = 3$. Note that we do this by going through the lists of timestamps of a and b, and not by revisiting the entire input sequence. In other words, to compute the upper bound of the cohesion of all itemsets that can still be generated from this node, we can only use the information about the occurrences of a and b. It turns out that $C(\{a,b\}) = \frac{2\times 2}{3+3} = \frac{2}{3}$, which is not cohesive enough. However, if we add c to itemset $\{a,b\}$, we know that the size of the new itemset will be 3, we know the number of occurrences of items from the new itemset will be 3, and the numerator will therefore be equal to 9. For the denominator, we have no such certainties, but we know that, in the worst case, the windows for the occurrences of a and b will not grow (i.e., each smallest window of $\{a,b\}$ will already contain an occurrence of c), and the windows for all occurrences of c will be minimal (i.e., of size 3). Indeed, when we evaluate the above upper bound, we obtain $C_{max}(\{a,b\},\{c\}) = \frac{3\times(2+1)}{6+3\times 1} = \frac{9}{9} = 1$. We see that even though the cohesion of $\{a,b\}$ is $\frac{2}{3}$, the cohesion of $\{a,b,c\}$ could, in the worst case, be as high as 1. And in our sequence acb, that is indeed the case. □

The above example also demonstrates the tightness of our upper bound, as the computed value can, in fact, turn out to be equal to the actual cohesion of a superset yet to be generated. We now present a formal proof of the soundness of the proposed upper bound. In order to do this, we will need the following lemma.

Lemma 2.1 (A ratio inequality)**.** *For any six positive numbers a,b,c,d,e,f, with $a \le b$, $c \le d$ and $e \le f$, it holds that*

1. *if* $\frac{a+c+e}{b+e} < 1$ *then* $\frac{a+c+e}{b+e} \le \frac{a+d+f}{b+f}$,
2. *if* $\frac{a+c+e}{b+e} \ge 1$ *then* $\frac{a+d+f}{b+f} \ge 1$.

Proof. We begin by proving the first claim. To start with, note that if $\frac{a+c+e}{b+e} < 1$, then $\frac{a}{b} < 1$. It follows that $\frac{a+e}{b+e} < 1$, and for any positive number f, with $e \le f$, it holds that $\frac{a+e}{b+e} \le \frac{a+f}{b+f}$. Subsequently, it holds that $\frac{a+c+e}{b+e} \le \frac{a+c+f}{b+f}$. Finally, for any positive number d, with $c \le d$, it holds that $\frac{a+c+f}{b+f} \le \frac{a+d+f}{b+f}$, and therefore $\frac{a+c+e}{b+e} \le \frac{a+d+f}{b+f}$. For the second claim, it directly follows that if $\frac{a+c+e}{b+e} \ge 1$, then $\frac{a+c}{b} \ge 1$, $\frac{a+d}{b} \ge 1$, and $\frac{a+d+f}{b+f} \ge 1$. □

Theorem 2.2 (Upper bound on cohesion)**.** *Given itemsets X and Y, with $X \cap Y = \emptyset$, for any itemset Z, with $X \subseteq Z \subseteq X \cup Y$ and $|Z| \le max_size$, it holds that $C(Z) \le C_{max}(X,Y)$.*

Proof. We know that $C(Z) \le 1$, so the theorem holds if $|X| = 0$. Assume now that $|X| > 0$. First, recall that $C(Z) = \frac{|Z|}{\overline{W}(Z)} = \frac{|Z||N(Z)|}{\sum_{t\in N(Z)} W_t(Z)}$. We can rewrite this expression as

$$C(Z) = \frac{(|X| + |Z \setminus X|)(|N(X)| + |N(Z \setminus X)|)}{\sum_{t\in N(X)} W_t(Z) + \sum_{t\in N(Z\setminus X)} W_t(Z)}.$$

Further note that for a given timestamp in $N(X)$, the minimal window containing Z must be at least as large as the minimal window containing only X, and for a given timestamp in $N(Z \setminus X)$, the minimal window containing Z must be at least as large as the size of Z. It follows that

$$\sum_{t\in N(X)} W_t(Z) \ge \sum_{t\in N(X)} W_t(X),$$
$$\sum_{t\in N(Z\setminus X)} W_t(Z) \ge |Z||N(Z \setminus X)|,$$

and, as a result,

$$C(Z) \leq \frac{|X||N(X)| + |Z \setminus X||N(X)| + |Z||N(Z \setminus X)|}{\sum_{t \in N(X)} W_t(X) + |Z||N(Z \setminus X)|}.$$

Finally, we note that, per definition,

$$|Z \setminus X| \leq \min(max_size, |X \cup Y|) - |X|,$$

and since Z is generated by adding items from Y to X, until either the size of Z reaches *max_size* or there are no more items left in Y,

$$|N(Z \setminus X)| \leq |N(Y_{max})|.$$

At this point, we will use Lemma 2.1 to take the proof further. Note that, per definition, $C(X) = \frac{|X||N(X)|}{\sum_{t \in N(X)} W_t(X)} \leq 1$. We now denote

$$\begin{aligned}
a =& |X||N(X)|, \\
b =& \sum_{t \in N(X)} W_t(X), \\
c =& |Z \setminus X||N(X)|, \\
d =& (\min(max_size, |X \cup Y|) - |X|)|N(X)| = |Y_{max}||N(X)|, \\
e =& |Z||N(Z \setminus X)|, \\
f =& |X \cup Y_{max}||N(Y_{max})|.
\end{aligned}$$

Since a, b, c, d, e and f satisfy the conditions of Lemma 2.1, we know that it holds that

$$\text{if } \frac{|X||N(X)| + |Z \setminus X||N(X)| + |Z||N(Z \setminus X)|}{\sum_{t \in N(X)} W_t(X) + |Z||N(Z \setminus X)|} < 1, \text{ then}$$

$$\frac{|X||N(X)| + |Z \setminus X||N(X)| + |Z||N(Z \setminus X)|}{\sum_{t \in N(X)} W_t(X) + |Z||N(Z \setminus X)|} \leq$$

$$\frac{|X||N(X)| + |Y_{max}||N(X)| + |X \cup Y_{max}||N(Y_{max})|}{\sum_{t \in N(X)} W_t(X) + |X \cup Y_{max}||N(Y_{max})|} \qquad (1)$$

and

$$\text{if } \frac{|X||N(X)| + |Z \setminus X||N(X)| + |Z||N(Z \setminus X)|}{\sum_{t \in N(X)} W_t(X) + |Z||N(Z \setminus X)|} \geq 1, \text{ then}$$

$$\frac{|X||N(X)| + |Y_{max}||N(X)| + |X \cup Y_{max}||N(Y_{max})|}{\sum_{t \in N(X)} W_t(X) + |X \cup Y_{max}||N(Y_{max})|} \geq 1. \qquad (2)$$

Finally, note that

$$\frac{|X||N(X)| + |Y_{max}||N(X)| + |X \cup Y_{max}||N(Y_{max})|}{\sum_{t \in N(X)} W_t(X) + |X \cup Y_{max}||N(Y_{max})|} =$$

$$\frac{|X \cup Y_{max}|(|N(X)| + |N(Y_{max})|)}{\sum_{t \in N(X)} W_t(X) + |X \cup Y_{max}||N(Y_{max})|} = C_{max}(X, Y). \qquad (3)$$

From Equations 1 and 3 it follows that

$$\text{if } \frac{|X||N(X)| + |Z \setminus X||N(X)| + |Z||N(Z \setminus X)|}{\sum_{t \in N(X)} W_t(X) + |Z||N(Z \setminus X)|} < 1, \text{ then } C(Z) \leq C_{max}(X, Y).$$

From Equations 2 and 3 it follows that

$$\text{if } \frac{|X||N(X)| + |Z \setminus X||N(X)| + |Z||N(Z \setminus X)|}{\sum_{t \in N(X)} W_t(X) + |Z||N(Z \setminus X)|} \geq 1, \text{ then } C_{max}(X,Y) \geq 1,$$

and since, per definition, $C(Z) \leq 1$, it follows that $C(Z) \leq C_{max}(X,Y)$.
This completes the proof. □

Since an important feature of computing an upper bound for the cohesion of all itemsets in a given branch of the search tree is to establish how much cohesion could grow *in the worst case*, we need to figure out which items from Y should be added to X to reach this worst case. As has been discussed above, the worst case is actually materialised by adding as many as possible items from Y, and by first adding those that have the most occurrences. However, if the *max_size* parameter is used, it is not always possible to add all items in Y to X. In this case, we can only add *max_size* $-|X|$ items to X, which is why we defined Y_{max} as we did in Equation 2.1. Clearly, if $|X \cup Y| \leq$ *max_size*, then $Y_{max} = Y$. If not, at first glance it may seem computationally very expensive to determine $|N(Y_i)|$ for every possible Y_i. However, we solve this problem by sorting the items in Y on support in ascending order. In other words, if $Y = \{y_1, \ldots, y_n\}$, with $support(y_i) \leq support(y_{i+1})$ for $i \in \{1, \ldots, n-1\}$, then we can compute $|N(Y_{max})|$ as

$$|N(Y_{max})| = \sum_{i \in \{n, n-1, \ldots, n-(max_size-|X|)\}} |N(\{y_i\})|.$$

As a result, the only major step in computing $C_{max}(X,Y)$ is that of computing $\sum_{t \in N(X)} W_t(X)$, as the rest can be computed in constant time. The procedure for computing $\sum_{t \in N(X)} W_t(X)$ is explained in detail in Section 2.3.3.

Example 2.6. We now give an example illustrating our pruning technique. Given an input sequence $\mathcal{S} = acdebbfgha$ and thresholds *min_coh* = 0.8 and *max_size* = 3, assume we are visiting the $\langle X, Y \rangle$ node of the search tree, with $X = \{a, b\}$ and $Y = \{c, d, e, f, g, h\}$. At this point, we will compute the sizes of the minimal occurrences of itemset $\{a, b\}$ for timestamps 1, 5, 6 and 10, and find that they all equal 5. As a result, the cohesion of $\{a, b\}$ will be equal to 0.4. However, we cannot be certain that we can prune this branch of the tree unless we know that none of the itemsets that can be generated within it cannot be cohesive. Therefore, we need to evaluate our upper bound for the cohesion of all such itemsets, $C_{max}(X,Y)$. We first compute

$$Y_{max} = \{Y_i | \max_{\substack{Y_i \subseteq Y, \\ |Y_i| \leq max_size - |X|}} |N(Y_i)|\} = \{c\}.$$

As discussed above, by sorting the items in Y on frequency, we know that Y_{max} can be obtained by picking items in order from Y until we have either reached the *max_size* constraint (as in this case) or run out of items. We then compute

$$C_{max}(X,Y) = \frac{|X \cup Y_{max}||N(X \cup Y_{max})|}{\sum_{t \in N(X)} W_t(X) + |X \cup Y_{max}||N(Y_{max})|}$$

$$= \frac{|\{a,b,c\}||N(\{a,b,c\})|}{\sum_{t \in N(X)} W_t(X) + |\{a,b,c\}||N(\{c\})|} = \frac{3 \times 5}{20 + 3 \times 1} = \frac{15}{23} \approx 0.65.$$

In other words, no itemset that can be generated within this branch of the search tree can have a cohesion higher than 0.65, and since the cohesion threshold is set to 0.8, we can safely prune the entire branch. □

2.3.3 Computing the Sum of Minimal Windows

For computing the upper bound $C_{max}(X,Y)$ we have to compute the sum of minimal windows, i.e. $\sum_{t\in N(X)} W_t(X)$. We need to compute the minimal window for each occurrence. However, the goal of this algorithm is to determine whether we can prune a branch of the search tree or not. We know that a branch can be pruned if the computed sum of windows is large enough. Therefore, we optimise the algorithm to stop computing minimal windows once the running sum is already large enough since in that case we can prune without computing the exact sum.

Theorem 2.3 (Upper bound sum of minimal windows)**.** *If the sum of minimal windows is larger than*

$$W_{max}(X,Y) = \frac{|X \cup Y_{max}||N(X)| + |N(Y_{max})|(1 - min_coh)}{min_coh}.$$

we abandon computing of the sum of minimal windows since $C_{max}(X,Y) < min_coh$*.*

Proof. This follows directly from the Definition 2.14 of $C_{max}(X,Y)$. We substitute $C_{max}(X,Y)$ with min_coh and substitute $\sum_{t\in N(X)} W_t(X)$ with $W_{max}(X,Y)$ and solve to $W_{max}(X,Y)$. □

The algorithm for computing the sum of minimal windows is shown in Algorithm 2.3. We use an *inverted index* to store the positions for each item. This makes traversal much faster since we only consider occurrences of items in the candidate pattern, that is, $|N(X)| \ll |\mathcal{S}|$. Remark that we do not consider streaming applications where the sequence is too large to fit in memory. For a given itemset X, the algorithm retrieves a list of all timestamps at which items of X occur from the inverted index in the *pos* variable (line 2). The *nextpos* variable keeps a list of next timestamps for each item, while *lastpos* keeps a list of the last seen occurrences for each item. We loop over each occurrence and compute the running sum of minimal windows. We *abandon* this routine if the current sum is larger than the upper bound (line 8). The first time we visit each item occurrence, we are not sure that we will encounter a smaller window in the continuation of the loop. Therefore, we define *active* and *final* windows. A final window is guaranteed to have a minimal size, while an active window could be replaced by a smaller window size. When a new item comes in, we update the working variables and compute the first and last position of the current window (line 17). If the smallest timestamp of the current window has changed, we go through the list of active windows and check whether a new shortest length has been found. If so, we update it (line 21). We then remove all windows for which we are certain that they cannot be improved from the list of active windows, and update the overall sum (line 23-24). Finally, we add the new window for the current timestamp to the list of active windows (line 25). Note that the sum of minimal windows is independent of Y, the items yet to be enumerated. Therefore, if the branch is not pruned, the recursive DFS procedure shown in Algorithm 2.2 will be called twice, but X will remain unchanged in the second of these calls, so we will not need to recompute the sum of windows, allowing us to immediately evaluate the upper bound in the new node of the search tree.

Example 2.7. We illustrate how the algorithm works on the following example. Assume we are given the input sequence $\mathcal{S} = aabccccacb$, and we are evaluating itemset $\{a,b,c\}$. Table 2.1 shows the values of the main variables as the algorithm progresses. As each item comes in, we update the values of *nextpos* and *lastpos*. In each iteration, we compute the current best minimal window for the given timestamp as max(*lastpos*)−min(*lastpos*)+1. We also update the values of any previous windows that might have changed for the better (this can only happen if min(*lastpos*) has changed), using either the current window above if it contains the timestamp of the window's event, or the window stretching from the relevant timestamp to max(*lastpos*).

Algorithm 2.3: SUM_MIN_WINS($\mathcal{S}, X, Y$) Compute sum of minimal windows of each occurrence of X in $\mathcal{S}$

```
   Input: An event sequence S, candidate itemset X, set of items Y
   Result: Sum of minimal windows or ∞ if sum is higher than W_max(X, Y)
   /* Maintain position for each item  */
 1 smw ← 0; index ← 0;
 2 pos ← positions for every item in X;
 3 nextpos ← {pos[i_1][0], pos[i_2][0], pos[i_3][0], ...};
 4 lastpos ← {−∞, −∞, −∞, ...};
 5 prev_min ← −∞;
 6 active_windows ← ∅;
 7 for index in N(X) do
       /* Abandon if running sum too high to be cohesive */
 8     if smw + (|N(X)| + |active_windows| − index) × |X| > W_max(X, Y) then
 9         return ∞
10     current_pos ← ∞; current_item ← ∅;
11     for i in X do              // Update last and next position of new item
12         if current_pos > nextpos[i] then
13             current_pos ← nextpos[i]; current_item ← i
14     lastpos[current_item] ← current_pos;             // Compute current window
15     nextpos[current_item] ← NEXT(pos[current_item], current_pos);
16     minpos ← min(lastpos);
17     maxpos ← max(lastpos);
       /* If new minimum in current window */
18     if minpos ≠ −∞ and minpos > prev_min then
           /* For each non-final window */
19         for window in active_windows do
20             newwidth ← maxpos − min(minpos, window.pos) + 1;
21             window.width ← min(window.width, newwidth);
               /* Check if minimal window is final */
22             if window.pos < minpos or window.width = |X|
                 or window.width < (maxpos − window.pos + 1) then
23                 active_windows ← active_windows \ {window};
24                 smw ← smw + window.width;
25     active_windows ← active_windows ∪
        {WINDOW(current_pos, maxpos − minpos + 1)} ;
26     prev_min ← minpos;
27 smw ← smw + ∑_{window ∈ active_windows} window.width;
28 return smw
```

Finally, before proceeding with the next iteration, we remove all windows for which we are certain that they cannot get any smaller from the list of active windows.

In the table, windows that are not active are marked with '-', while definitively determined windows are shown in bold. We can see that, for example, at timestamp 4, we have determined the value of the first four windows. Window w_1 cannot be improved on, since timestamp 1 has already dropped out of *lastpos*, while the other three windows cannot be improved since 3 is the absolute minimum for a window containing three items. At timestamp 8, we know that

Table 2.1: Example of computing minimal windows of itemset $\{a, b, c\}$ in sequence *aabccccacb*

t	i	*nextpos*	*lastpos*	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}
0	-	(1, 3, 4)	(∞, ∞, ∞)	-	-	-	-	-	-	-	-	-	-
1	a	(2, 3, 4)	(1, ∞, ∞)	∞	-	-	-	-	-	-	-	-	-
2	a	(8, 3, 4)	(2, ∞, ∞)	∞	∞	-	-	-	-	-	-	-	-
3	b	(8, 10, 4)	(2, 3, ∞)	∞	∞	∞	-	-	-	-	-	-	-
4	c	(8, 10, 5)	(2, 3, 4)	**4**	**3**	**3**	**3**	-	-	-	-	-	-
5	c	(8, 10, 6)	(2, 3, 5)	-	-	-	-	4	-	-	-	-	-
6	c	(8, 10, 7)	(2, 3, 6)	-	-	-	-	4	5	-	-	-	-
7	c	(8, 10, 9)	(2, 3, 7)	-	-	-	-	4	5	6	-	-	-
8	a	(∞, 10, 9)	(8, 3, 7)	-	-	-	-	**4**	5	6	6	-	-
9	c	(∞, 10, ∞)	(8, 3, 9)	-	-	-	-	-	5	6	6	7	-
10	b	(∞, ∞, ∞)	(8, 10, 9)	-	-	-	-	-	**5**	**4**	**3**	**3**	**3**

the length of w_5 must be equal to 4, since any new window to come must stretch at least from timestamp 5 to a timestamp in the future, i.e., at least 9. Finally, once we have reached the end of the sequence, we mark all current values of still active windows as determined. □

2.4 Mining Representative Sequential Patterns

In this section, we describe our algorithm for discovering representative sequential patterns based on frequent cohesive itemsets. For example, when mining patterns consisting of words used in the *On the Origin of Species by Means of Natural Selection* by *Charles Darwin* (see Section 2.8 for more details about the dataset) we discover that itemset {*tierra, del, fuego*} has a cohesion of 1. However, we also find that sequential pattern *tierra del fuego* has an occurrence ratio of 1 within the occurrences of its underlying itemset {*tierra, del, fuego*}. In other words, in every minimal occurrence of itemset {*tierra, del, fuego*}, the word *tierra* occurs before the word *del*, followed by *fuego*. In this section we describe how, starting from a frequent cohesive itemset and its minimal occurrences, we discover representative sequential patterns.

2.4.1 Computing Minimal Windows for Sequential Patterns

Computing the number of occurrences of sequential patterns, as defined in Section 2.2.2, brings with it additional complexity. So far, we were only interested in the size of the minimal window at each occurrence. However, at first glance, given a cohesive itemset $X = \{i_1, \ldots, i_n\}$, we must now count each occurrence of up to $n!$ sequential permutations in the worst case. To compute the number of occurrences correctly, we must also deal with the fact that *more than one* sequential pattern can occur within one minimal occurrence of the itemset. For example, given sequence $a_1\ b_2\ a_3$ we need to take into account that both sequential patterns (a_1, b_2) and (b_2, a_3) occur at timestamp 2 since the occurrences of both are equally long as the minimal occurrence of itemset $\{a, b\}$. For brevity of this chapter, Algorithm SUM_MIN_WINS$_{\text{seq}}$ for computing the *minimal windows for sequential patterns* is included in Appendix A.1.

2.4.2 Algorithm

We now describe the algorithm FIND_SEQUENTIAL_PATTERNS, shown in Algorithm 2.4, which returns the set of all sequential patterns that occur within the minimal windows of underlying itemset X. In the main algorithm (see Algorithm 2.2) we only report sequential patterns that occur often enough, that is, we remove sequential patterns where the occurrence ratio is lower than *min_or*. Remark that we have to deal with *multiple permutations* within the same window caused by *duplicate* items. For example, given itemset $\{a, b, c, d\}$ and a minimal window $a\underline{b}c\underline{b}d$, both sequential patterns $abcd$ and $acbd$ occur within the same minimal window since b occurs more than once. In general, our algorithm should discover any permutation of $|X|$ elements. A naïve approach would enumerate all $|X|!$ candidate sequential patterns and check for each minimal window of X if the candidate occurs. Our algorithm, however, only generates candidate sequential patterns that occur in at least one window, making the method feasible even if $|X|$ is large.

FIND_SEQUENTIAL_PATTERNS first calls $\text{SUM_WIN_WINS}_{\text{seq}}$. Remark that here we do not require the total sum of minimal windows, however, this is needed when evaluating $C_{max}(X, Y)$

Algorithm 2.4: FIND_SEQUENTIAL_PATTERNS($\mathcal{S}, X$) Generate sequential patterns based on cohesive itemset X

Input: An event sequence $\mathcal{S}$, frequent cohesive itemset X
Result: Multiset of sequential patterns *sps* based on minimal windows of X. The count of each sequential pattern corresponds to $occ_{se}(sp)$

```
1  ⟨smw, min_windows_seq⟩ ← SUM_MIN_WINS_seq(S, X, ∅);
2  sps ← MULTISET();
   /* Main loop over each occurrence.  */
3  for win in min_windows_seq do
4  |  win_occurrences ← ∅;
   |  /* Inner loop over each minimal window instance */
5  |  for win_ins in win do
6  |  |  pos ← {⟨i, t⟩| i ∈ X ∧ win_ins.min ≤ t ≤ win_ins.max};
   |  |  /* Remove positions not relevant for enumerating sequential
   |  |     patterns */
7  |  |  for n = 1; n < |pos|; n = n + 1 do
8  |  |  |  ⟨i_n, t_n⟩ ← pos[n];
9  |  |  |  ⟨i_{n+1}, t_{n+1}⟩ ← pos[n + 1];
10 |  |  |  if i_n = i_{n+1} or i_n = i_0 or i_n = i_{|pos|} then
11 |  |  |  |  pos ← pos \ pos[n];
   |  |  /* Enumerate sequential patterns using the cartesian product of
   |  |     positions */
12 |  |  for i in X do
13 |  |  |  pos_i ← {⟨i, t⟩ | ⟨i, t⟩ ∈ pos};
14 |  |  cart_prod ← pos_1 × pos_2 × ... × pos_{|X|};
   |  |  /* Update occurrences in window */
15 |  |  for X_pos in cart_prod do
16 |  |  |  win_occurrences ← win_occurrences ∪ TO_SEQUENCE(X_pos);
   |  /* Update total occurrences in sequence.  */
17 |  sps ← sps ⊎ win_occurrences;
18 return sps;
```

during depth-first search. Next, we define an empty multiset (line 2) which is returned (line 18) and updated with the discovered sequential patterns in each window (line 17). The idea here is that we count each permutation that occurs at each occurrence, and return this multiset, for example returning $\{abcd : 5, acbd : 4\}$. The outer loop (lines 3 to 17) loops over all occurrences of the itemset. We create an empty set (line 4) to count any distinct *sequential pattern* found within the current occurrence. Note that adding each sequential pattern directly to the multiset would result in potentially doubly counting a sequential pattern occurrence. Next, in the inner loop (lines 5 to 16) we loop over possibly multiple minimal window instances at each occurrence. In the inner loop we generate sequential patterns occurring within each minimal window instance. We first fetch the positions for each item of X within the window (line 6) from the input sequence and retrieve a list of $\langle i, t\rangle$ positions. Next, we *filter* out positions that are not relevant for generating candidate sequential patterns: For the boundary items at the minimal and maximal positions, we know (by definition of a minimal window) that no other position of those items will result in a smaller window. Thus, each sequential pattern must start with the item found at the left boundary and end with the item on the right boundary. Therefore, we can safely ignore any other positions of the boundary items within the minimal window (line 11). We also remove any position where two or more occurrences of the same item directly follow each other (line 11). Next, we enumerate all possible permutations using a *cartesian product* (line 14) of the filtered position lists of each item. In the simplest case, all elements will occur just once, and the cartesian product will result in exactly one sequential pattern. If we do have multiple positions for some items, the cartesian product will combine them with positions of other items and generate multiple sequential patterns. We then add these sequential patterns to the current set of already discovered sequential patterns within the window.

Example 2.8. In order to illustrate FIND_SEQUENTIAL_PATTERNS we provide an example. Assume an itemset $X = \{a, b, c, d\}$ and input sequence

$$\begin{matrix} a & z & b & c & b & z & b & c & d. \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{matrix}$$

Note that, in this case, the entire input sequence is also a minimal window of X. The position list is then $\{\langle a,1\rangle\, \langle b,3\rangle\, \langle c,4\rangle\, \langle b,5\rangle, \langle b,7\rangle, \langle c,8\rangle\, \langle d,9\rangle\}$. We filter out the second occurrence of b since it it is directly followed by the another occurrence of b. We then take the cartesian product of the position lists of each item, thereby generating all candidate sequential patterns within this window:

$$\begin{matrix} a & \times b & \times c & \times d \\ 1 & 3 & 4 & 9 \\ & 7 & 8 & \\ & & & \\ & & & \end{matrix} \quad = \quad \begin{matrix} a \times b \times c \times d \\ 1 \quad 3 \quad 4 \quad 9 \\ 1 \quad 7 \quad 4 \quad 9 \\ 1 \quad 3 \quad 8 \quad 9 \\ 1 \quad 7 \quad 8 \quad 9 \end{matrix} \quad = \quad \begin{matrix} \text{to_sequence} \\ \text{a b c d} \\ \text{a c b d} \\ \text{a b c d} \\ \text{a b c d} \end{matrix}$$

We then convert the result of the cartesian product trivially into distinct sequential patterns, that is $abcd$ and $acbd$, and increment the support for both sequential patterns. □

Example 2.9. For a second example, assume $X = \{b, c\}$ and the input sequence is the same as above. SUM_MIN_WINS$_{\text{seq}}$ would then generate a minimal window at $t \in \{3, 4, 5, 7, 8\}$ of length 2 and two minimal window *instances* at $t = 4$ (bc and cb). We find two sequential patterns, $sp_1 = bc$ occurring at $t \in \{3, 4, 7, 8\}$ and $sp_2 = cb$ occurring at $t = \{4, 5\}$. Given a minimal occurrence ratio threshold $min_or = 0.8$, we conclude that sp_1 is representative, and sp_2 is not. □

2.5 Mining Dominant Episodes

In this section, we describe our algorithm for finding dominant episodes within the minimal occurrences of cohesive itemsets. In the previous section, given an itemset X, we enumerated all representative sequential patterns. Here we find the single most dominant episode (or partial order) of X. We take a more direct approach, and compute the *intersection* of the *top k* sequential patterns (or total orders) of X ranked on occurrence ratio. Likewise, we determine the highest possible threshold, t_{max}, on sequential pattern occurrence ratio. The value of k is specific to a particular itemset X, and is determined by iteratively using more sequential patterns until the resulting episode satisfies the minimal occurrence ratio threshold *min_por*.

FIND_DOMINANT_EPISODE, our algorithm for finding the dominant episode of a frequent cohesive itemset, is shown in Algorithm 2.5. Three parameters must be provided: a cohesive itemset X, a set of sequential patterns *sps* that is computed using FIND_SEQUENTIAL_PATTERNS, and the minimal occurrence ratio threshold *min_por*. We start by computing the absolute minimal support required (line 1). Then we sort the sequential patterns on descending support. Next, we initialise the candidate episode with all nodes of itemset X, and no edges. We then start our main loop (lines 5 to 18) by generating candidate episodes based on the intersection of the top k total orders (or sequential patterns). For each sequential pattern, we compute

Algorithm 2.5: FIND_DOMINANT_EPISODE($\mathcal{S}, X, sps, min_por$) Generate a dominant episode based on cohesive itemset X

Input: An event sequence $\mathcal{S}$, frequent cohesive itemset X, multiset of sequential patterns *sps* (computed by Algorithm 2.4), threshold on occurrence ratio *min_por*

Result: Dominant episode G and $|occ_{po}(G)|$. Possibly $E(G) = \emptyset$.

```
1  min_sup_G ← ⌈|N(X)| × min_por⌉;
2  sort sps descending on occurrence ratio;
3  G ← G(X, ∅);
4  support_G ← 0;
   /* Generate candidate episodes incrementally by taking the
      intersection of the first k total orders */
5  for k = 1; k ≤ |sps|; k = k + 1 do
6      sp ← sps[k];
7      G_sp ← G(X, ∅);
       /* Compute transitive closure current total order */
8      for i ← 1 to |sp| − 1 do
9          for j ← i + 1 to |sp| do
10             E(G_sp) ← E(G_sp) ∪ ⟨sp[i], sp[j]⟩;
11     if E(G) = ∅ then
12         E(G) ← E(G_sp);
13     else
14         E(G) ← E(G) ∩ E(G_sp);
15     if E(G) = ∅ then
16         break;
17     support_G ← COMPUTE_SUPPORT_EPISODE(S, G);
18     if support_G ≥ min_sup_G then break ;
19 return ⟨G, support_G⟩;
```

the transitive closure of the total order. For the first sequential pattern, we then initialise the candidate episode with this set of edges (line 12). For the remaining sequential patterns, we iteratively take the intersection (line 14) with the previous partial order, thereby keeping only the edges between items that hold for top-2, top-3 and finally top-k sequential patterns. As we remove edges, based on the intersection, the occurrence ratio of the resulting episode grows (or remains unchanged). We return the candidate episode if its occurrence ratio has reached the required minimal occurrence ratio (line 18). Note that if the set of edges becomes empty, we break the main loop (line 15) and return the itemset itself (an episode that imposes no order at all on its events). To compute the support of an episode based on minimal windows of an itemset X, we use the COMPUTE_SUPPORT_EPISODE procedure included in Appendix A.2.

Example 2.10. In order to illustrate FIND_DOMINANT_EPISODE we provide the following example. Suppose we have a cohesive itemset $X = \{a, b, c\}$ and input sequence $\mathcal{S} = abc \dots abc \dots acb$. FIND_SEQUENTIAL_ PATTERNS(X) finds two sequential patterns: $sp_1 = abc$ occurs 6 times and $sp_2 = acb$ occurs 3 times. Next, we execute FIND_DOMINANT_EPISODE($X, sps = \{abc : 6, acb : 3\}, min_por = 0.8$). We first compute the transitive closure $G_{sp_1} = G(X, \{a \rightarrow b, b \rightarrow c, a \rightarrow c\})$ which is our first candidate G_1. The support of G_1 is 6, which is strictly less than the required absolute minimal support of $\lceil N(X) \times min_por \rceil = \lceil 9 \times 0.8 \rceil = 8$ (line 1 in Algorithm 2.5). Next, we compute $G_{sp_2} = G(X, \{a \rightarrow c, c \rightarrow b, a \rightarrow b\}$ and compute the intersection to get our second candidate $G_2 = G(X, \{a \rightarrow b, b \rightarrow c, a \rightarrow c\} \cap \{a \rightarrow c, c \rightarrow b, a \rightarrow b\}) = G(X, \{a \rightarrow b, a \rightarrow c\})$. We find that $support(G_2) = 9$, and we have thus found our dominant episode w.r.t. min_por, in which a occurs before b and c, but no order is imposed between b and c. Here, $occ_ratio_{po}(G_2) = 1$ and $t_{max} = occ_ratio_{se}(sp_2) = 3/9$. □

2.6 Mining Association Rules

We conclude this section with an algorithm for *efficiently* discovering *confident association rules* based on cohesive itemsets. Before presenting the algorithm, we introduce a theorem that we use to compute the confidence of rules $Y \Rightarrow X \setminus Y$, with $Y \subset X$ and $|Y| \geq 2$, without additional dataset scans.

2.6.1 Efficiently Computing Confidence

When discovering a cohesive itemset X, we need to compute the exact minimal window $W_t(X)$ containing X for each timestamp t at which an item $x \in X$ occurs. In fact, we compute the sum of all such windows for each $x \in X$ in SUM_MIN_WINS, before adding them up into the overall sum needed to compute $C(X)$. With these sums still in memory, we can easily compute the confidence of all association rules of the form $x \Rightarrow X \setminus \{x\}$, with $x \in X$, that can be generated from itemset X. Formally, the confidence of such a rule is equal to

$$c(x \Rightarrow X \setminus \{x\}) = \frac{|X|}{\overline{W}(\{x\}, X \setminus \{x\})},$$

$$\text{where } \overline{W}(\{x\}, X \setminus \{x\}) = \frac{\sum_{t \in N(\{x\})} W_t(X)}{|N(\{x\})|},$$

and these are precisely the sums of windows we have already computed when discovering itemset X itself. We now show that, in practice, it is sufficient to limit our computations to rules of precisely this form (i.e., rules where the antecedent consists of a single item), as the

confidence of all rules $Y \Rightarrow Z$, with $|Y| \geq 2$, can be derived from the confidence values of rules of this form.

Theorem 2.4 (Computing rule confidence based on single antecedents)**.** *Given a frequent cohesive itemset X and its two disjunct subsets Y and Z (i.e., $X = Y \cup Z$ and $Y \cap Z = \emptyset$), such that $|Y| \geq 2$ and $|Z| \geq 1$, it holds that*

$$c(Y \Rightarrow Z) = \frac{|N(Y)|}{\sum_{y \in Y} \frac{|N(\{y\})|}{c(\{y\} \Rightarrow Z \cup Y \setminus \{y\})}}.$$

Proof. We begin the proof by noting that

$$\sum_{t \in N(Y)} W_t(Y \cup Z) = \sum_{y \in Y} \sum_{t \in N(\{y\})} W_t(Y \cup Z).$$

A trivial mathematical property tells us that the sum of some numbers is equal to their average multiplied by their quantity, and therefore

$$\sum_{t \in N(\{y\})} W_t(Y \cup Z) = \overline{W}(\{y\}, Z \cup Y \setminus \{y\}) \cdot |N(\{y\})|.$$

As a result, we can conclude that

$$\overline{W}(Y, Z) = \frac{\sum_{t \in N(Y)} W_t(Y \cup Z)}{|N(Y)|} = \frac{\sum_{y \in Y} \overline{W}(\{y\}, Z \cup Y \setminus \{y\}) \cdot |N(\{y\})|}{|N(Y)|},$$

which in turn implies that

$$c(Y \Rightarrow Z) = \frac{|Y \cup Z|}{\overline{W}(Y, Z)} = \frac{|Y \cup Z| \cdot |N(Y)|}{\sum_{y \in Y} \overline{W}(\{y\}, Z \cup Y \setminus \{y\}) \cdot |N(\{y\})|}.$$

Meanwhile, from the definition of confidence, we can derive that

$$c(\{y\} \Rightarrow Z \cup Y \setminus \{y\}) = \frac{|Y \cup Z|}{\overline{W}(\{y\}, Z \cup Y \setminus \{y\})},$$

and therefore it holds that

$$\overline{W}(\{y\}, Z \cup Y \setminus \{y\}) = \frac{|Y \cup Z|}{c(\{y\} \Rightarrow Z \cup Y \setminus \{y\})},$$

from which it directly follows that

$$c(Y \Rightarrow Z) = \frac{|N(Y)|}{\sum_{y \in Y} \frac{|N(\{y\})|}{c(\{y\} \Rightarrow Z \cup Y \setminus \{y\})}}.$$

□

As a result, once we have evaluated all the rules of the form $x \Rightarrow X \setminus \{x\}$, with $x \in X$, we can then evaluate all other rules $Y \Rightarrow X \setminus Y$, with $Y \subset X$ and $|Y| \geq 2$, without further dataset scans.

2.6.2 Algorithm

The algorithm for discovering association rules is shown in Algorithm 2.6. First, we compute the confidence of all rules where the left-hand-side consists of a single item, and cache this result in memory (lines 1 to 8). Then we generate the powerset of X (line 9) and for each proper subset Y of X we compute the confidence based on Theorem 2 (lines 10 to 15). We return all rules that exceed the *min_conf* threshold (line 16). Note that, unlike traditional approaches, which first find all frequent itemsets and only then start generating association rules, we generate rules in parallel with the cohesive itemsets as shown in Algorithm 2.2. Finally, remark that the pseudocode of Algorithms 2.4, 2.6 and A.2 begins by computing the minimal windows of itemset X. While this is included for the sake of formal completeness, this line is not actually executed at this point. These minimal windows are computed already when evaluating $C_{max}(X, Y)$ in line 1 of Algorithm 2.2, and are then stored and reused later in Algorithms 2.4, 2.6 and A.2.

Algorithm 2.6: FIND_RULES($\mathcal{S}, X, min_conf$) Generate confident association rules from cohesive itemset X

```
   Input: An event sequence S, frequent cohesive itemset X, threshold on confidence
          min_conf
   Result: Set of confident association rules
 1 ⟨smw, min_windows⟩ ← SUM_MIN_WINS(S, X, ∅);
   /* Compute confidence of rules i ⇒ X \ {i} */
 2 conf_items ← ∅;
 3 for i in X do
 4 |   pos_i ← {t | ⟨i, t⟩ ∈ S};
 5 |   smw_i ← 0;
 6 |   for win in min_windows do
 7 |   |   if win.pos ∈ pos_i then smw_i ← smw_i + win.width ;
 8 |   conf_items[i] ← |X| / (smw_i / |N({i})|);
   /* Generates candidate rules of form Y ⇒ X \ Y */
 9 rules ← ∅; pset ← POWERSET(X);
10 for Y in pset do
11 |   if |Y| > 0 and |Y| < |X| then
   |   |   /* Compute confidence Y ⇒ X \ Y based on conf_items */
12 |   |   smw_Y ← 0;
13 |   |   for i in Y do
14 |   |   |   smw_Y ← smw_Y + |N({i})| / conf_items[i];
15 |   |   conf_Y ← |N(Y)| / smw_Y;
16 |   |   if conf_Y ≥ min_conf then
   |   |     rules ← rules ∪ {Y → X \ Y} ;
17 return rules;
```

2.7 Setting Parameters and Top-*k* Mining

Given the exponential nature of the itemset candidate space, FCI_{seq} can take a very long time to complete, depending on the parameters. It is therefore crucial to set the parameters sensibly. This is especially the case for the cohesion threshold (*min_coh*), which determines whether

parts of the search space can be pruned or not. Here we provide some insight into how a user could come up with a good parameter setting.

Setting Parameters

First, we remark that setting θ might be done by a domain expert, for example by looking at the possible long tail of infrequent items, and deciding if itemsets consisting of these infrequent items are worth exploring. The *max_size* parameter is optional since normally no dataset will contain any cohesive patterns longer than a certain size due to data characteristics (we show this experimentally in Section 2.8.4), but if a user is only interested in shorter patterns, this parameter can be set according to personal choice (it can also be used to reduce runtimes, if necessary).

However, if no domain expert is available, we propose the following procedure for setting each parameter. The idea of the procedure is to start with parameter values that only require a short time, typically minutes, to mine patterns. New parameter settings can be explored in small steps, thereby lengthening the runtime gradually to re-run FCI_{seq} to find more patterns. We start with a relatively high value of θ (depending on the size of the dataset), a small value for *max_size*, e.g., 4, and a high value of *min_coh*, e.g., 0.9. or 1. This first run should execute very fast, with high potential for pruning, and a limited candidate space. After this initial run, the user can incrementally decrease *min_coh*, in steps of 0.1, until a sufficient number of patterns is found. After this step, θ can be decreased to smaller values to see if any new patterns involving less frequent items appear high in the ranking. We remark that, unlike frequent pattern mining, low values for θ do not seem to significantly increase runtime in our experiments (see Section 2.8.4). This is because patterns consisting of low-frequent items are often easily pruned, as the mean minimal window size of supersets consisting of other low-frequent and high-frequent items are often large, allowing us to effectively prune most candidates. Similarly, *max_size* can be increased to see if any new large patterns appear high in the ranking. The user can stop decreasing θ and increasing *max_size* if no new interesting patterns are found or if the runtimes become prohibitive. We remark that it would interesting to further study incremental or *streaming* algorithms as discussed in Chapter 5.

The two occurrence ratio thresholds, used for mining sequential patterns and episodes, should be set high since the idea is to discover *representative* total and partial orders that capture most of the itemset's occurrences. If the occurrence ratio is low, the order of items (total or partial) can hardly be considered representative. Similarly, the confidence threshold should be set high in order to produce reliable association rules.

Top-k Mining

Finally, we remark that our algorithm can quite easily be adapted to allow the mining of *top-k* most cohesive patterns without the need to set *min_coh* in advance. To do this, we could maintain a *heap* of patterns during depth-first search. We would only add patterns to the heap if fewer than k patterns are in the heap, or if the cohesion of the current candidate pattern is higher than the minimal cohesion of a pattern currently in the heap (in which case the pattern with minimal cohesion would be removed from the heap). After k candidates are added, we can use the minimal value of cohesion in the heap of current candidates as a dynamic value for *min_coh* and use it for pruning using $C_{max}(X, Y)$ as before. As more patterns are discovered, the lowest value for cohesion in the heap increases, thereby pruning more candidates. Moreover, using this dynamic value as an *increasing* threshold for the upper bound would satisfy the invariant that at each step all pruned candidates are safely pruned using

$C_{max}(X, Y) < min_heap$, and thus have a lower cohesion than the pattern with the minimal cohesion in the final top-k set of patterns.

2.8 Experiments

In this section, we compare our method with related state-of-the-art mining algorithms that take a *single* event sequence as input — WINEPI, LAXMAN[2], MARBLES_W and Compact Minimal Windows (CMW) (Tatti 2014a). As discussed in Section 2.1, these algorithms use a variety of frequency-based quality measures to evaluate the patterns, or, in the case of CMW, re-rank the output according to the difference, or leverage, between actual and expected minimal window lengths.

Since the available implementations[3] were made with the goal of discovering partially ordered episodes, we had to post-process the output in order to filter out only itemsets (Tatti 2014b). Additionally, in some cases, we had to slightly amend the implementations to generate not only closed but all frequent patterns. Therefore, making any kind of runtime comparisons would be unfair on these methods, since general episode mining requires the generation of many more candidate patterns than itemset mining.

In Section 2.8.1 we use a Hidden Markov Model-based generator to create several *synthetic* data sets. We use this generator to reproduce the benchmark study of Zimmermann (2014a,b) thereby creating synthetic sequences and embedded patterns under different assumptions. We then compare the performance of the state-of-the-art methods with FCI_{seq}, by reporting the rank of the discovered embedded patterns. In Section 2.8.2 we use different *real-world* text datasets and compare the top-k episodes of different state-of-the-art methods from a quality perspective. In Section 2.8.3 we compare the mining of *association rules* on text datasets. In Section 2.8.4 we end the experimental section with a *performance* analysis of FCI_{seq}. We remark that our implementation, datasets and experimental scripts are all publicly available[4].

2.8.1 Comparison on Synthetic Benchmark

Varying Noise Probability

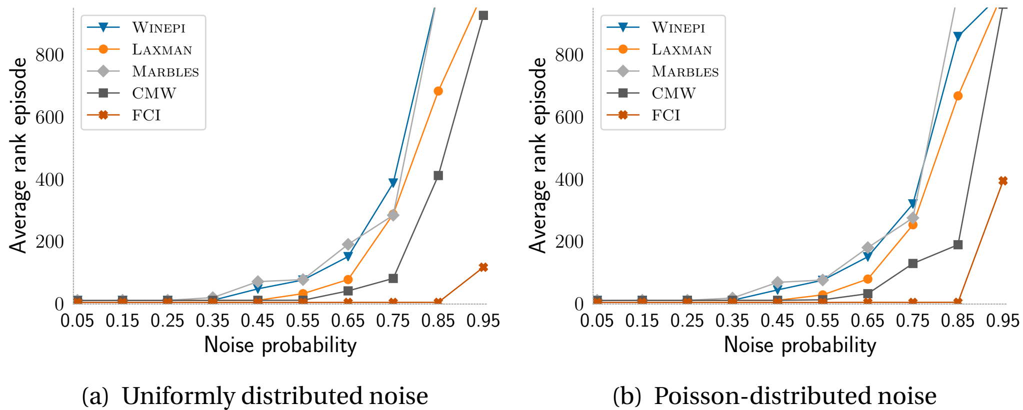
For the first experiment, we investigate the effect noise probability has on discovering a single embedded pattern in a synthetic sequence. The synthetic sequence consists of a total of 5000 events and 20 distinct event types. The maximum delay between two events is between 0 and 20. We embed a single serial episode with 4 elements. We then generate 10 variations, while varying the chance of a random event between each pattern embedding, that is p, between 0.05 and 0.95 in steps of 0.1. More details can be found in the original work of Zimmermann (2014a).

For FCI_{seq} we set *max_size* to 5 and *min_coh* to a value lower than the cohesion of the embedded pattern. For WINEPI, LAXMAN, MARBLES_W and CMW we set the *window* to be large enough for any pattern embedding, that is 50. For WINEPI, LAXMAN, MARBLES_W and CMW we set the *threshold* small enough that at least thousands of patterns are reported in a reasonable time, that is $t = 80$ for WINEPI, $t = 5$ for LAXMAN and CMW (which uses LAXMAN for episode generation), and $t = 1$ for MARBLES_W. In addition, we set *alpha*, that controls the scaling of the ranking based on compactness of window sizes, to 0.5 for CMW, and split the sequence into

[2] The algorithm was given no name by its authors.
[3] The implementation of CMW was provided by the author, but is not publicly available.
[4] https://bitbucket.org/len_feremans/fci_public

two equal parts: one for training and candidate generation, and one for testing if patterns are significant as done in the original paper (Tatti 2014a).

We run two variations: in the first, we assume that noise events and delays between both pattern and noise events are sampled using a uniform distribution, and for the second variation we generate noise events based on the Poisson distribution. The average rank of the discovered pattern within a synthetic sequence with varying noise probability is shown in Figure 2.2. Note that if an embedded pattern is not ranked within the top 1000 or is not found at all, we cap the rank to 1000, which is why we do not show these scores on the plots, as they would be misleading). Due to the inherent randomness of the Zimmermann generator, we run each experiment 5 times, and report the rank of the discovered episode(s) averaged over these 5 runs. We see that FCI$_{seq}$ consistently outperforms the other methods and always ranks the embedded patterns very highly, while other methods underperform in the presence of noise.



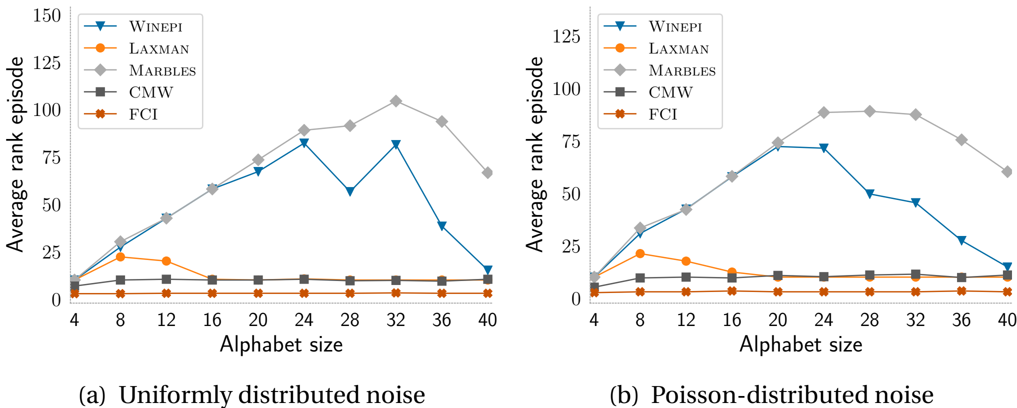
(a) Uniformly distributed noise (b) Poisson-distributed noise

Figure 2.2: Effect of *varying noise* on the discovery of a single episode by state-of-the-art methods and FCI$_{seq}$

Varying the Size of the Alphabet

In the second experiment, we investigate the effect of an increasing number of event types, denoted by M. We vary M between 4 and 40 in steps of 4. The synthetic sequences have a length of 5000 events, and the maximum delay between two events is uniformly between 0 and 20. We also embed a single serial episode with 4 elements. The uniform noise probability is fixed to 0.5. For mining the patterns we use the same parameters as in our previous experiment.

The average rank of the embedded pattern for varying alphabet size is shown in Figure 2.3. For the second variation of this experiment, we generate noise that is distributed using the Poisson distribution. The reported average rank varies less between methods than in the previous experiment. We see that LAXMAN, with its non-overlapping occurrence semantics, performs better then WINEPI. WINEPI and MARBLES also seem to perform better for an alphabet size of 40 than 20. Here, we conjecture that with an alphabet size of 40 the chance that random events co-occur more frequently than the embedded pattern is lower than if we have only 20 events. Once again, we see that FCI$_{seq}$ outperforms the other methods on every single experiment. When ranking on frequency subpatterns will be ranked first, which explains why FCI$_{seq}$ is the only method to rank the embedded pattern first.



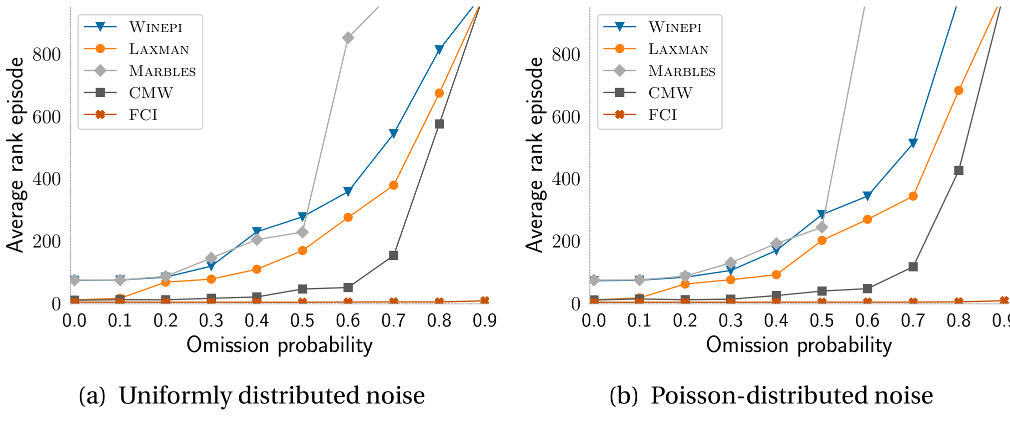
(a) Uniformly distributed noise (b) Poisson-distributed noise

Figure 2.3: Effect of *varying the alphabet size* on the discovery of a single episode by state-of-the-art methods and FCI$_{seq}$

Varying Probability of Omissions

In the third experiment, we vary the probability that a source event of the embedded pattern is *not* included, in order to mimic this type of failure that occurs in real-world datasets. We vary the failure probability o between 0 and 0.9 in steps of 0.1. As before, the synthetic sequences have a length of 5000 events consisting of 20 different events, the maximum delay between two events is distributed uniformly between 0 and 20, we embed a single serial episode with 4 elements, and the uniform noise probability is 0.5. For mining patterns, we use the same parameters as in our previous experiments.

The average rank of the pattern embedding for varying the probability of omissions is shown in Figure 2.4. Here, too, we perform a second variation of the experiment where the noise is distributed using the Poisson distribution. We see that our algorithm is not affected much by the omission probability, while all other methods begin to struggle as the omission probability rises.



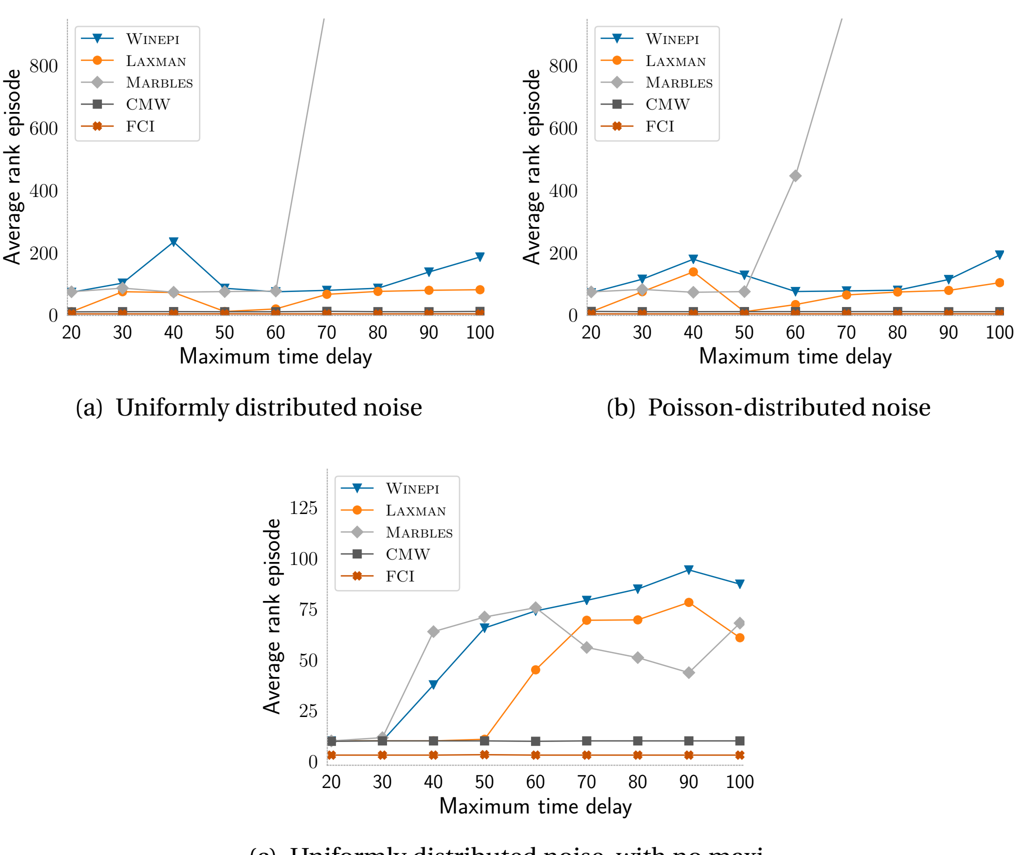
(a) Uniformly distributed noise (b) Poisson-distributed noise

Figure 2.4: Effect of *varying the probability of omitting events* of source episodes on the discovery of a single episode by state-of-the-art methods and FCI$_{seq}$

Varying Maximum Time Delay

In the fourth experiment, we study the effect of a larger time delay between consecutive events, and we vary the maximal delay g between 20 and 100 in steps of 10. We vary this delay both for regular sequence events and for the events belonging to pattern embeddings. For other generator parameters we use the same settings as before, that is a sequence length of 5000, alphabet size of 20, noise probability of 0.5, omission probability of 0 and a single serial episode with 4 elements. We also use the same parameters for mining as above, except for the *window* which we increase to 100.

The average rank of the pattern embedding for varying the maximal time delay is shown in Figure 2.5. For the second variation we generate noise that is distributed using the Poisson distribution. As discussed in Zimmermann (2014a), we also consider a third variant where the maximum delay between two consecutive events of an embedded pattern is not constrained by g. As in the previous experiments, FCI_{seq} consistently discovers the embedded pattern in the top 10. Unlike previous experiments, the performance of CMW is also comparable.



(a) Uniformly distributed noise

(b) Poisson-distributed noise

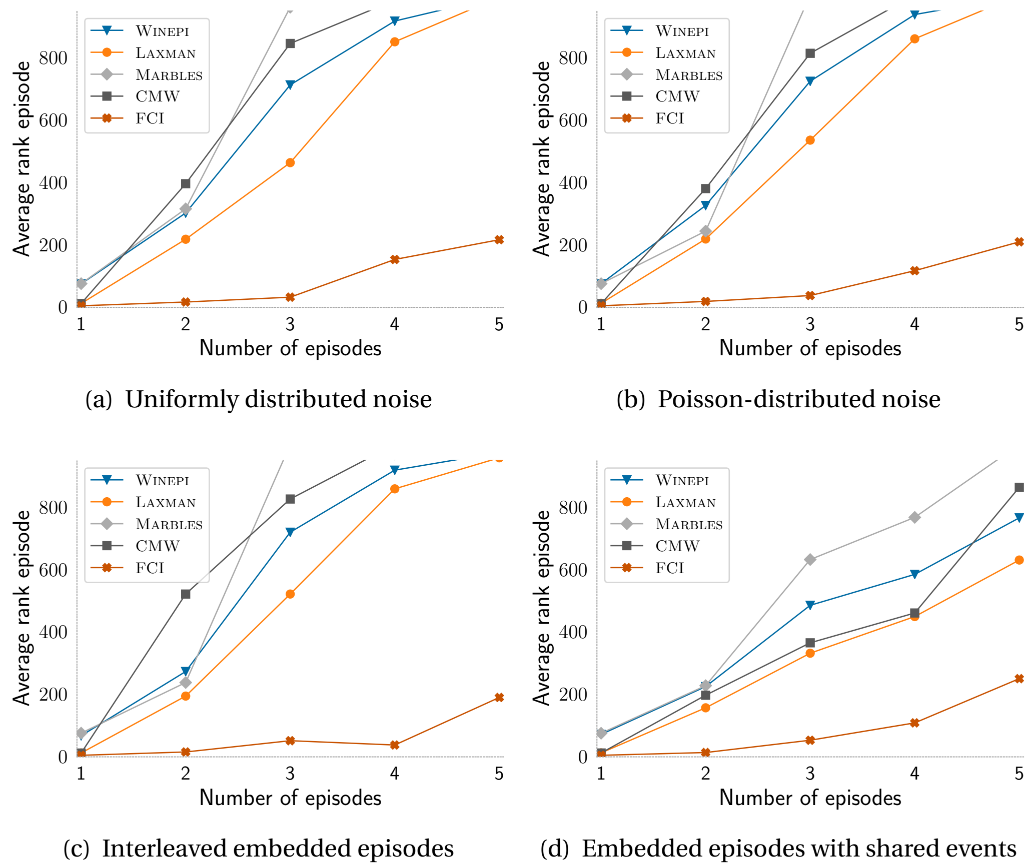
(c) Uniformly distributed noise, with no maximum delay between pattern events

Figure 2.5: Effect of *varying maximum time delay* on the discovery of a single episode by state-of-the-art methods and FCI_{seq}

Varying the Number of Patterns

In our fifth and final variation, we study the effect of increasing the number of patterns. Therefore, we vary the number of patterns n between 1 and 5. All other parameters (both for sequence generation and pattern mining) were set as in the previous experiments. The average rank for

the discovered patterns is shown in Figure 2.6. In addition to experimenting with both uniformly and Poisson-distributed noise, we also consider two other variations where embedded episodes are interleaved (occurrences of two episodes may overlap) and events are shared (two embedded patterns may contain the same event) (Zimmermann 2014a). We see that the number of different embedded patterns affects the performance of our algorithm more adversely than the parameters discussed previously, but FCI$_{seq}$ still outperforms all other methods by quite a margin.



(a) Uniformly distributed noise

(b) Poisson-distributed noise

(c) Interleaved embedded episodes

(d) Embedded episodes with shared events

Figure 2.6: Effect of *varying the number of episodes* and the discovery of them by state-of-the-art methods and FCI$_{seq}$

Conclusion

When looking at the results of the above experiments, we conclude that, on the Zimmermann benchmark, FCI$_{seq}$ clearly outperforms the state-of-the-art methods. Cohesion seems to be a more robust measure to a variety of artificially induced types of noise that occur in real-world settings. Compared to the frequency-based methods, this is not surprising, since frequency is a poor proxy for interestingness. While CMW seems to perform generally better than frequency-based methods, it also seems far less robust to the different types of variation than FCI$_{seq}$.

While it would be tempting to conclude FCI$_{seq}$ is superior on any dataset, we also acknowledge that our method has drawbacks. More specifically, when we consider the different parameters of the Zimmermann generator, we see two parameters, for which FCI$_{seq}$ would have trouble with respect to recovering patterns. The first parameter r controls if event types are *repeated* in an embedded pattern. Since FCI$_{seq}$ first mines itemsets, we do not generate any

patterns containing repeating items. A consequent issue is that we are unable to differentiate different, more complex, patterns, if the alphabet size is very small, e.g., given a DNA sequence with only 4 distinct items. A second parameter s controls if events are *shared* between multiple source episodes. Since we always consider the minimal window length at each item occurrence, having two (or more) embedded patterns share the same item will result in some occurrences of the shared item being far from the occurrences of other items in both patterns, and therefore in smaller values for cohesion for both patterns. As a consequence, the subset of the embedded patterns without the shared item would have a larger value for cohesion, and it is possible that the full pattern would not be recovered. The last issue that remains is that, since we consider all occurrences, our measure is not robust to outliers in minimal window length. These issues are, however, addressed by the method presented in Chapter 3.

2.8.2 Quality Comparison on Text Datasets

Datasets

We selected two text datasets in which the discovered patterns can be easily discussed and explained. *Species* contains the complete text of *On the Origin of Species by Means of Natural Selection* by Charles Darwin (Darwin 1859). *Trump* contains all tweets of President Donald Trump between January 1, 2017 and March 1, 2018 (Trump 2017). We preprocessed both sequences using the Porter Stemmer, removed stop words, transformed words to lower case, and removed any special characters. Since the *Trump* dataset consists of many short sequences, we appended all tweets to create a single long sequence. In addition, we also removed HTML content, such as URLs and entities. After preprocessing, the *Species* dataset has a sequence length of $|S| = 85\,447$ and contains 5 547 distinct items. For *Trump*, the *single* sequence length is $|S| = 27\,837$ and contains 4 061 distinct items.

Cohesive Itemsets

For each of the existing methods, we set the frequency threshold low enough in order to generate thousands of patterns. For FCI$_{seq}$, we did the same with the cohesion threshold. We then sorted the output on the respective quality measures — the sliding window frequency for WINEPI, the non-overlapping minimal window frequency for LAXMAN, the weighted window frequency for MARBLES$_{W}$, the leverage-based score for CMW, and cohesion for FCI$_{seq}$. For FCI$_{seq}$, we used the sum of support of individual items making up an itemset as the second criterion for ranking. For other methods pattern size is used as a second criterion for ranking. Finally, we use alphabetical order of patterns to break ties in all four methods. The frequency threshold was set to 30 for WINEPI, 5 for LAXMAN, and 1 for MARBLES$_{W}$ in both datasets, with the sliding window size set to 15. For CMW we set *alpha* to 0.5 and split the sequence in two: the first half is used for discovering patterns while the second half is used for testing whether patterns are significant (Tatti 2014a). CMW uses the frequent episodes produced by LAXMAN as input. We run FCI$_{seq}$ with the cohesion threshold set to 0.015 for *Species* and 0.02 for *Trump*, and we set the support threshold to 5 for both datasets. Since none of the state-of-the-art methods produced any itemsets consisting of more than 6 items, we set the *max_size* parameter to 6 to reduce runtimes.

The top 5 patterns discovered by the different methods are shown in Table 2.2. We can see that there are clear differences between the patterns discovered by FCI$_{seq}$ and CMW, and those discovered by the frequency-based methods, which produce very similar results. First of all, the patterns ranked first and second in our output for the *Species* dataset are of size 3, which

Table 2.2: Top 5 *itemsets* discovered by FCI$_{seq}$ and the state-of-the-art methods on *Species* and *Trump*

	FCI$_{seq}$	WINEPI	LAXMAN	MARBLES$_W$	CMW
Species	del, fuego, tierra	natur, select	natur, select	natur, select	absenc, island, mammal, ocean, terrestri
	facit, natura, saltum	speci, varieti	form, speci	speci, varieti	altern, glacial, north, period, south
	del, fuego	form, speci	speci, varieti	distinct, speci	bat, island, mammal, ocean, speci, terrestri
	del, tierra	natur, speci	natur, speci	form, speci	bat, island, mammal, ocean, terrestri
	facit, saltum	distinct, speci	distinct, speci	condit, life	cross, fertil, hybrid, mongrel, offspr, varieti
Trump	puerto, rico	fake, new	fake, new	fake, new	ab, japan, minist, prime
	hunt, witch	cut, tax	cut, tax	cut, tax	high, hit, market, stock, time
	harbor, pearl	america, great	america, great	america, great	abc, cnn, fake, nbc, new
	lago, mar	great, make	great, peopl	america, make	alabama, big, luther, strang, vote
	arabia, saudi	america, make	great, make	great, make	lowest, market, stock, unemploy, year

would be theoretically impossible for WINEPI and MARBLES$_W$, and highly unlikely for LAXMAN, since all three use anti-monotonic quality measures. Second, we observe that the patterns we discover are in fact quite rare in the dataset, but they are very strong since all occurrences of these patterns are highly cohesive. Concretely, the phrase *tierra del fuego* occurs seven times in the book, and none of these words occurs anywhere else in the book. The value of this pattern is therefore quite clear — if we encounter any one of these three words, we can be certain that the other two can be found nearby. A similar argument holds for the expression "Natura non facit saltus", Latin for "nature does not make jumps", of which "non" is considered a stopword, and removed during preprocessing. CMW also discovers interesting, albeit *different* itemsets. CMW ranks *tierra del fuego* 32nd and *natura facit saltus* 27th. The top pattern of CMW relates to chapter XIII in the book, with a subsection titled "absence of bathracians and terrestrial mammals on oceanic islands". FCI$_{seq}$ is unable to find this pattern, as the cohesion is very low overall since *absence, terrestri, mammal, ocean, island* infrequently co-occur together in the book. If we would, however, segment this very long book, and run FCI$_{seq}$ on each individual chapter, the cohesion, *local* to chapter XIII, would also be high for this itemset.

In the *Trump* dataset, the top 5 patterns produced by FCI$_{seq}$ all have a cohesion of 1, which indicates that they always occur next to each other in tweets, in this case in 27, 23, 8, 7 and 7 tweets, respectively. Like *tierra del fuego*, these are examples of itemsets that are highly correlated but occur too infrequently to rank highly in existing state-of-the-art methods. For example, *pearl harbor*, *saudi arabia* and *mar (a) lago* do not occur in the top 300 of the existing state-of-the-art methods.

We conclude that in order to find less frequent, but strongly correlated patterns such as *tierra del fuego* or *mar (a) lago* with existing state-of-the-art methods, the user would need to wait a long time before a huge output was generated, and would then need to trawl through

thousands of itemsets in the hope of finding them. FCI_{seq}, on the other hand, ranks them at the very top. From the perspective of the frequency-based methods, top patterns typically consist of words that occur very frequently in the dataset, regardless of whether the occurrences of the words making up the itemset are correlated or not. The top patterns reported by CMW also seem very interesting, but quite different from those produced by FCI_{seq}. A disadvantage of CMW is that only half of the sequence is available for training, causing the method to miss out on any patterns that only occur in the test part. A second disadvantage are the so-called *free-rider episodes* where an item added independently to an existing high-leverage episode also has a high score. For a complete picture, we provide the top 25 itemsets discovered by all five methods in both datasets in Appendix A. While the top patterns are different, there is still some overlap between the output generated by the various methods. For example, the pattern *natur(al) select(ion)*, ranked first in the *Species* dataset by the frequency-based methods, was ranked 16th by FCI_{seq}, which shows that our method is also capable of discovering very frequent patterns, as long as they are also cohesive. Table 2.3 shows the size of the overlap between the itemsets discovered by FCI_{seq} and those discovered by the other methods. We compute the size of the overlap within the top k itemsets for each method, for varying values of k.

Table 2.3: *Overlap* in the top k itemsets discovered by FCI_{seq} and the state-of-the-art methods on *Species* and *Trump*

	overlap@k	WINEPI	LAXMAN	MARBLES$_W$	CMW
Species	100	12	13	11	2
	500	28	35	25	2
	1 000	45	53	38	4
Trump	100	15	14	16	0
	500	37	47	34	18
	1 000	54	67	50	30

Representative Sequential Patterns

For evaluating representative sequential patterns we show the top 5 sequential patterns discovered by FCI_{seq} and set the occurrence ratio threshold for sequential patterns, *min_or*, to 0.7, and thus only report sequential patterns for itemsets where the sequential pattern occurs in at least 70% of itemset occurrences. We set *min_coh* to 0.015 (0.02 for *Trump*), $\theta = 5$ and *max_size* = 6, and compare the discovered representative sequential patterns with total orders

Table 2.4: Top 5 *sequential patterns* discovered by FCI_{seq} on *Species*

sp	$C(X)$	$support(X)$	$occ_ratio_{se}(sp)$
tierra, del, fuego	1.0	21	1.0
natura, facit, saltum	1.0	18	1.0
del, fuego	1.0	14	1.0
tierra, del	1.0	14	1.0
facit, saltum	1.0	12	1.0

reported by state-of-the-art methods. The results for FCI_{seq} for *Species* are shown in Table 2.4. The patterns are first sorted on cohesion, then on occurrence ratio, then on support, and finally alphabetically, if all other measures are equal. These results show that the items making up the most cohesive itemsets always occur in a specific order. Therefore, when this is the case, the representative sequential patterns form a more informative way to represent the most interesting patterns.

For FCI_{seq}, WINEPI, LAXMAN, MARBLES_W and CMW the top 5 sequential patterns are shown in Table 2.5 for both datasets. Due to the *min_or* threshold of 0.7, we only report a representative sequential pattern for those itemsets that actually have one. In fact, out of the 1 130 discovered cohesive itemsets in *Species*, only 21 have a representative sequential pattern, for all others there is no single specific order that would be representative for the occurrences of the itemset. In *Trump*, only 22 out of 16372 itemsets had a representative sequential pattern. For both *Species* and *Trump*, the top 5 sequential patterns had an occ_ratio_{se} of 1, indicating that *all* occurrences of the underlying itemset came in the order defined by the sequential pattern, clearly demonstrating the usefulness of outputting these patterns. Conversely, for itemsets where the order is not important, our method outputs no sequential pattern at all. Unlike FCI_{seq}, the frequency-based methods again rank many spurious patterns highly. Note, for example, that all three methods output both *variety speci(es)* and *speci(es) variety* in the top 10. This clearly demonstrates that, while the two words often co-occur due to them both being very frequent, there is no sequential relationship between them. Furthermore, similarly to itemsets, we remark that the top 1000 of all frequency-based methods contained very few ($< 5\%$) sequential patterns consisting of more than two items, and none of them ranked sequential patterns *tierra del fuego* and *natura (non) facit saltum* in the top 1000. On the other hand, the most interesting sequential patterns found by the frequency-based methods are ranked highly

Table 2.5: Top 5 *sequential patterns* discovered by FCI_{seq} and the state-of-the-art methods on *Species* and *Trump*

	FCI_{seq}	WINEPI	LAXMAN	MARBLES_W	CMW
Species	tierra, del, fuego	natur, select	natur, select	natur, select	struggl, exist, geometr, ratio, increas
	natura, facit, saltum	varieti, speci	varieti, speci	distinct, speci	variat, superven, earli, ag, inherit
	del, fuego	speci, varieti	speci, form	varieti, speci	form, life, chang, simultan, world
	tierra, del	distinct, speci	speci, varieti	condit, life	variat, superven, earli, inherit, ag
	facit, saltum	speci, form	form, speci	speci, varieti	steril, speci, cross, hybrid, offspr
Trump	puerto, rico	fake, new	fake, new	fake, new	stock, market, hit, time, high
	witch, hunt	tax, cut	tax, cut	tax, cut	job, stock, market, time, high
	pearl, harbor	america, great	america, great	america, great	greatest, witch, hunt, histori
	mar, lago	make, america	make, great	make, america	stock, market, hit, high
	saudi, arabia	make, great	make, america	unit, state	presid, moon, south, korea

by FCI$_{seq}$, too. For example, *natur(al) select(ion)*, *fake new(s)* and *tax cut(s)* can all be found in our top 20 sequential patterns for the respective dataset.

For sequential patterns, we see that the re-ranking of CMW for sequential patterns candidates generated by LAXMAN, produces a more interesting set of patterns. The top sequential pattern *struggl(e) (for) exist(ence) geometr(ic) ratio (of) increas(e)* appears in chapter III. FCI$_{FCI}$ does not rank this pattern highly, since *struggle* also often co-occurs with *life*, or on its own as verb, causing large minimal windows for these occurrences. Likewise, the word *increase* (or variants like *increasing* with the same stem) are very common, and frequently occur in chapter III, without any instance of the other words nearby. Accordingly, the interestingness of this pattern is very low concerning our proposed cohesion measure. In *Species*, CMW also reports both *tierra del fuego* and *natura (non) facit saltum* in the top 25. For *Trump*, the situation is different, in that some top patterns found by FCI$_{seq}$ are ranked rather low by CMW, e.g. *pearl harbor* is ranked 9921st.

Dominant Episodes

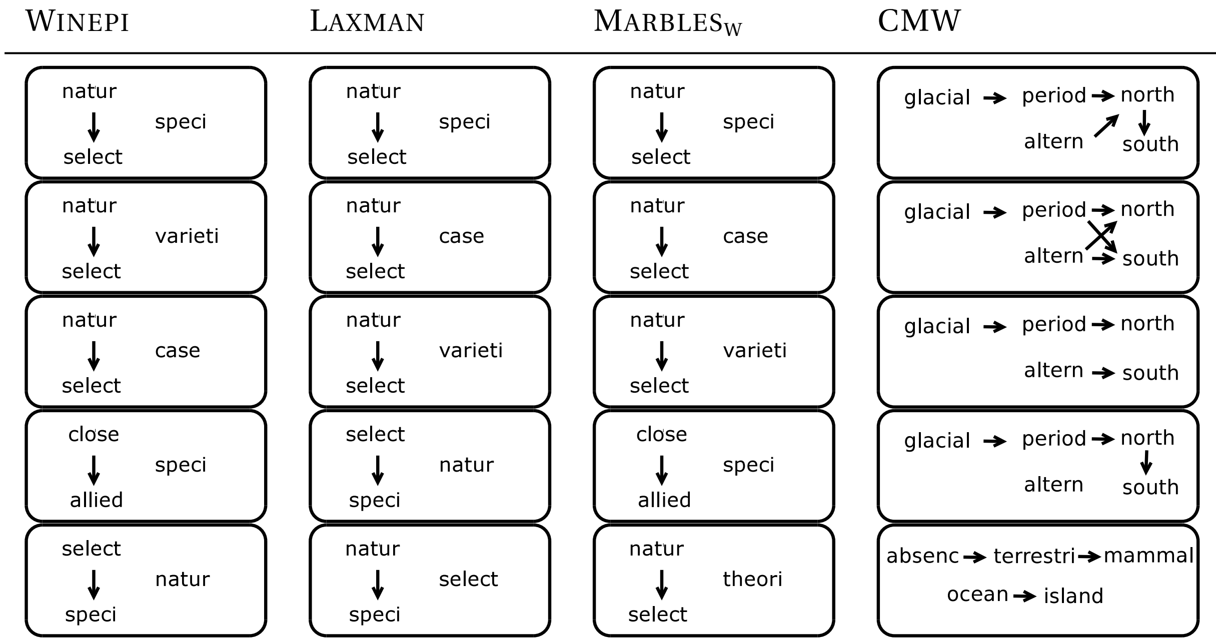
To discover dominant episodes we run FCI$_{seq}$ and set the minimum occurrence ratio for episodes, min_por, to 0.7, $min_coh = 0.015$ (0.02 for *Trump*), $\theta = 5$ and $max_size = 5$. We thus report episodes $G = (V(G), E(G))$ for itemsets X, where $V(G) = X$ and the partial order defined by $E(G)$ occurs in at least 70% of itemset occurrences. In *Species*, the dominant episode for 658 out of 1 130 cohesive itemsets was either a sequential pattern (a total order) or the itemset itself (no order). For the remaining 472 itemsets, the dominant episode was a partial order. Note that for itemsets of size 2, the only possible episodes represent either an itemset or a sequential pattern. As a result, all the partial orders of interest consisted of three or more items. Therefore, we exclude itemsets and sequential patterns from the episode output.

The results of FCI$_{seq}$ for *Species* are shown in Table 2.6. Note that the episodes are ranked on the cohesion of the underlying itemset, after filtering. The episodes containing *hexagon(al), prism, pyramid, rhomb, sphere* are due to a section in the book discussing the making of the honeycomb structure of bees, while the episode containing *leptali(s), ithomia, mimick* is due to a section where the similarities of these two butterfly species are discussed. While the specific semantics are less of interest in text datasets, what is interesting is that these are partial orders that hold for more than 70% of occurrences.

Table 2.6: Top 5 *dominant episodes* discovered by FCI$_{seq}$ on *Species*

G	$C(X)$	$support(X)$	$occ_ratio_{po}(G)$
leptali ↓ ithomia; mimick	0.096	20	0.75
hexagon ↓ prism; sphere	0.022	38	0.92
prism ↙ pyramid, prism ↘ rhomb; hexagon; sphere	0.021	44	0.86
gener ↓ speci; natur; form; distinct	0.020	46	0.76
prism ↙ pyramid, prism ↘ rhomb; hexagon	0.020	29	0.86

Table 2.7: Top 5 *episodes* discovered by the state-of-art methods on *Species*



Episodes reported by state-of-the-art methods on *Species* are shown in Table 2.7. Here, too, we omit episodes defining either a total order or no order at all. While the episode output of FCI$_{seq}$ and CMW provides additional information about partial orders present in the occurrences of itemsets that have no representative sequential pattern, the three frequency-based methods produce various combinations of very frequent items, which is not very informative. Finally, note that by using the *min_por* threshold, we ensure that we produce a single dominant episode per underlying itemset, which is representative of the occurrences of that itemset. Other methods produce many partial orders for the same itemset which can result not only in spurious patterns being discovered, but also in an undesirably large output size, i.e., for CMW the top 100 episodes consist of variations with different edges of the itemsets of episodes shown in Table 2.7.

2.8.3 Association Rules

Text Datasets

Using FCI$_{seq}$ we generate association rules for both datasets with the confidence threshold *min_conf* set to 0.7, and parameters *min_coh*, *max_size* and θ as defined in the previous section. We again compare rules with WINEPI, MARBLES$_W$, and with MARBLES$_M$, which uses a confidence measure based on non-overlapping minimal windows (as defined by LAXMAN). CMW is only used for re-ranking episodes, not mining association rules. We set *min_conf* = 0.7 for the state-of-art methods, and the frequency threshold to 40 for WINEPI, 10 for MARBLES$_M$, and 1 for MARBLES$_W$ in both datasets, with the sliding window size set to 15.

We slightly adjusted the underlying implementation of the state-of-art methods, namely CLOSEPI, and made the modified code publicly available in our repository. Since CLOSEPI mines *general episodes*, instead of only parallel episodes, or itemsets, and then generates rules with potentially both general episodes on the left- and right-hand side, this causes an order-of-magnitude more rules, and requires additional computing resources. Therefore, we made sure

that parallel episodes, closed by a partial episode, are not removed, and instead removed all non-parallel episodes before mining association rules.

The top 5 rules, ranked on confidence, for *Species* and *Trump* for all methods are shown in Table 2.8. Rules were ranked first on the confidence measure related to each method, and then on the support of the antecedent to break ties. For *Species* and *Trump* the top 5 rules for FCI_{seq} have a confidence of 1.0. Most of these top rules consist of itemsets that are fully cohesive, meaning that if one of the items occurs, the others always occur next to them. Only *migrat(ation)* ⇒ *chain* is different since the underlying itemset is not fully cohesive, but the association rule with *migrat(ation)* as antecedent is fully confident. Interestingly, MARBLES_W and MARBLES_M also report rules of fully cohesive itemsets ranked in the top 5, such as *tierra del fuego* and *natura facit saltum*. Unlike itemsets ranked on frequency, the ranking on confidence produces quite different results between all methods, especially between the three different frequency-based approaches. While all methods find very interesting rules, a disadvantage of the three state-of-the-art methods is that they often rank rules with frequent items such as *speci(es), natur(al)* or *select(ion)* as the consequent very highly, when this is, in most cases, due to these items accidentally occurring in the vicinity of the antecedent, and not due to an actual association. On the other hand our method fails to discover rules such as *divis(ion), kingdom* ⇒ *anim(al)*. The phrase "division of the animal kingdom" occurs about 10 times in *Species*. FCI_{seq} does not report the itemset or any subset or resulting rules, because when we consider all occurrences of items in the antecedent — both *kingdom* and *divis(ion)* — and then compute the average minimal window lengths, in this instance, cohesion is very low (< 0.001), or, in other words, the majority of the occurrences of the three items do not co-occur anywhere. Once again, we provide the top 25 rules of both datasets in the Appendix.

Table 2.8: Top 5 *rules* discovered by FCI_{seq} and the state-of-the-art methods on *Species* and *Trump*

	FCI_{seq}	WINEPI	MARBLES_M	MARBLES_W
Species	fuego, tierra ⇒ del	divis, kingdom ⇒ anim	hive ⇒ bee	divis, kingdom ⇒ anim
	del, fuego ⇒ tierra	averag, genera ⇒ speci	mivart ⇒ mr	fuego, tierra ⇒ del
	del, tierra ⇒ fuego	cuckoo, lai, nest ⇒ egg	case, select, structur ⇒ natur	independ, ordinari ⇒ view
	facit, natura ⇒ saltum	varieti, zone ⇒ inter-medi	candol ⇒ de	natura, saltum ⇒ facit
	natura, saltum ⇒ facit	genera, present, vari-eti ⇒ speci	case, organ, select ⇒ natur	inherit, superven ⇒ earli
Trump	puerto⇒ rico	hit, stock⇒ market	cut, reform⇒ tax	honor, minist⇒ prime
	rico⇒ puerto	high, hit, stock⇒ market	puerto⇒ rico	confer, joint⇒ press
	hunt⇒ witch	bill, reform, tax⇒ cut	rico⇒ puerto	immigr, merit⇒ base
	witch⇒ hunt	biggest, cut, histori⇒ tax	witch⇒ hunt	greatest, hunt⇒ witch
	migrat⇒ chain	ab, prime⇒ minist	hunt⇒ witch	donald, proclaim⇒ trump

Character Sequence Datasets

As a second experiment, we run association rule mining on text split on each character. We are interested in finding association rules between letters that are specific within each language — in this case, English, French and Dutch. We use the complete text of *David Copperfield* by Charles Dickens in English and translations in French and Dutch (Dickens 1850). We removed special characters, transformed words to lower case, tokenised the text on individual characters and added the '_' symbol to denote spaces between words. We limited the three sequences to the first $|S| = 500\,000$ characters. The dictionary consists of 26 letters and '_' .

We run FCI$_{seq}$ with $min_coh = 0$, $max_size = 4$, $minsup = 50$ and $min_conf = 0.3$. In Table 2.9 we show the top 5 rules, ranked on confidence, discovered for each language, split into three categories. We first show rules consisting of two letters, then rules consisting of three letters, and, finally, rules containing the space between words, or '_'. We see that some reported patterns are common in all three languages, and some patterns are discriminative for a specific language. For example $q \Rightarrow u$ is a typical combination found in all three languages, as q is almost always followed by a u. A typical Dutch rule is $j \Rightarrow i$, where *ij* is a very common combination of letters, while rule $j \Rightarrow e$ is very specific for French. Rule $y \Rightarrow _$ is typical for English and Dutch, where y often occurs either at the start or at the end of a word, and rarely in the middle. The same holds for $j \Rightarrow _$ in French, where j is mostly found at the start of the word. This experiment confirms that our method finds valuable association rules, and tends not to rank spurious rules highly.

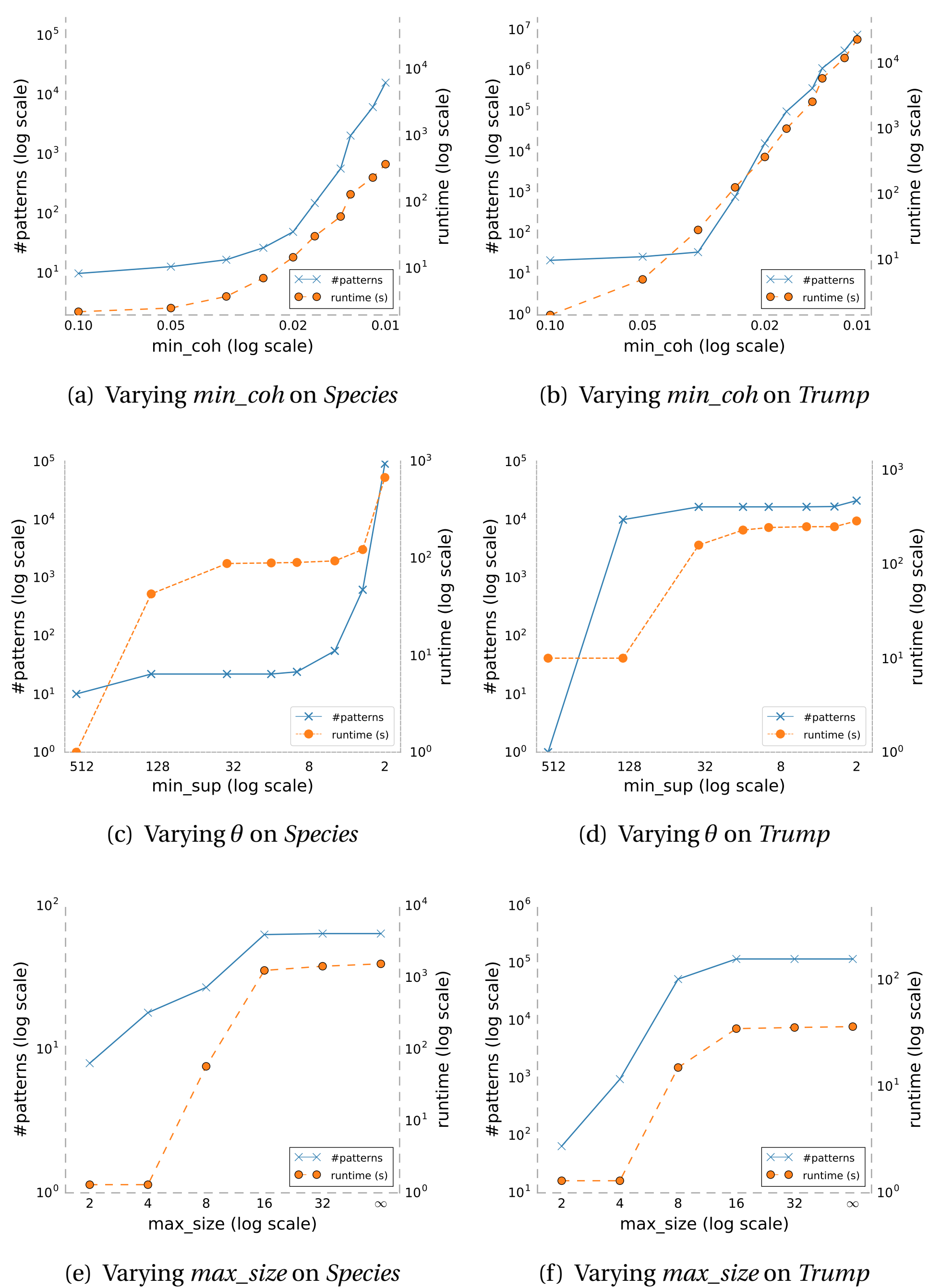
Table 2.9: Top 5 *rules* discovered by FCI$_{seq}$ on sequences of characters in different languages

Category	ENGLISH	FRENCH	DUTCH
Two letters	$c(q \Rightarrow u) = 0.984$	$c(q \Rightarrow u) = 0.980$	$c(q \Rightarrow u) = 0.661$
	$c(v \Rightarrow e) = 0.686$	$c(j \Rightarrow e) = 0.609$	$c(j \Rightarrow i) = 0.625$
	$c(x \Rightarrow e) = 0.616$	$c(g \Rightarrow e) = 0.589$	$c(b \Rightarrow e) = 0.605$
	$c(r \Rightarrow e) = 0.432$	$c(d \Rightarrow e) = 0.542$	$c(n \Rightarrow e) = 0.603$
	$c(z \Rightarrow e) = 0.416$	$c(r \Rightarrow e) = 0.540$	$c(g \Rightarrow e) = 0.594$
Three letters	$c(q \Rightarrow i, u) = 0.524$	$c(q \Rightarrow u, e) = 0.678$	$c(q \Rightarrow i, u) = 0.667$
	$c(q \Rightarrow u, e) = 0.520$	$c(b \Rightarrow a, e) = 0.357$	$c(q \Rightarrow i, n) = 0.626$
	$c(q \Rightarrow t, u) = 0.399$	$c(q \Rightarrow u, i) = 0.348$	$c(q \Rightarrow u, n) = 0.576$
	$c(v \Rightarrow a, e) = 0.362$	$c(l \Rightarrow a, e) = 0.346$	$c(q \Rightarrow u, e) = 0.485$
	$c(q \Rightarrow i, e) = 0.357$	$c(v \Rightarrow a, e) = 0.346$	$c(q \Rightarrow r, e) = 0.483$
Letters near word boundary	$c(y \Rightarrow _) = 0.918$	$c(j \Rightarrow _) = 0.924$	$c(y \Rightarrow _) = 0.979$
	$c(w \Rightarrow _) = 0.863$	$c(q \Rightarrow u, _) = 0.906$	$c(q \Rightarrow _) = 0.894$
	$c(d \Rightarrow _) = 0.835$	$c(q \Rightarrow _) = 0.824$	$c(x \Rightarrow _) = 0.846$
	$c(q \Rightarrow _, u) = 0.794$	$c(d \Rightarrow _) = 0.790$	$c(z \Rightarrow _) = 0.800$
	$c(b \Rightarrow _) = 0.790$	$c(q \Rightarrow u, e, _) = 0.785$	$c(m \Rightarrow _) = 0.781$

2.8.4 Performance Analysis

Effect of Hyperparameters on Runtime

We tested the behaviour of our itemset mining algorithm when varying the cohesion, support and size thresholds on the two text datasets. We set default values for *max_size* and θ to 4 or 5 and set *min_coh* to 0.02. We then investigate the effect on the number of patterns and the



(a) Varying *min_coh* on *Species*

(b) Varying *min_coh* on *Trump*

(c) Varying θ on *Species*

(d) Varying θ on *Trump*

(e) Varying *max_size* on *Species*

(f) Varying *max_size* on *Trump*

Figure 2.7: Effect of *min_coh*, θ and *max_size* thresholds on the number of patterns and runtime

runtime, in log scale, of varying each threshold while selecting the default value for the others. For the experiment with varying *max_size*, we set θ to 350 for *Species* and to 150 for *Trump*. The results are shown in Figure 2.7.

As expected, we see that the number of patterns increases as the cohesion and support thresholds are lowered. In particular, when the cohesion threshold is set too low, the size of the output explodes, as even random combinations of frequent items become cohesive enough. However, as the support threshold decreases, the number of patterns stabilises since rarer items typically only make up cohesive itemsets with each other, so only a few new patterns are added to the output (when we lower the support threshold to 2, we see another explosion as nearly the entire alphabet is considered frequent).

In all settings, it took no more than a few minutes to find tens of thousands of patterns. Note that with reasonable support and cohesion thresholds, we could even set the *max_size* parameter to ∞ without encountering prohibitive runtimes, allowing us to discover patterns of arbitrary size (in practice, the size of the largest pattern is limited due to the characteristics of the data, so output size stops growing at a certain point). The state-of-the-art methods use a relevance window that defines how far apart two items may be in order to be still considered part of a pattern. As a consequence they can never find patterns of arbitrary size. For example, using a window of size 15 implies that no pattern consisting of more than 15 items can ever be discovered.

Effect of Pruning

In a second performance experiment we vary the maximum size of patterns, and report the number of candidates visited by the main DFS routine (Algorithm 2.2) with pruning versus the number of candidates that is *theoretically possible* without pruning. Given an alphabet of $|\Omega|$ items, the number of itemsets of length 2 up to *max_size* that is theoretically possible is given by $\sum_{k=2}^{max_size} \binom{|\Omega|}{k}$. We run $\textsc{Fci}_{seq}$ on *Species* and set θ to 5, resulting in $|\Omega| = 5\,547$ different words, and set *min_coh* to 0.5. In Table 2.10, we report the number of candidates visited versus the number of possible candidates for varying *max_size*. Remark that running up to *max_size* = 48 took only 41 minutes on a laptop. We conclude that pruning on cohesion is effective in narrowing the search in an otherwise intractable search space.

Table 2.10: Effect of varying *max_size* on the *number of candidates* enumerated by $\textsc{Fci}_{seq}$ on *Species*

max_size	Candidates visited by $\textsc{Fci}_{seq}$	Theoretically possible number of candidates
8	4.8×10^{6}	1.3×10^{22}
16	5.0×10^{6}	1.3×10^{40}
24	5.4×10^{6}	2.4×10^{56}
32	8.0×10^{6}	2.8×10^{71}
40	2.3×10^{7}	4.4×10^{85}
48	1.3×10^{8}	1.3×10^{99}

Runtime for Other Patterns and Association Rules

In a final performance experiment, we study the extra time required for mining representative sequential patterns, dominant episodes and association rules, in addition to cohesive itemsets. We set the threshold values to 0.015 or 0.02 for *min_coh* and 4 or 5 for θ and *max_size*. For mining sequential patterns and episodes we set *min_or* and *min_por* to 0.5 and for rule mining we set *min_conf* to 0.7. The runtime of additionally mining representative sequential patterns, dominant episodes and association rules on *Species* and *Trump* is shown in Figure 2.8.

Since sequential pattern, episode and association rule mining is triggered for each cohesive itemset, additional runtime costs for mining other types of patterns is relative to the number of reported cohesive itemsets. If the number of cohesive itemsets is small, such as for *Species* with 1 339 itemsets, the additional time required for finding sequential patterns or episodes is small, compared to the time for only mining cohesive itemsets. For *Trump* the number of

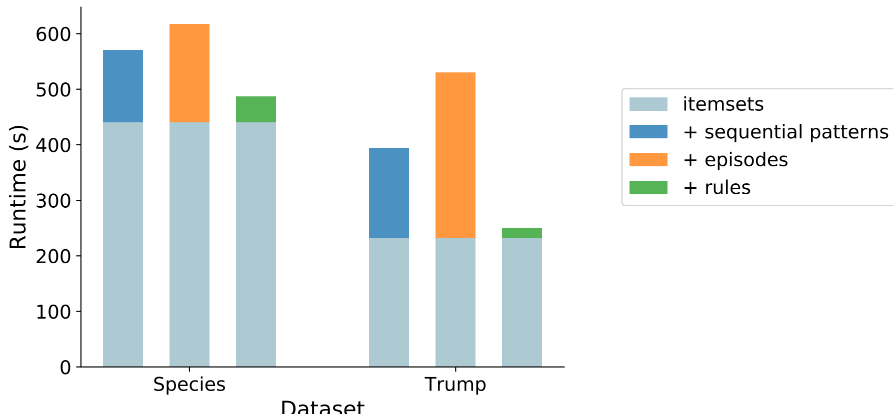


Figure 2.8: Effect of mining representative sequential patterns, dominant episodes and association rules on runtime on *Species* and *Trump*

itemsets is larger, that is 16393, and the additional time needed for finding sequential patterns is naturally higher, and even higher for dominant episodes (note, however, that episode mining requires the execution of the sequential pattern mining phase, even if sequential patterns are not required for output). For mining rules, the additional runtime cost for mining confident rules based on cohesive itemsets is relatively small for both datasets. This is mainly due to the efficient computation of confidence, as described in Section 2.6. We conclude that our algorithms perform efficiently in a variety of settings.

2.9 Related Work

We have examined the most important related work in Section 2.1, and experimentally compared our work with the existing state-of-the-art methods in Section 2.8. Here, we place our work into the wider context of sequential pattern mining.

Frequent Pattern Mining

At the heart of most pattern mining algorithms is the need to reduce the exponential search space into a manageable subspace. When working with an anti-monotonic quality measure, such as frequency, the Apriori property can be deployed to generate candidate patterns only if some or all of their subpatterns have already proved frequent. This approach is used in both breadth-first-search (BFS) and depth-first-search (DFS) approaches, such as APRIORI (Agrawal and Srikant 1994), ECLAT (Zaki 2000) and FP-GROWTH (Han et al. 2004) for itemset mining in transaction databases, GSP (Srikant and Agrawal 1996), SPADE (Zaki 2001), BIDE (Wang and Han 2004) and PREFIXSPAN (Pei et al. 2004) for sequential pattern mining in sequence databases, or WINEPI (Mannila et al. 1997) and MARBLES (Cule et al. 2014) for episode mining in event sequences.

Constraint Frequent Pattern Mining

In other related work, Méger and Rigotti (2004) and Pei et al. (2007) incorporated *temporal constraints* in pattern mining. Two types of constraints are either a maximal window constraint, i.e., the maximum elapsed time between the first and last event of an occurrence of the pattern, and a maximal gap constraint, i.e., the maximum elapsed time between any two consecutive

events in each occurrence. The main difference between our work and these approaches is that we always consider *all* event occurrences, instead of counting only pattern occurrences satisfying the temporal constraints. Furthermore, we do not only count occurrences, but use the window length of each occurrence as a weight used in ranking the *top* patterns. Finally, we remark that Zimmermann (2014a) recently proposed a benchmark study that compared different episode miners. We have adopted his framework (in Section 2.8.1) and found that the reported experimental results of the gap constraint techniques are considerably worse than the results of FCI$_{seq}$ with respect to recovering patterns embedded in the data.

Mining Interesting Patterns

For computational reasons, non-anti-monotonic quality measures are rarely used or are used to re-rank the discovered patterns in a post-processing step. Tatti (2014a) proposed a way to measure the significance of an episode by comparing the lengths of its occurrences to expected values of these lengths if the occurrences of the patterns' constituent items were scattered randomly. In a later work, Tatti (2015) introduced the EPIRANK algorithm to re-rank episodes in a dataset consisting of multiple sequences based on *leverage* (Webb 2010). Both methods, however, use the output of an existing frequency-based episode miner (Tatti and Cule 2012), and then compute the new measures for the discovered patterns. In this way, the rare patterns, such as those discussed in Section 2.8, will once again not be found. Our FCI$_{seq}$ algorithm falls into the DFS category, and the proposed quality measure is not anti-monotonic, but rather than evaluating it in a post-processing step, we rely on an alternative pruning technique to reduce the size of the search space. We believe the additional computational effort to be justified, as we manage to produce intuitive results, with the most interesting patterns, which existing state-of-the-art methods sometimes fail to discover at all, ranked at the very top.

Petitjean et al. (2016) proposed an alternative measure of interestingness for ranking sequential patterns, which does not satisfy the anti-monotonicity property either, and an algorithm (SKOPUS) to *directly* enumerate candidate sequential patterns satisfying the interestingness measure. This measure, like EPIRANK, is based on leverage and compares the support against the expected support assuming independence. Unlike our approach, SKOPUS takes as input a database of many, typically short, sequences, rather than a single sequence.

Multiple authors have also stepped away from mining all frequent patterns and rather tried to reduce the number of patterns often based on information theoretic approaches such as the *Minimal Description Length* (Grünwald 2007), thereby producing a smaller set of patterns that covers the sequence, or, in most cases, a database of many sequences. Methods such as SQS (Tatti and Vreeken 2012), GOKRIMP (Hoang et al. 2014), and ISM (Fowkes and Sutton 2016) follow this approach. These methods take a database of typically short sequences as input, which is different from our approach. Another key difference is that rather than enumerating as few candidates as possible and then selecting the best candidates according to an interestingness measure, they employ *heuristic search* to incrementally build a set of non-redundant patterns, instead of trying to enumerate an exact set of patterns, based on a definition of interestingness.

Natural Language Processing

Finding interesting pairs (or n-grams) of co-occurring words in natural language has also been extensively studied by the Natural Language Processing community (Manning and Schütze 1999). Here the goal is to find *collocations*, that is co-occurring words that are typical in a corpus, such as *strong tea* or *the rich and the famous*. Specific to the Natural Language domain,

collocations are also characterised by the semantic concept of limited compositionality, that is the meaning of the collocation of words is only weakly related to the meaning of the individual words, e.g., *strong* has a different meaning in the phrase *strong tea* than in *strong man*. In the context of mining collocations, different methods start with counting all frequent bi-grams or n-grams within a fixed window, after filtering words based on part of speech tags (Justeson and Katz 1995). Church and Mercer (1993) and Manning and Schütze (1999) have proposed ranking collocations based on various statistics, such as mean and variance of word distances, and based on various hypothesis tests, such as the t test, χ^2 test, and likelihood ratios, or using pointwise mutual information. In essence, the different methods are frequency-based methods using fixed windows with re-ranking based on different statistics. Beside the domain-specific analysis of this problem, there are major technical differences with our approach. We are also interested in mining co-occurrences of potentially larger sets of items efficiently, while the focus by the Natural Language Processing community is more on defining interestingness measures on bi-grams (or smaller sets of items). We remark that, for mining smaller sets of items within a small fixed window, it is relatively straightforward to design an algorithm that generates all possible n-grams in a reasonable time. However, for larger itemsets and window sizes, any algorithm would need to at least prune the search space of possible candidates in some way, to overcome the combinatorial explosion induced by larger itemsets and window sizes.

Process Mining Applications

Another possible application of discovering episodes with a high cohesion is in business process mining. Traditionally process mining focuses on the discovery of an end-to-end process modelled for instance using a petri net (Van Der Aalst et al. 2007). Here, the input process log consists of several traces, or multiple sequences, each consisting of a tuple that includes a timestamp, an activity (or event) code and possibly other attributes, such as the person executing the activity. However, recently different authors have suggested adapting pattern mining for discovering local interesting patterns in process logs. For instance, Leemans and van der Aalst (2014) proposes to find frequent episodes in a business process using WINEPI (Mannila et al. 1997). Here, episodes are of interest since they can model both consecutive and concurrent activities (or events). Related, Tax et al. (2016) proposes to extend frequent pattern mining, but with richer type of patterns, namely process trees that offer additional flexibility, such as choice, between activities in the pattern. Both approaches rank patterns using frequency. An uncovered episode with high cohesion is indicative of a set of activities that mostly co-occur near each other in the process log. For instance, it could be that after a certain activity a occurs, it should always be verified using either activity b or c within a certain time frame for legal or financial reasons. Therefore, we can expect these activities to co-occur in the business process log and discover a highly cohesive dominant episode $G(\{a,b,c\},\{a \rightarrow b, a \rightarrow c\})$. Additionally, if a is not followed by b this can be used to flag a sequence of activities not conforming with this pattern. If we only mine frequent patterns, less frequent but highly cohesive dominant episodes would be missed. Therefore, we argue that discovering cohesive dominant episodes could be of interest in process mining especially to support tasks such as conformance checking.

Quantile-based Cohesion

In the next Chapter, we will propose an alternative interestingness measure for evaluating sequential patterns in a single long sequence (Feremans et al. 2018). We will evaluate what percentage of the pattern's minimal occurrences are small enough, where small enough is

defined by multiplying a user-defined parameter with the size of a pattern. For example, if this parameter is set to 2, a window of size 6 will be small enough for a pattern of size 3. This method does not take the size of the minimal windows of a pattern into account, as long as they are small enough. For example, if any window of size 10 or smaller is considered small enough, then a window of size 2 will score just as much as a window of size 9. This method is, however, robust to outliers and optimised for sequential patterns.

2.10 Conclusion

In this Chapter, we present a novel method for finding valuable patterns in event sequences. First of all, we evaluate the quality of the discovered itemsets using cohesion, a measure of how far apart the items making up the itemset are on average. In this way, we reward strong patterns that are not necessarily very frequent in the data, which allows us to discover patterns that existing frequency-based algorithms fail to find. Since cohesion is not an anti-monotonic measure, we rely on an alternative pruning technique, based on an upper bound of the cohesion of candidate patterns that have not been generated yet. We show both theoretically and empirically that the method is sound, the upper bound tight, and the algorithm efficient, allowing us to discover large numbers of patterns reasonably quickly.

Based on the discovered cohesive itemsets, we then search for representative sequential patterns and dominant episodes, which offer additional information about the order in which the items making up the itemsets occur. If no order is representative of the occurrences of an itemset, we report no sequential pattern or partial order. Furthermore, we mine association rules, with a confidence measure based on the cohesion of the antecedent and consequent, rather than the frequency-based definition common in literature. We integrate the mining process of all four pattern types into a single efficient algorithm.

Experimental results demonstrate that our approach produces a more intuitive ranking of patterns than existing frequency-based state-of-the-art methods. For all pattern types, we rank interesting patterns highly, while avoiding spurious patterns that consist of unrelated items that often co-occur purely because they all occur very frequently. For sequential patterns and, particularly, episodes, we limit the number of patterns that can be generated from a single itemset, thus avoiding a pattern explosion common for existing algorithms. Our experiments confirm both the high quality of our output and the efficiency of our algorithm.

"To raise new questions, new possibilities,
to regard old problems from a new angle,
requires creative imagination
and marks real advance in science."

- Albert Einstein

CHAPTER 3

Mining Quantile-based Cohesive Patterns in Sequences

Finding patterns in long event sequences is an important data mining task. Two decades ago, research focused on finding all frequent patterns, where the anti-monotonic property of frequency was used to design efficient algorithms. Recent research focuses on producing a smaller output containing only the most interesting patterns.

In this Chapter[1], we introduce a new interestingness measure by computing the proportion of the occurrences of a pattern that are cohesive. This measure is robust to outliers and is applicable to sequential patterns. We implement an efficient algorithm based on constrained prefix-projected pattern growth and pruning based on an upper bound to uncover the set of top-k quantile-based cohesive sequential patterns.

We perform experiments and compare our method with existing state-of-the-art methods for mining interesting sequential patterns. We show that our algorithm is efficient and produces qualitatively interesting patterns on large event sequences that other methods fail to find.

[1]This chapter is based on work published in the SIAM International Conference on Data Mining as "Mining Top-k Quantile-based Cohesive Sequential Patterns" by Len Feremans, Boris Cule and Bart Goethals (Feremans et al. 2018).

3.1 Introduction

Pattern discovery in sequential data is an established field in data mining. The earliest research focused on the setting where data consisted of *many* sequences, where a pattern was defined as a sequence that re-occurred in a high enough number of such input sequences. Among the algorithms that produce a ranking of the most frequent sequential patterns, given a large database of typically short sequences, are GSP (Srikant and Agrawal 1996) and PREFIXSPAN (Pei et al. 2004). For mining patterns in a *single* long sequence, the first method was proposed by Mannila et al. (1997). Their WINEPI method uses a sliding window of a fixed length to traverse the sequence, and a pattern is then considered frequent if it occurs in a high enough number of these sliding windows. Laxman et al. (2007) reformulate frequency as the maximal number of non-intersecting *minimal windows* of the pattern in the sequence. In this context, a minimal window of the pattern in the sequence is defined as a subsequence of the input sequence that contains the pattern, such that no smaller subsequence also contains the pattern. All of the above algorithms are able to generate all frequent patterns by leveraging the so-called APRIORI property (Agrawal and Srikant 1994). This property implies that the frequency of a pattern is never smaller than the frequency of any of its superpatterns. In other words, frequency is an *anti-monotonic* quality measure. While this property is computationally very practical since large candidate patterns can be generated from smaller patterns, the undesirable side-effect is that single items and small patterns, in general, will always be ranked higher than their superpatterns. Another argument against classical frequency-based techniques is that they report sets or sequences of items where items occur frequently together in a window, but do not account for all individual occurrences of these items. If two items occur frequently, and through pure randomness often occur near each other, they will together form a top-ranked pattern, even though they are not correlated.

Recent research, however, stepped away from mining *all* frequent patterns. Some authors reduce the number of patterns, for example, using information theoretic approaches such as *Minimal Description Length* (Grünwald 2007), thereby producing a smaller set of patterns that covers the sequence best (Hoang et al. 2014; Tatti and Vreeken 2012; Fowkes and Sutton 2016). Other authors propose different *measures of interestingness*, that do not benefit from an anti-monotonic quality measure to prune the search space of candidate patterns but produce a more interesting ranking of patterns (Cule et al. 2016; Petitjean et al. 2016; Tatti 2015).

As presented in the previous chapter, Cule et al. (2009) introduced a new interestingness measure called *cohesion*, defined as a measure of how near each other the items making up an interesting itemset occur on average. However, just like frequency-based methods, cohesion has its drawbacks. For example, suppose items a and b occur very frequently next to each other in the input sequence. Now suppose that one occurrence of b is very far from the nearest a. Since cohesion is inversely proportional to the mean of *all* minimal windows, itemset $\{a, b\}$ would score low on cohesion. Furthermore, cohesion is only defined for itemsets and is not a suitable measure for sequential patterns.

In this work, we tackle this problem by measuring the proportion of a pattern's occurrences that are cohesive, where we consider an occurrence to be cohesive if the minimal window length is small relative to the size of the pattern. We call this measure the *quantile-based cohesion* of the pattern. This is a more robust measure that is not susceptible to random outliers. While we concentrate on sequential patterns, the work presented here can directly be applied to other pattern types, such as itemsets, too. In the example above, itemset $\{a, b\}$ would, evaluated by our new measure, score very highly.

We illustrate the various interestingness measures in Figure 3.1. Here, we show a frag-

Receiving the top-maul from Starbuck, he advanced towards the main-mast with the hammer uplifted in one hand, exhibiting the gold with the other, and with a high raised voice exclaiming: Whosoever of ye raises me a white-headed whale with a wrinkled brow and a crooked jaw; whosoever of ye raises me that white-headed whale, with three holes punctured in his starboard fluke - look ye, whosoever of ye raises me that same white whale, he shall have this gold ounce, my boys!

"Huzza! huzza!" cried the seamen, as with swinging tarpaulins they hailed the act of nailing the gold to the mast.

It's a white whale," I say, resumed Ahab, as he threw down the top-maul; a white whale. "Skin your eyes for him, men; look sharp for white water; if ye see but a bubble, sing out."

All this while Tashtego, Daggoo, and Queequeg had looked on with even more intense interest and surprise than the rest, and at the mention of the wrinkled brow and crooked jaw they had started as if each was separately touched by some specific recollection.

"Captain Ahab," said Tashtego, "that white whale must be the same that some call Moby Dick."

"Moby Dick?" shouted Ahab. "Do ye know the white whale then, Tash?"

"Does he fan-tail a little curious, sir, before he goes down?" said the Gay-Header deliberately.

"And has he a curious spout, too," said Daggoo, "very bushy, even for a parmacetty, and mighty quick, Captain Ahab?"

"And he have one, two, tree - oh! good many iron in him hide, too, Captain," cried Queequeg disjointedly, "all twiske-tee betwisk, like him - him - "

Figure 3.1: Fragment of the novel Moby Dick written by Herman Melville. We highlight 4 sequential patterns having a high value of quantile-based cohesion

ment of the novel Moby Dick written by Herman Melville with four sequential patterns highlighted. These four patterns are all ranked in the top-10 using quantile-based cohesion. Frequency defined as the number of minimal non-overlapping windows, as proposed by Laxman et. al, would report $(white, whale)$, $(captain, ahab)$ and $(moby, dick)$ in the top-10, but not $(wrinkled, brow, crooked, jaw)$ because this pattern does not occur frequently enough. We also remark that other patterns, such as $(ahab, dick)$, are ranked highly using frequency alone, despite not being correlated. The cohesion-based FCI_{seq} algorithm from Chapter 2, does not rank $(captain, ahab)$ high, due to fact that the two items, though correlated, also occasionally appear far from each other, resulting in a relatively large mean of minimal window lengths.

Since quantile-based cohesion is not anti-monotonic, designing an efficient algorithm to exactly find all sequential patterns with high quantile-based cohesion is not trivial. To facilitate our search, we define an *upper bound* that computes the maximal quantile-based cohesion for any superpattern of the current candidate sequential pattern that could still be generated to *prune* candidate patterns. Our algorithm uses *constrained prefix-projected pattern growth* to generate all candidates and uses this upper bound for additional pruning. Computation of all minimal windows and generation of candidates becomes more efficient as the projected input sequence becomes smaller, ensuring that our algorithm is also efficient on larger event sequences with many items. We perform several experiments to validate that that our algorithm perform well on artificial and text datasets from a qualitative and performance perspective compared to state-of-the-art methods.

The remainder of this Chapter is organised as follows. In Section 3.2 we formally describe the problem setting and define the patterns we aim to discover. Section 3.3 provides a detailed

description of our algorithm and upper bound. In Section 3.4 we present an experimental evaluation of our method and compare with several related state-of-the-art methods. We present an overview of the most relevant related work in Section 3.5 and conclude our work in Section 3.6.

3.2 Problem Setting

Definition 3.1 (Event sequence)**.** *The input dataset consists of a* single *sequence of* items *(or* events*), that is* $\mathcal{S} = (\langle i_1, t_1\rangle, \ldots, \langle i_n, t_n\rangle)$*, where* $i_k \in \Omega$ *is an item coming from a finite domain* Ω *of all possible items, and* t_k *is a timestamp. The sequence is ordered chronologically so for any* $1 < k \leq n$*, it holds that* $t_{k-1} \leq t_k$.

We recall that this definition is similar to Definition 2.1, however, in this Chapter we have a different intake, such as allowing multiple items at a single timestamp and mining sequential patterns with possibly repeating items.

A *window* $\mathcal{S}[t_a, t_b]$ is a subsequence of $\mathcal{S}$ between timestamps t_a and t_b, that is, $\mathcal{S}[t_a, t_b]$ contains all $\langle i_k, t_k\rangle \in \mathcal{S}$ for which $t_a \leq t_k \leq t_b$. We define the window *length* as $|\mathcal{S}[t_a, t_b]| = t_b - t_a$. For simplicity we omit the timestamps from our *examples* and write a sequence as $(i_1, \ldots, i_n)$ thereby assuming the timestamps are consecutive integers.

Definition 3.2 (Sequential pattern)**.** *A sequential pattern is denoted as* $X_s = (s_1, \ldots, s_m)$*, representing a pattern that consists of items* s_1 *until* s_m *in that order, where* $s_k \in \Omega$*. A sequential pattern* X_s *occurs in a window* $\mathcal{S}[t_a, t_b]$*, denoted by* $X_s \prec \mathcal{S}[t_a, t_b]$*, if all items in* X_s *occur in the specified order in the window, that is,*

$$X_s = (s_1, \ldots, s_m) \prec \mathcal{S}[t_a, t_b] \Leftrightarrow$$

$$\exists t_1, \ldots, t_m \in [t_a, t_b] : t_1 < t_2 < \ldots < t_m : \forall j \in \{1, \ldots, m\} : \langle i_j, t_j\rangle \in \mathcal{S} \wedge s_j = i_j.$$

We do not require each item s_k *to be unique, that is, sequential patterns can contain repeating items.*

We remark that, unlike windows, a sequential pattern occurrence allows gaps between items. To evaluate a pattern, we make use of *minimal windows*.

Definition 3.3 (Minimal window)**.** *For every* distinct *item* $i \in X_s$ *and every timestamp* t *where* $\langle i, t\rangle \in \mathcal{S}$*, we define the minimal window at timestamp* t *as the shortest window around* t *that contains an occurrence of* X_s*, that is,*

$$W_t(X_s, \mathcal{S}) = \begin{cases} \infty & \text{if } \nexists \mathcal{S}[t_a, t_b] : t_a \leq t \leq t_b \wedge X_s \prec \mathcal{S}[t_a, t_b] \\ \min\limits_{\mathcal{S}[t_a, t_b]} \{|\mathcal{S}[t_a, t_b]| \mid t_a \leq t \leq t_b \wedge X_s \prec \mathcal{S}[t_a, t_b]\} & \text{otherwise.} \end{cases}$$

Note that a sequential pattern is sometimes not covered by any window at timestamp t. For example, given the sequential pattern $X_s = (a, b)$ and the sequence $\mathcal{S} = (\ldots, b, a)$, for the last a there is no window that would cover X_s. In this case, we say the minimal window has a length of ∞. Since we measure the proportion of occurrences that are cohesive, this is not a problem, as large minimal windows are discarded anyway.

Definition 3.4 (Support sequential pattern)**.** *We denote the set of occurrences of the pattern as* $cover(X_s, \mathcal{S}) = \{t \mid \langle i, t\rangle \in \mathcal{S} \wedge i \in X_s\}$*, and define its support as* $support(X_s, \mathcal{S}) = |cover(X_s, \mathcal{S})|$ *where we omit the* $\mathcal{S}$ *argument if it is clear from the context.*

We are now ready to define *quantile-based cohesion.* This measure tackles the problems of both frequency-based and cohesion-based methods, as illustrated in Section 3.1.

Definition 3.5 (Quantile-based cohesion)**.** *Given a cohesion threshold α, that determines, relative to the pattern size, if a pattern occurrence is cohesive enough, we compute the proportion of the occurrences that are cohesive. We define the quantile-based cohesion w.r.t α for a sequential pattern X_s in sequence $\mathcal{S}$ as*

$$C_{quan}(X_s) = \frac{|\{t|\ t \in cover(X_s) \wedge W_t(X_s) < \alpha \cdot |X_s|\}|}{support(X_s)},$$

where we omitted the $\mathcal{S}$ argument if it is clear from the context.

The parameter α can be any real number higher than 1.0. If α is set to 1.01 we only count occurrences of the sequential pattern where there are no gaps between items. If α is 2.01 we count occurrences where there are at most $|X_s|$ gaps between items. In our experiments, we set α to 2, 3, 4 or 5. Remark that for timestamped sequences the length (or duration) of a minimal window is represented by a unit of time, e.g., only occurrences within $\alpha \times |X_s|$ *minutes* are considered nearby.

Given a frequency threshold θ we ignore all *infrequent* items, that is, if $support(i) < \theta$, we do not use item $i \in \Omega$ in the generation of candidate patterns. Unlike frequency-based methods this does not limit longer patterns, in fact, our definition of support increases (monotonically) for longer patterns. The rationale behind this threshold is to filter items occurring only a couple of times in the sequence. For example, if we create a sequence of words in a novel we are less interested in patterns that only occur once or twice. Finally, our goal is to solve the following problem:

Problem 3.1 (Top-k quantile-based cohesive sequential patterns)**.** *Given a single sequence of items $\mathcal{S}$, a cohesion threshold α, a frequency threshold θ, a size limit max_size, and the number of desired patterns k, find each sequential pattern X_s where*

1. *$|X_s| \leq max_size$,*
2. *for each $i \in X_s, support(i) \geq \theta$,*
3. *X_s is ranked in the top-k set of patterns according to $C_{quan}(X_s)$ w.r.t α.*

We remark that while quantile-based cohesion is more robust to outliers than cohesion, it does not take the size of the minimal windows into account. In response to this observation, we propose a variant, namely *weighted quantile-based cohesion*, discussed in Appendix B.

3.3 Mining Quantile-based Cohesive Sequential Patterns

In this section, we present a detailed description of our algorithm for mining the top-k quantile-based cohesive sequential patterns. We first show how we generate candidates using prefix-projected pattern growth. We then discuss how we can prune large numbers of potential candidates by computing an upper bound on quantile-based cohesion. Parameters for controlling our algorithm include *k*, α and *max_size*.

3.3.1 Prefix-projected Pattern Growth

Our algorithm combines ideas from two different methods. At its core, our algorithm is similar to the depth-first search from Chapter 2 for mining cohesive itemsets. We first generate candidates in a depth-first way. For each candidate, we compute the set of minimal windows and prune a candidate and associated superpatterns based on an upper bound of quantile-based cohesion. There are two *bottlenecks* in this baseline algorithm. First, we would have to compute the set of minimal windows for each candidate, which requires visiting all occurrences of items in the current candidate X_s. Second a naive approach would generate new candidate patterns by combining the current candidate with all items in Ω. In order to address both bottlenecks, we integrate this approach with the strategy of *recursively projecting the input sequence*, similar to prefix-projected pattern growth first used in PREFIXSPAN (Pei et al. 2004). During the depth-first search we generate candidate *supersequences*.

Definition 3.6 (Supersequence)**.** *A candidate supersequence Z_s is generated by adding items from a set Y at the end of the current candidate sequential pattern X_s, that is, given $X_s = (s_1, \dots, s_n)$ and $Y = \{y_1, \dots, y_l\}$. We define the set of all possible supersequences $\mathcal{Z}(X_s, Y)$ as*

$$\mathcal{Z}(X_s, Y) = \{Z_s \mid Z_s = (s_1, \dots, s_n, z_{n+1}, \dots, z_m)$$
$$\wedge\ |Z_s| \leq max_size\ \wedge\ \forall i \in [n+1, m] : z_i \in Y\}.$$

Definition 3.7 (Projection)**.** *The set of candidate items Y that can be used to generate super sequences can be computed based on the projection of $\mathcal{S}$ on $X_s = (s_1, \dots, s_n)$. We define this projection, denoted by $\mathcal{P}_{X_s}(\mathcal{S})$, as:*

$$\mathcal{P}_{X_s}(\mathcal{S}) = \{\mathcal{S}[t_a, t_b] \mid \langle s_1, t_a \rangle \in \mathcal{S} \wedge X_s \prec \mathcal{S}[t_a, t_b]\},$$

with $t_b = t_a + \alpha \cdot max_size$. If $X_s = \emptyset$ we initialise Y to Ω.

Note that we restrict the length of the projected windows to $\alpha \cdot max_size$. We define the *suffix* of a window given a sequential pattern X_s as the subsequence after the first occurrence of X_s,

$$suffix(\mathcal{S}[t_a, t_b], X_s) = \mathcal{S}[t + \epsilon, t_b] \mid t_a \leq t \leq t_b \wedge X_s \prec \mathcal{S}[t_a, t] \wedge \nexists t' : t' < t \wedge X_s \prec \mathcal{S}[t_a, t'],$$

where we use ϵ to enforce that $\mathcal{S}[t_a, t]$ is non-overlapping with $\mathcal{S}[t + \epsilon, t_b]$. Here, $\epsilon = \min\{t_{k+1} - t_k \mid \langle i_k, t_k \rangle, \langle i_{k+1}, t_{k+1} \rangle \in \mathcal{S} \wedge t_k \neq t_{k+1}\}$, i.e., the smallest possible time period between two non-simultaneous events.

Definition 3.8 (Candidate items)**.** *Given the projection on X_s of $\mathcal{S}$ we can define the* multiset *of all possible candidate items Y^+ as*

$$Y^+ = \bigcup_{\mathcal{S}[t_a, t_b] \in \mathcal{P}_{X_s}} \left(\biguplus_{s_k \in suffix(\mathcal{S}[t_a, t_b], X_s)} \{s_k\} \right),$$

where we use a multiset-union $\uplus$, which allows us to bound the number of possible repetitions allowed for each item z_i in candidate supersequences Z_s. A similar definition based on the set-union is used to compute Y.

We now discuss the intuition behind these definitions. We first observe that we are only interested in computing the number of minimal windows that are smaller than $\alpha \cdot |X_s|$. Therefore, it is not necessary to compute all minimal windows. A second observation is that given

pattern $X_s = (s_1)$ we can guarantee that for any supersequence, each interesting minimal window will start with an occurrence of s_1 and must be smaller than $\alpha \cdot max_size$. Therefore, we can compute the set of minimal windows based on the projection of (s_1) induced on $\mathcal{S}$ for every supersequence that starts with (s_1). On the first level, the set of projected windows might not be much smaller than $|\mathcal{S}|$, but as the pattern becomes longer, the projection will become much smaller. For instance, given $Z_s = (s_1, s_2)$ each window $\mathcal{S}[t_a, t_b] \in \mathcal{P}_{(s_1)}$ can be removed from $\mathcal{P}_{(s_1,s_2)}$ if $(s_1, s_2) \not\prec \mathcal{S}[t_a, t_b]$. Furthermore, each individual window in the projection will shrink based on *suffix*$(\mathcal{S}[t_a, t_b], X_s)$. Thus by computing the projected windows, two bottlenecks of the depth-first search are resolved. First, minimal windows for a candidate $Z_s = (s_1, \ldots, s_{k+1})$ can be computed *incrementally* on the *monotonically decreasing* projection induced by $X_s = (s_1, \ldots, s_k)$. Second, candidate items s_{k+1} for generating candidates at each next level, are not selected from the full set Ω but must occur in Y^+. Finally, an additional benefit of applying prefix-projected pattern growth is that our upper bound becomes tighter, as discussed in Section 3.3.3.

Algorithm 3.1: QCSP($\mathcal{S}, k, \alpha, max_size$) Mine top-$k$ quantile-based cohesive sequential patterns in a single sequence

Input: An event sequence $\mathcal{S}$, number of patterns k, cohesion threshold α, pattern size limit max_size

Result: (At most) k sequential patterns ranked according to C_{quan} w.r.t. α

```
1  stack ← [⟨∅, S, Ω⟩];
2  heap ← MAKE_HEAP(k);
3  min_coh ← 0.0;
4  while stack ≠ ∅ do
5      ⟨X_s, P_{X_s}, Y⟩ ← stack.POP();
6      if Y = ∅ then
7          if |X_s| > 1 ∧ C_quan(X_s) > min_coh then
8              heap.PUSH(⟨X_s, C_quan(X_s)⟩);
9              if heap.SIZE() > k then
10                 heap.POP();
11                 min_coh ← heap.MIN()
12     else
13         if X_s ∩ Y = ∅ ∧ mingap(X_s) + |Z_max| > α · (|X_s| + |Z_max|) then continue ;
14         if C_maxquan(X_s, Y) ≤ min_coh then
               continue ;
15         s_{k+1} ← FIRST(Y);
16         stack.PUSH(⟨X_s, P_{X_s}, Y \ {s_{k+1}}⟩);
17         if |X_s| = max_size then
               continue ;
18         Z_s ← (X_s, s_{k+1});
19         P_{Z_s} ← PROJECT(S, Z_s, P_{X_s}, α, max_size);
20         Y_{Z_s} ← PROJ_CANDIDATES(S, Z_s, P_{Z_s});
21         stack.PUSH(⟨Z_s, P_{Z_s}, Y_{Z_s}⟩);
22 return heap;
```

Algorithm

The main algorithm for mining quantile-based cohesive sequential patterns (QCSP) is shown in Algorithm 3.1. We maintain a stack for performing the recursive prefix-projected search. We maintain three variables during recursion: the current candidate sequential pattern X_s, the projection on X_s of the sequence and a set of candidate items for generating supersequences Z_s where X_s is a prefix of Z_s. We initialise this stack by setting X_s to the empty sequence, the projection is $\mathcal{S}$ itself, and the initial set of candidate items Ω (line 1). The initial set of candidate items Ω is constructed by filtering frequent items w.r.t. θ. Moreover, we sort frequent items on support in ascending order as patterns consisting of less frequent items are faster to evaluate. Next, we initialise an empty heap that contains at most k patterns sorted on quantile-based cohesion (line 2). This top-k of most quantile-based cohesive patterns is also returned after the main prefix-projected search loop has finished (lines 4-21). In the main loop, we first pop the current node from the stack (line 5). We investigate if this is a leaf, that is, an unpruned sequential pattern, with no more supersequences to enumerate. We add the candidate to the heap of top-k most cohesive patterns if its quantile-based cohesion is higher than the current worst candidate pattern in the heap (line 8). Note that the first k candidates are added without any condition, but the minimal quantile-based cohesion will increase as more and more candidates are discovered, which in turn affects the pruning. If the candidate is not a leaf, we evaluate the *mingap* and *upper bound* function $C_{maxquan}(X_s, Y)$, which are explained in the next subsection (line 13-14). If the current candidate and all its supersequences cannot be pruned, we generate a supersequence $Z_s = (X_s, s_{k+1})$ using the first item in Y (line 18), the set of possible items to generate candidates of length $|X_s| + 1$. We compute the projection of Z_s induced on $\mathcal{S}$ by calling the function PROJECT (line 19). After the projection is computed we can traverse it to enumerate all possible items to form supersequences, which is the main goal of the function PROJ_CANDIDATES (line 20).

3.3.2 Incremental Computation of Prefix-projections

PROJECT computes the projection $\mathcal{P}_{Z_s}(\mathcal{S})$ incrementally based on the projection of $\mathcal{P}_{X_s}(\mathcal{S})$. The pseudocode for computing the projection of $Z_s = (s_1, \ldots, s_k, s_{k+1})$ incrementally is shown in Algorithm 3.3. If $k = 0$, i.e., if $X_s = \emptyset$, we create a window of size $\alpha \cdot max_size$ at every occurrence of $\langle s_{k+1}, t\rangle \in \mathcal{S}$. If $k > 0$, we check for each window in the previous projection if Z_s occurs, and, if it occurs, we take the suffix. We also maintain the original timestamp t at which each window starts. Note that the suffix starts with the first event after timestamp t, whereby we use ϵ to denote the smallest possible time period between two non-simultaneous events.

Algorithm 3.2: PROJ_CANDIDATES($\mathcal{S}, X_s, \mathcal{P}_{X_s}$) Compute candidate itemset Y based on projection

Input: An event sequence $\mathcal{S}$, $X_s = (s_1, \ldots, s_k)$, projection P_{X_s}
Result: Set of possible items $s_{k+1} \in Y$ to generate super sequences $Z_s = (s_1, \ldots, s_k, s_{k+1})$

```
1 Y ← ∅;
2 for ⟨t, S[t_a, t_b]⟩ in P_Xs do
3 |   Y ← Y ∪ S[t_a, t_b];
4 sort Y on descending support in P_Xs;
5 return Y;
```

Algorithm 3.3: PROJECT($\mathcal{S}, Z_s, \mathcal{P}_{X_s}, \alpha, max_size$) Computes pseudo-projection incrementally

```
Input: An event sequence S, super sequence Z_s = (s_1,...,s_k,s_{k+1}), projection P_{X_s} where
       X_s = (s_1,...,s_k), cohesion threshold α, pattern size limit max_size
Result: Projection P_{Z_S}
/* Level 1 (|X_s| = 0): projection of maxwin size */
1  P' ← ∅;
2  if |X_s| = 0 then
3  |  maxwin ← ⌊α · max_size⌋;
4  |  for t ← 1 to |S| do
5  |  |  if S[t] = s_{k+1} then
6  |  |  |  t_a ← t + ε;
7  |  |  |  t_b ← t + maxwin;
8  |  |  |  P' ← P' ∪ ⟨t, S[t_a, t_b]⟩;
/* Level >1: take suffix or remove projection window */
9  else
10 |  for ⟨t, S[t_a, t_b]⟩ in P_{X_s} do
11 |  |  if s_{k+1} ∈ S[t_a, t_b] then
12 |  |  |  t_a' ← first occurrence of s_{k+1} in S[t_a, t_b];
13 |  |  |  P' ← P' ∪ ⟨t, S[t_a' + ε, t_b]⟩;
14 return P';
```

Compute Candidate Items based on Projection

PROJ_CANDIDATES computes the set of possible candidate items Y. The algorithm is shown in Algorithm 3.2. We compute candidate items for forming a supersequence of length $k+1$ based on the projection of its prefix X_s. The procedure consists of taking the union of all items found in the suffix of each projected subsequence. To compute Y^+ for $C_{maxquan}$ a variant of this procedure is needed. This is omitted from the pseudocode but is trivial to compute.

Example 3.1. We use an example, shown in Figure 3.2, to illustrate the runtime behavior of our algorithm. We show the trace on the example sequence $(a,b,c,_,_,_,_,b,a,c)$ with parameters $\alpha = 2$, $max_size = 3$ and $k = 2$. In theory, there are $|\Omega|^2 + |\Omega|^3$ candidates possible. The first candidate generated is $X_s = (a)$. The projected window size is at most $\alpha \cdot max_size = 6$, and there are two windows in $\mathcal{P}_{(a)}$. After the projection, we find that only b and c are left in Y, the set of remaining items to form candidates that start with a. As c occurs twice and b only once, c is visited first, and our next candidate is $X_s = (a,c)$. Since the suffix of $\mathcal{P}_{(a,c)}$ does not contain any more items, it becomes a leaf, and we add sequential pattern (a,c) to the heap with a quantile-based cohesion of 1. Next we project on $X_s = (a,b)$ and remove one window in the projection. The only possible suffix left is c. We then generate $X_s = (a,b,c)$. This candidate pattern is a leaf node, and is added to the heap with a quantile-based cohesion of 0.5. Next, $X_s = (a,b)$ is visited again as a leaf, but not added to the heap, because its quantile-based cohesion of 0.5 is not strictly higher than the minimal score of $C_{quan}((a,b,c)) = 0.5$ of the top 2 patterns currently in the heap. Next, $X_s = (b)$ is generated, and then $X_s = (b,c)$. The quantile-based cohesion of $X_s = (b,c)$ is 1 and it is added to the heap, replacing the previously lowest-scoring element (a,b,c). Then, $X_s = (b,a)$ is generated. The maximal cohesion of (b,a), with $Y = \{c\}$ given by $C_{maxquan}((b,a),\{c\}) = \frac{4}{6}$ is less than the the current minimal score of 1, so we can prune this branch. Finally, candidates $X_s = (c)$ and $X_s = (c,b)$ are generated. $X_s = (c,b)$ is pruned since

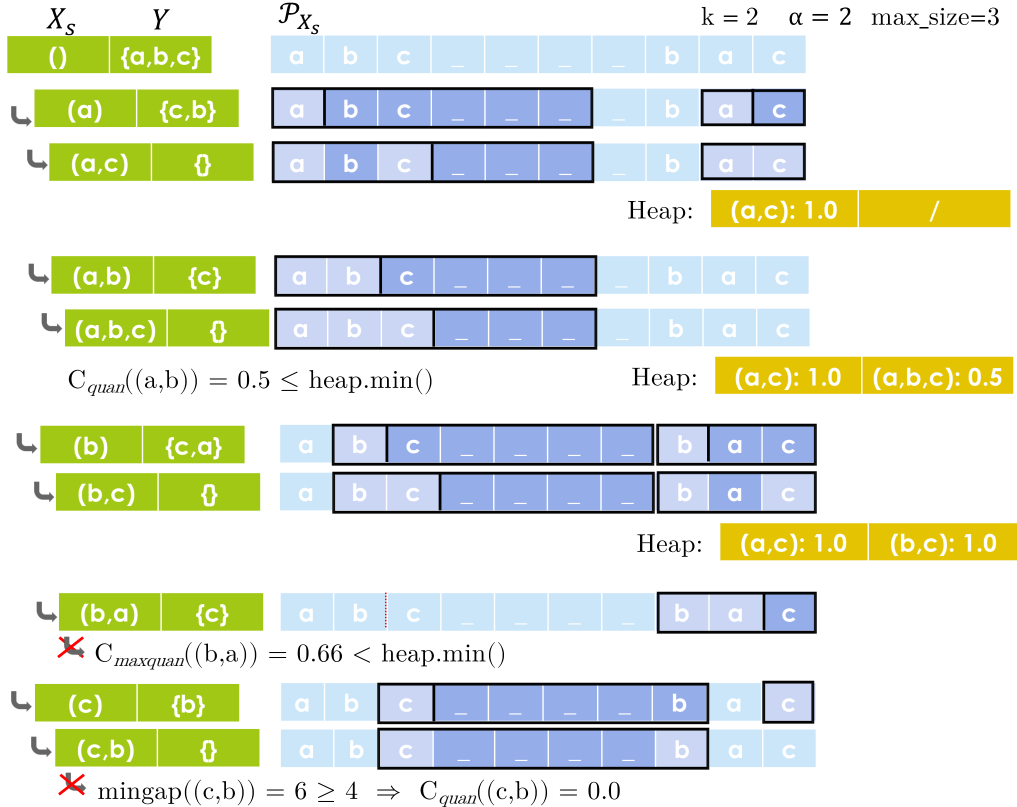


Figure 3.2: Illustrative example of QCSP, a prefix-projected pattern growth algorithm that employs pruning using mingap and an upper bound on quantile-based cohesion

the minimal gap is 6, which is higher than $\alpha \times 2$, hence $C_{quan}((c,b)) = 0$. The final top-k heap contains (b,c) and (a,c) both with a cohesion of 1. □

3.3.3 Pruning

At any node in the search tree, let X_s denote the current candidate sequential pattern, while Y denotes all items that can still be added to X_s to form supersequences. If we compute an upper bound on the quantile-based cohesion for all candidates $Z_s \in \mathcal{Z}(X_s, Y)$, and this maximal score is lower than *min_coh* we can prune the branch. Here, *min_coh* corresponds to the current minimum quantile-based cohesion of any pattern in the heap (or 0 if the heap is not full). In this Section, we derive this upper bound.

Limit Quantile-based Cohesion using Mingap

Our first *upper bound* comes from the observation that in some cases not a single occurrence of two items is cohesive. Intuitively, if the *minimal gap*, that is the minimal window length of any occurrence of (a,b) is already too high, the likelihood that (a,b) or any superpattern is cohesive is small.

Definition 3.9 (Minimal gap)**.** *We define the minimal gap as*

$$mingap(X_s) = \min_{t \in cover(X_s)} W_t(X_s).$$

Theorem 3.1 (Limit quantile-based cohesion using mingap). *Let X_s be a candidate pattern and Y the set of items that can still be added to X_s. Then, for each $Z_s \in \mathcal{Z}(X_s, Y)$ generated as candidate by Algorithm 3.1,*

$$C_{quan}(Z_s) = C_{quan}(X_s) = 0$$
$$\text{if } mingap(X_s) + |Z_{max}| > \alpha \cdot (|X_s| + |Z_{max}|) \text{ and } X_s \cap Y = \emptyset,$$
$$\text{where } |Z_{max}| = \max_{\mathcal{S}[t_a,t_b]\in\mathcal{P}_{X_s}} |suffix(\mathcal{S}[t_a,t_b], X_s)|.$$

Proof. We know that for any window of any supersequence $Z_s \in \mathcal{Z}(X_s, Y)$, where $X_s \cap Y = \emptyset$ it holds that

$$\forall\ t \in cover(Z_s) : W_t(Z_s) + (|Z_s| - |X_s|) \geq W_t(X_s).$$

Furthermore, we know by definition, that

$$\forall\ t \in cover(X_s) : W_t(X_s) \geq mingap(X_s).$$

Formally we want to prove that, $\forall Z_s \in \mathcal{Z}(X_s, Y)$,

$$C_{quan}(Z_s) = 0$$
$$\iff \frac{|\{t|\ t \in cover(Z_s) \land W_t(Z_s) < \alpha \cdot |Z_s|\}|}{support(Z_s)} = 0$$
$$\iff \nexists t \in cover(Z_s) : W_t(Z_s) < \alpha \cdot |Z_s|.$$

From the previous equations, it follows that the smallest minimal window of Z_s is bounded by

$$\min_{t\in cover(Z_s)} W_t(Z_s) \geq mingap(X_s) + (|Z_s| - |X_s|),$$

and we can deduce that

$$C_{quan}(Z_s) = 0$$
$$\iff mingap(X_s) + (|Z_s| - |X_s|) \geq \alpha \cdot |Z_s|.$$

What remains to be proven is that no pattern $Z_s \in \mathcal{Z}(X_s, Y)$ generated as candidate by Algorithm 3.1 can be longer than $|X_s| + |Z_{max}|$. In each recursive call, the algorithm adds *precisely* one item to the candidate sequence and removes *at least* one item from each window in the projection. Therefore, X_s can *at most* grow by the number of items in the largest suffix, and it directly follows that, for each generated candidate Z_s,

$$|Z_s| \leq |X_s| + |Z_{max}|.$$

Finally, we can derive that we can prune a candidate pattern X_s and any supersequence Z_s if

$$mingap(X_s) + |Z_{max}| \geq \alpha \cdot (|X_s| + |Z_{max}|).$$

This concludes the proof. □

Note that the restriction $X_s \cap Y = \emptyset$ remains necessary because otherwise, the previous inequality between minimal window length $W_t(X_s)$ and its *extension* $W_t(Z_s)$ does not hold. This *mingap* bound is thus only applicable to prune candidates when no repeating elements are in any suffix of the current projection, that is $X_s \cap Y = \emptyset$.

Example 3.2. We illustrate this upper bound with an example. Assume $X_s = (a, b)$ and we have the following set of projected windows:

$$\mathcal{P}_{(a,b)} = \{(\mathbf{a}, \mathbf{b}, d, d, c), (\mathbf{a}, _, \mathbf{b}, d, c, c), (\mathbf{a}, _, _, \mathbf{b}, c, e)\}.$$

The union of the remaining items in the suffixes is $Y = \{c, d\} \cup \{c, d\} \cup \{c, e\} = \{c, d, e\}$. However, we want to count all occurrences of the repeating cs and ds, because $\mathcal{Z}(X_s, Y)$ can contain patterns such as (a, b, c, c) or (a, b, d, d). We therefore take the multiset-union and compute $Y^+ = \{c, c, d, d, e\}$. Using this definition the longest possible pattern would be bound by $|Y^+| = 5$. However, we can further lower this upper bound, by considering each window separately. Since the longest suffix in our example has size 3, we will never be able to add more than 3 items to X_s, which is why we define $|Z_{max}|$ as

$$|Z_{max}| = \max_{\mathcal{S}[t_a, t_b] \in \mathcal{P}_{X_s}} |suffix(\mathcal{S}[t_a, t_b], X_s)|.$$

□

Example 3.3. As a second example, assume $X_s = (a, b)$ and we have the following set of projected windows:

$$\mathcal{P}_{(a,b)} = \{(\mathbf{a}, _, _, _, \mathbf{b}, d, f, _), (\mathbf{a}, _, _, _, \mathbf{b}, c, d, _), (\mathbf{a}, _, _, _, \mathbf{b}, _, c, e)\}.$$

The set of remaining items is $Y = \{c, d, e, f\}$. Note that $X_s \cap Y = \emptyset$. $\mathcal{Z}(X_s, Y)$ can contain superpatterns such as (a, b, f), (a, b, d, f) or (a, b, c, c). Here, $mingap(X_s)$ is 5 and $|Z_{max}|$ is 2 (ignoring gaps). The longest possible pattern has length $|X_s| + |Z_{max}| = 2 + 2$. If we assume $\alpha = 1$ than we can prune since $5 + 2 > 1 \cdot (2 + 2)$. □

Upper Bound on Quantile-based Cohesion

We now present a bound on the number of remaining cohesive minimal windows of any candidate supersequence $Z_s \in \mathcal{Z}(X_s, Y)$, even if repeating items are possible (that is $X_s \cap Y \neq \emptyset$). We can prune X_s, and all supersequences, if the maximal value for quantile-based cohesion is lower than the current value of *min_coh*.

Theorem 3.2 (Upper bound on quantile-based cohesion)**.** *Let X_s be a candidate pattern and Y the set of items that can still be added to X_s. Then for each $Z_s \in \mathcal{Z}(X_s, Y)$ generated as candidate by Algorithm 3.1,*

$$\begin{aligned}
C_{quan}(Z_s) &\leq C_{maxquan}(X_s, Y), \\
&\quad where \\
C_{maxquan}(X_s, Y) &= 1.0 - \frac{|\{t|\ t \in \beta \wedge W_t(X_s) \geq \alpha \cdot |Z'_{max}|\}|}{support(Z'_{max})}, \\
|Z'_{max}| &= min(max_size, |X_s| + |Y^+|), \\
support(Z'_{max}) &= \sum_{i\ \in\ X_s \cup Y} support(i), \\
&\quad \beta = \{t \mid \langle i, t \rangle \in \mathcal{S} \wedge\ i \in X_s \wedge \nexists \langle j, t \rangle \in \mathcal{S} : j \in Y\}.
\end{aligned}$$

Proof. For any pattern Z_s we defined support using

$$support(Z_s, \mathcal{S}) = |\{t \mid \langle i, t \rangle \in \mathcal{S} \wedge i \in Z_s\}|.$$

We can partition all items in Z_s in two disjoint sets. Let $Z_s = (X_s, Y_s)$, then the items in Z_s are either in $X_s \setminus Y_s$, or in $Y_s = (Y_s \setminus X_s) \cup (X_s \cap Y_s)$. We can rewrite the previous equation as:

$$support(Z_s, \mathcal{S}) = |\{t \mid \langle i,t\rangle \in \mathcal{S} \wedge i \in X_s \setminus Y_s\} \cup \{t \mid \langle i,t\rangle \in \mathcal{S} \wedge i \in Y_s\}|.$$

This is important: as we allow for repetitions, the set $X_s \cap Y_s$ might not be empty. For example, given $X_s = (a,b)$ and $Z_s = (a,b,a)$ an occurrence of item $\langle a,t\rangle$ might have a minimal window as the first item in Z_s or as the third item in Z_s. This complicates matters since we cannot state that the minimal window at t of X_s is smaller or equal than that of Z_s. Another issue is caused since we allow multiple items to occur at the *same timestamp* t. We want to exclude timestamps of items in $X_s \setminus Y_s$ also in Y_s and we refine the previous equation to define the following *disjoint* partition of timepoints:

$$\begin{aligned}
support(Z_s, \mathcal{S}) &= |\{t \mid t \in \beta\} \cup \{t \mid t \in \gamma\}| \\
&= |\{t \mid t \in \beta\}| + |\{t \mid t \in \gamma\}|, \ where \\
\beta_{Z_s} &= \{t \mid \langle i,t\rangle \in \mathcal{S} \wedge i \in X_s \wedge \nexists \langle j,t\rangle \in \mathcal{S} : j \in Y_s\}, \\
\gamma_{Z_s} &= \{t \mid \langle i,t\rangle \in \mathcal{S} \wedge i \in Y_s\}.
\end{aligned}$$

We now use this partition to bound the maximal number of minimal windows for any supersequence. For any $Z_s \in \mathcal{Z}(X_s, Y)$ it holds that

$$\forall\ t \in \beta_{Z_s} : W_t(Z_s) \geq W_t(X_s). \tag{1}$$

In other words, the minimal window length of any superpattern Z_s is at least as high as the minimal window length of X_s at a timestamp t where an item $i \in X_s$ occurs, but no items from Y_s occur.

Given the current value of *min_coh* (the quantile-based cohesion of the kth pattern in the current top-k, or 0 if we have found fewer than k patterns), we only want to enumerate candidates Z_s where $C_{quan}(Z_s)$ could turn out to be higher than or equal to *min_coh*. We now derive:

$$\begin{aligned}
C_{quan}(Z_s) &= \frac{|\{t | t \in cover(Z_s) \wedge W_t(Z_s) < \alpha \cdot |Z_s|\}|}{support(Z_s)} \\
&= support(Z_s)^{-1} \cdot (\ |\{t \mid t \in \beta_{Z_s} \wedge W_t(Z_s) < \alpha \cdot |Z_s|\}| + \\
&\qquad |\{t \mid t \in \gamma_{Z_s} \wedge W_t(Z_s) < \alpha \cdot |Z_s|\}|\ ) \\
&= 1.0 - support(Z_s)^{-1} \cdot (\ |\{t \mid t \in \beta_{Z_s} \wedge W_t(Z_s) \geq \alpha \cdot |Z_s|\}| + \\
&\qquad |\{t \mid t \in \gamma_{Z_s} \wedge W_t(Z_s) \geq \alpha \cdot |Z_s|\}|\ ) \\
&\leq 1.0 - support(Z_s)^{-1} \cdot (\ |\{t|\ t \in \beta_{Z_s} \wedge W_t(Z_s) \geq \alpha \cdot |Z_s|\}|).
\end{aligned} \tag{2}$$

We finalise the proof by producing bounds for the main three elements of the above equation. For every candidate Z_s generated by Algorithm 3.1, we can bound the size of Z_s, the size of β_{Z_s}, and the support of Z_s. First of all, given that Algorithm 3.1 produces candidates by adding items in Y^+ to X_s, until there are either no more items left to add or we have reached the size threshold *max_size*, it directly follows that, for every candidate Z_s generated by Algorithm 3.1,

$$|Z_s| \leq min(max_size, |X_s| + |Y^+|) = |Z'_{max}|. \tag{3}$$

Second, note that $\beta \subseteq \beta_{Z_s}$, and, therefore

$$|\{t|\ t \in \beta_{Z_s} \wedge W_t(Z_s) \geq \alpha \cdot |Z_s|\}| \ \geq\ |\{t|\ t \in \beta \wedge W_t(Z_s) \geq \alpha \cdot |Z_s|\}|. \tag{4}$$

Since $\beta \subseteq \beta_{Z_s}$, from Equation 1 it follows that

$$\forall\ t \in \beta : W_t(Z_s) \geq W_t(X_s),$$

and, therefore,

$$|\{t|\ t \in \beta \wedge W_t(Z_s) \geq \alpha \cdot |Z_s|\}| \ \geq\ |\{t|\ t \in \beta \wedge W_t(X_s) \geq \alpha \cdot |Z_s|\}|. \quad (5)$$

From Equation 3, it follows that

$$|\{t|\ t \in \beta \wedge W_t(X_s) \geq \alpha \cdot |Z_s|\}| \ \geq\ |\{t|\ t \in \beta \wedge W_t(X_s) \geq \alpha \cdot |Z'_{max}|\}|, \quad (6)$$

and, by combining Equations 4, 5 and 6, we obtain

$$|\{t|\ t \in \beta_{Z_s} \wedge W_t(Z_s) \geq \alpha \cdot |Z_s|\}| \ \geq\ |\{t|\ t \in \beta \wedge W_t(X_s) \geq \alpha \cdot |Z'_{max}|\}|. \quad (7)$$

Finally, since any candidate pattern Z_s can only contain items from X_s and Y, it follows that

$$support(Z_s) \leq \sum_{i \in X_s \cup Y} support(i) = support(Z'_{max}). \quad (8)$$

From Equations 2, 7 and 8, it now directly follows that

$$\begin{aligned}
&C_{quan}(Z_s)\\
&\leq 1.0 - support(Z_s)^{-1} \cdot (\ |\{t|\ t \in \beta \wedge W_t(X_s) \geq \alpha \cdot |Z'_{max}|\}|)\\
&\leq 1.0 - support(Z'_{max})^{-1} \cdot (\ |\{t|\ t \in \beta \wedge W_t(X_s) \geq \alpha \cdot |Z'_{max}|\}|)\\
&= C_{maxquan}(X_s, Y).
\end{aligned}$$

This concludes the proof. □

Example 3.4. We illustrate this bound using an example. Suppose $X_s = (a, b)$ occurs 10 times and $Y = (c)$ occurs 2 times. The minimal window lengths of X_s are $2,2,2,30,30,30,30,30,\infty$ and ∞. Let us further assume that $min_coh = 0.5$ and $\alpha = 2$. The maximal window length possible is $\alpha \cdot 3 = 6$, and there are seven windows of X_s larger than 6. There is only one (non-repeating) item $\{c\}$ left in Y^+ and only $Z_s = Z'_{max}(X) = (a, b, c)$ is possible. $support((a, b, c)) = 12$, thus $C_{maxquan}(X_s, Y) = 1 - \frac{7}{12} = \frac{5}{12}$ which is lower than min_coh so we can prune (a, b, c) (and any supersequences). We remark that, unlike pruning based on mingap, when X_s and Y are not disjoint, we can still count windows for non-overlapping items in X_s, for example given $X_s = (a, b, c)$ and $Y = \{c, d\}$, we can compute all windows of (a, b, c) for all occurrences of items in $X_s \setminus Y = \{a, b\}$. □

3.4 Experiments

In our experiments, we use one synthetic dataset and three text datasets for easy interpretation of patterns. We compare QCSP with three state-of-the-art methods in terms of performance and the quality of output. FCI$_{seq}$ from Chapter 2 finds all cohesive *itemsets*, SKOPUS (Petitjean et al. 2016) finds the top-k sequential patterns with the highest leverage, and GOKRIMP (Hoang et al. 2014) finds a set of patterns that best compresses the input. For all three methods, we use publicly available implementations developed in Java (Feremans 2019; Fournier-Viger et al. 2016). The implementation of QCSP in Java and used datasets are publicly available[2]. Since we compare our method with state-of-the-art algorithms for both the *single* sequence and the *multiple* sequences setting, we use two versions of each dataset, one for each setting.

[2]https://bitbucket.org/len_feremans/qcsp_public

3.4.1 Datasets

The *Synthetic* dataset is created by generating a single sequence of 2000 items randomly selected between 6 and 50. Next, we insert the sequential pattern $(1,2,3,4,5)$ with at most 5 gaps at 40 random non-overlapping locations. We transform this sequence $\mathcal{S}$ into a set of sequences $\mathcal{S}'$ by using a sliding window of size 20. The *Moby* dataset consists of all words in the novel Moby Dick written by Herman Melville (Melville 1851). We preprocessed the text using the Porter Stemmer, and removed the stopwords. We transformed the single sequence $\mathcal{S}$ into a set of sequences $\mathcal{S}'$ by creating a separate sequence for each sentence. *JMLR* consists of abstracts of papers from the Journal of Machine Learning Research, where each abstract is preprocessed as in *Moby*. Each abstract is considered a separate sequence. Since our method requires a single sequence, we transform this dataset by concatenating the abstracts, adding $\alpha \cdot max_size$ timestamps between any two abstracts, thus avoiding generating patterns that span over two different abstracts. Finally, *Trump* consists of tweets of president Trump from 1 January 2016 until 2 October 2017 (Trump 2017). We removed all re-tweets and preprocessed the tweet texts as in *Moby*, and converted into a single sequence as in *JMLR*. Table 3.1 shows the basic characteristics of each dataset, where $|\Omega|$ denotes the number of distinct items in the dataset and $\mu(\mathcal{S}')$ the average length of a sequence in the multiple sequences setting.

Table 3.1: Characteristics of datasets for comparing the performance of QCSP versus state-of-the-art methods

Dataset	$\lvert\mathcal{S}\rvert$	$\lvert\Omega\rvert$	$\lvert\mathcal{S}'\rvert$	$\mu(\mathcal{S}')$
Synthetic	2000	50	2000	20.0
Moby	113264	2059	10066	11.2
JMLR	75515	3846	787	96.0
Trump	57518	1069	5670	17.9

3.4.2 Performance Comparison

Runtime for QCSP and state-of-the-art methods

It is troublesome to compare the runtime of all algorithms directly for several reasons. First of all, the runtime depends on the chosen parameters and input representation (single sequence or many sequences), which are different for each method. Second, FCI_{seq} solves a different problem since it mines itemsets. And while QCSP allows sequential patterns to contain repeating items, this is not the case for SKOPUS or GOKRIMP. However, despite these restrictions, we include this experiment to get an idea of the overall runtime required for each method. For QCSP and SKOPUS, we set $max_size = 5$ and $k = 50$. For FCI_{seq}, we set $max_size = 5$, and for QCSP, we set α to 2. For all experiments, we use $\theta = 10$, and remove infrequent items during preprocessing. Since we cannot directly control the number of patterns in FCI_{seq}, we report the runtime for the run with the highest cohesion threshold that leads to discovering at least k patterns. Finally, GOKRIMP has no parameters to tune.

Table 3.2 shows the runtimes on all datasets. We note that FCI_{seq} is notably slower than other methods, while GOKRIMP is notably faster. However, GOKRIMP is different in nature, since it does not attempt to generate all candidates and performs *greedy search* using heuristics. Furthermore, GOKRIMP always produced fewer than 50 patterns. Overall, we can conclude that the runtime of QCSP is both acceptable and competitive.

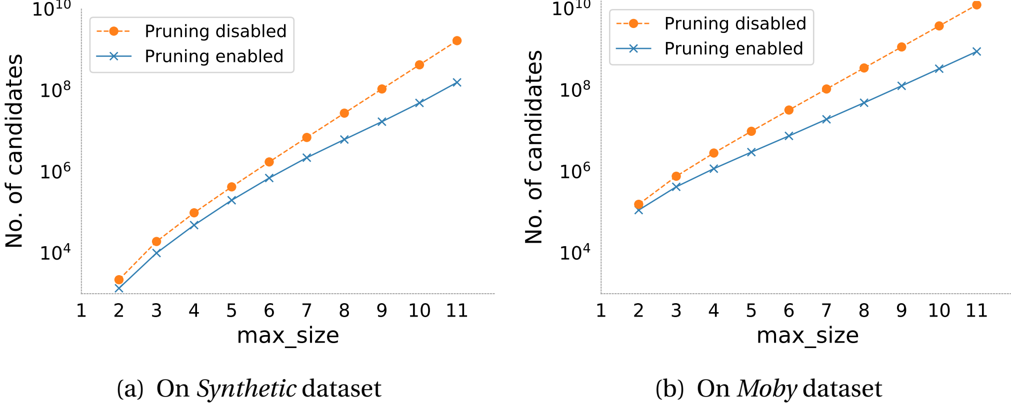
Table 3.2: Runtimes for QCSP and state-of-the-art methods on all datasets

Dataset	FCI$_{seq}$	SKOPUS	GOKRIMP	QCSP
Synthetic	44.2s	19.8s	1.0s	1.7s
Moby	126.0s	47.8s	2.0s	18.4s
JMLR	255.4s	40.6s	2.0s	25.9s
Trump	196.9s	2.5s	5.0s	8.1s

Effect of Pruning

We now analyse the impact on performance of pruning based on mingap and based on an upper bound on quantile-based cohesion. To evaluate the performance of our pruning technique, we run our algorithm on the *Synthetic* and *Moby* datasets, while varying *max_size*, and mine the top-k sequential patterns using prefix-projected pattern growth, with and without pruning. In this experiment, α is set to 2.0 and k is set to 20.

In Figure 3.3 we show the number of candidates (in log scale) on the *Synthetic* and *Moby* datasets, with and without pruning. We vary *max_size* between 2 and 11. From these results, we conclude that pruning has a significant impact on the number of candidates, which are reduced by an order of magnitude for patterns of larger sizes, thereby also reducing memory consumption. Concerning performance, the number of candidates (and runtime) grows almost *exponentially* with the maximal pattern length, which follows from the fact that the number of possible candidate patterns also increases exponentially with $\mathcal{O}(\Omega^{max_size})$. We do not include runtime plots, but for the *Synthetic* dataset, the increase in runtime is marginal, taking 2026 seconds without pruning and 1880 seconds with pruning for $max_size = 11$. For the *Moby* dataset, the increase in runtime is high when *max_size* is high, taking 196 minutes without pruning and 113 minutes with pruning for $max_size = 11$. The moderate increase in runtime on the *Synthetic* dataset, and lower values of *max_size* on the *Moby* dataset, is due to the fact that during each iteration we must compute the pruning functions, thereby computing the number of minimal windows of X_s based on $\mathcal{P}_{X_s}$, which generates overhead. Overall, we conclude that pruning on a larger search space seems to have a large effect on runtime, while for smaller search space, the effect is marginal. We remark that in future work it would be interesting to investigate approximations based on the current bounds, that potentially prune slightly fewer



(a) On *Synthetic* dataset (b) On *Moby* dataset

Figure 3.3: Number of candidates (log scale) visited by QCSP with and without pruning

candidates, but are faster to compute in any setting. We also ran QCSP for varying the value of k between 10 and 10 000 on *Moby* using a fixed *max_size* of 5. The effect of a higher k on runtime behaviour is not very large, resulting in a runtime of 5 seconds for $k = 10$ and slightly increased to 25 seconds for $k = 10000$.

We conclude that, on the selected datasets, pruning reduces the number of candidates by an *order of magnitude* and improves runtime performance for higher values of *max_size*. We also conclude that the effect of a higher k on runtime behaviour is not very large. The effect of varying *max_size* is clear — if *max_size* increases, the total runtime grows rapidly since the number of possible candidate patterns increases *exponentially*. Even so, on *Moby*, our largest dataset, with a length of over 100 000 items, finding the top 20 quantile-based cohesive sequential patterns up to a maximal size of 10 took only 39 minutes.

3.4.3 Quality Comparison

In our final set of experiments, we compare the quality of the patterns found by the different methods. We use the same parameters as in the previous Section and increase k to 250.

Synthetic Datasets

Table 3.3 shows the top 5 sequential patterns discovered by the various methods on the *Synthetic* dataset, where we embedded the sequential pattern $(1,2,3,4,5)$. We see that both GOKRIMP and QCSP rank the desired pattern first. FCI$_{seq}$ ranks the pattern in the 9th position because, due to the randomness of gaps in our generator, for some subsequences (but not all) the ratio between pattern length and average minimal window length is larger. Surprisingly, SKOPUS does not report the sequence in the top 500. It seems that the definition of expected support appears to be biased towards shorter sequential patterns.

Table 3.3: Top 5 sequential patterns for QCSP and state-of-the-art methods on *Synthetic*

FCI$_{seq}$	SKOPUS	GOKRIMP	QCSP
{1,2}	3,4,5	1,2,3,4,5	1,2,3,4,5
{3,4}	2,3,4	3,4,5	2,3,4,5
{4,5}	1,2,3	1,2,3	3,4,5
{3,4,5}	2,3,5	4,5	4,5
{1,2,3}	2,4,5	1,2	3,4

Real-world Datasets

Table 3.4 shows the top 5 sequential patterns discovered by the various methods on the text datasets, with patterns found only by a single method shown in bold (we provide the top 20 for all methods in Appendix B). FCI$_{seq}$ reports representative sequential patterns as post-processing, meaning that some occurrences of the sequential pattern might be missed (e.g., if a minimal window of $\{b, a\}$ is smaller than the minimal window of sequential pattern (a, b)). Additionally, QCSP is robust to outliers and reports sequential patterns with repeating items. Therefore, we have omitted the results for FCI$_{seq}$ as the output of QCSP will subsume the representative sequential patterns reported by FCI$_{seq}$.

Table 3.4: Top 5 sequential patterns for QCSP and the state-of-the-art methods on *Moby*, *JMLR* and *Trump*

	SKOPUS	GOKRIMP	QCSP
Moby	sperm, whale	sperm, whale	moby, dick
	white, whale	moby, dick	**mrs, hussey**
	though, yet	white, whale	**ii, octavo**
	old, man	mast, head	**crow, nest**
	moby, dick	old, man	**iii, duodecimo**
JMLR	**paper, show**	support, vector, machin	**mont, carlo**
	paper, result	real, world	**nearest, neighbor**
	paper, experi	machin, learn	support, vector
	paper, algorithm	state, art	**http, www**
	support, vector	reproduc, hilbert, space	**cross, valid**
Trump	make, america	make, america, great, again	puerto, rico
	make, great	U,S	witch, hunt
	crooked, hillary	crooked, hillary	elizabeth, warren
	hillary, clinton	fake, news	**las, vega**
	america, great	ted, cruz	goofy, elizabeth

SKOPUS, GOKRIMP and QCSP all seem to report interesting patterns. The main difference between the patterns produced by QCSP and GOKRIMP is that the former ranks on C_{quan} and only considers the *proportion* of cohesive occurrences. As such, a sequential pattern that occurs cohesively in 2 out of a total of 2 occurrences, ranks as highly as a sequence that occurs cohesively in 100 out of a total of 100 instances. By sorting the top 250 quantile-based cohesive patterns on support rather than C_{quan}, we get a ranking very close to GOKRIMP. In other words, most of the patterns found by GOKRIMP are ranked relatively highly by QCSP, but, crucially, not vice versa — many interesting patterns discovered by QCSP are not found at all by GOKRIMP. SKOPUS mostly reports short patterns, and its top 250 for *Trump* consists only of patterns of length 2. There is little overlap between the output of SKOPUS and QCSP. For instance, in *JMLR*, 59 patterns found in the top 250 by SKOPUS start with *paper*, and 44 end with *result*, while the top 250 produced by QCSP contains no patterns starting with *paper*, and only one pattern ends with *result*, namely (*experi,result*). Unlike other methods, QCSP ranks many *long* patterns in the top 250. Patterns such as (*crooked*, *hillary*, *clinton*) and (*repeal*, *replace*, *obamacare*) in the *Trump* dataset, or (*reproduce*, *kernel*, *hilbert*, *space*) and (*markov*, *chain*, *monte*, *carlo*) in the *JMLR* dataset are not found in the top 250 of the other methods at all. Less frequent patterns with high quantile-based cohesion, such as (*las,vegas*), (*mrs,hussey*), (*nearest,neighbor*), (*cross,validation*) or (*bayesian*, *network*) are not reported at all in the top 250 of SKOPUS or the limited pattern set of GOKRIMP, despite the fact that, for example, 46 out of 47 occurrences of (*puerto,rico*) are cohesive in *Trump* tweets, which clearly makes it an important pattern. We conclude that QCSP is capable of finding interesting patterns that other methods fail to discover, while not missing out on interesting patterns that other methods do find.

3.5 Related Work

Frequent Pattern Mining

In a single long input sequence, WINEPI (Mannila et al. 1997), LAXMAN (Laxman et al. 2007) and MARBLES (Cule et al. 2014) mine all frequent episodes, where frequency is defined using sliding windows, minimal windows and weighted minimal windows, respectively. It has been experimentally validated in recent research that the top-k most frequent patterns are often not very interesting (Cule et al. 2016; Petitjean et al. 2016; Fowkes and Sutton 2016).

Mining Interesting Patterns

SKOPUS by Petitjean et al. (2016) and EPIRANK by Tatti (2015) are two approaches that rank sequential patterns and general episodes, respectively, based on an elaborate definition of *leverage*. We have compared our method with SKOPUS and found that we typically rank longer patterns higher. EPIRANK ranks episodes based on the output of an existing frequent sequential pattern miner (Tatti and Cule 2012). As such, it is unlikely that less frequent or longer, but strongly quantile-based cohesive, sequential patterns reported by QCSP would be discovered by this method. Theoretically, we could mine a huge set of candidate sequential patterns by setting the support threshold very low, but this would result in pattern explosion, especially for enumerating longer patterns.

Related to EPIRANK, Tatti (2014a) also proposed a way to measure the mean and variance of minimal windows of episode occurrences, and rank episodes by comparing these values with the expected length according to the independence model. Like EPIRANK, this method also requires as input the output of an existing frequency based episode miner, making it, too, either unlikely or inefficient to produce less frequent or longer cohesive candidates than our direct approach.

Pattern reduction based on *Minimal Description Length* (MDL) produces a smaller set of patterns that covers the sequence best. We have compared with GOKRIMP (Hoang et al. 2014), which is related to other MDL-based algorithms such as SQS (Tatti and Vreeken 2012) and ISM (Fowkes and Sutton 2016). All these methods take multiple sequences as input, which is different from our approach. Another difference is that rather than enumerating as few candidates as possible and then selecting the best candidates according to an interestingness measure, they employ *heuristic search* to build a set of non-redundant patterns incrementally. Applying heuristic search to finding top quantile-based cohesive patterns is also feasible. However, we would lose the guarantee that our output contains the *exact* set of top-k most quantile-based cohesive patterns.

Constrained Frequent Pattern Mining

QCSP is also related to research in *constraint pattern mining*. Pei et al. (2007) defines a generic sequential pattern algorithm, PG, based on pattern-growth, that can handle a variety of constraints. Like Pei et al. we use a *length constraint* induced by *max_size* and a *gap constraint* induced by $\alpha \cdot max_size$. However, unlike PG, our algorithm is optimised by pruning using an upper bound on quantile-based cohesion and discovers patterns ranked according to this robust interestingness measure directly. Additionally, we take a single sequence as input and use minimal windows.

Cohesion-based Pattern Mining

As mentioned in Section 3.1, our definition of quantile-based cohesion for sequential patterns is an extension of the definition of cohesion for itemsets discussed in the previous chapter (Cule et al. 2019, 2016). For future work, it would be interesting to study if the robust definition of quantile-based cohesion and the QCSP algorithm could be applied for discovering quantile-based cohesive itemsets, episodes and association rules.

3.6 Conclusion

In this Chapter, we presented a novel method for finding interesting sequential patterns in event sequences. Compared to other interestingness measures, our quantile-based cohesion is not biased towards shorter patterns or patterns consisting of very frequent items. Furthermore, our measure is robust to the presence of outliers, and flexible, since we do not use a sliding window of fixed length, as is common in existing methods. We define quantile-based cohesion as the proportion of occurrences of the pattern that are cohesive, i.e., where the minimal window is small. This measure is easy to interpret and reports both frequent and less frequent, but always strongly correlated, sequential patterns, that other methods often fail to find. Since quantile-based cohesion is not an anti-monotonic measure, we rely on an upper bound to prune candidate patterns and their supersequences. We include this pruning function in a variant of constraint-based sequential pattern mining based on pattern growth and show both theoretically and empirically that our algorithm works efficiently.

In future work, we intend to investigate adapting our algorithm for the *anytime* setting, by prioritising more likely candidates in order to find the most interesting patterns quickly, rather than waiting for the entire output. Additionally, we are interested in applications where pattern mining is not the end goal in itself. For example, quantile-based cohesive sequential patterns could be used in tasks such as prediction, classification or anomaly detection within temporal data.

“Artificial intelligence would be the ultimate version of Google.
The ultimate search engine that would understand everything on the web.
It would understand exactly what you wanted,
and it would give you the right thing.
We’re nowhere near doing that now.
However, we can get incrementally closer to that,
and that is basically what we work on.”

- Larry Page

CHAPTER 4

Extreme Multi-label Classification using Instance and Feature Neighbours

Extreme multi-label classification problems occur in different applications such as prediction of tags or advertisements. In this Chapter[1]*, we propose a new algorithm that predicts labels using a linear ensemble of labels from instance- and feature-based nearest neighbours. In the feature-based nearest neighbours method, we precompute a matrix containing the similarities between each feature and label. For the instance-based nearest neighbourhood, we create an algorithm that uses an inverted index to compute cosine similarity on sparse datasets efficiently. We extend this baseline with a new top-k query algorithm that combines term-at-a-time and document-at-a-time traversal with pruning based on a partition of the dataset.*

On ten real-world datasets, we find that our method outperforms state-of-the-art methods such as multi-label k-nearest neighbours, instance-based logistic regression, binary relevance with support vector machines and FastXML *on different evaluation metrics. We also find that our algorithm is orders of magnitude faster than these baseline algorithms on sparse datasets, and requires less than 20 ms per instance to predict labels for extreme datasets without the need for expensive hardware.*

[1] This chapter is based on work published in the International Journal of Data Science and Analytics as "Combining instance and feature neighbours for extreme multi-label classification" by Len Feremans, Boris Cule, Celine Vens and Bart Goethals (Feremans et al. 2020).

4.1 Introduction

Multi-label classification problems occur in a large variety of domains, such as text categorization, where a document has multiple categories, scene classification, where various regions of an image have a label, and bioinformatics, where we are interested in predicting numerous functions for a gene. In this work, we consider *sparse* datasets that occur naturally in these domains, i.e., where features correspond to patterns, such as term frequency-inverse document scores for word occurrences in texts or clusters in images.

Two main strategies exist for solving the multi-label task. The first strategy is to reduce the multi-label problem into a combination of single-label problems. The *binary relevance* method ignores label dependencies and trains a separate model to predict each label independently of other labels using one-versus-all sampling (Tsoumakas and Katakis 2006). *Classifier chains* approximate label dependencies, but also require to train a separate model for each label (Read et al. 2011). If the set of labels L is large, training $|L|$ different models using binary relevance or classifier chains is *not scalable*. A second strategy is to *adapt* existing single-label classifiers to output multiple labels. Well-known adaptations of single-label classification techniques have been made to adaBoost (Schapire and Singer 1999), decision trees (Vens et al. 2008), support vector machines (Elisseeff et al. 2001), k-nearest neighbour (Zhang and Zhou 2007) and others.

Trending challenges in multi-label classification research include methods that account for possible *dependencies between labels*, deal with *label skew* (where most labels are only covered by a few instances), and consider the *computational cost* of building a model (Gibaja and Ventura 2014). *Extreme multi-label classification* is an active research topic that considers the computational cost of generating a model when the number of labels is very high. Recently several methods have been proposed that try to address these challenges. These methods reduce the *dimensionality* of the label space (Tagami 2017; Bhatia et al. 2015) or build a hierarchical *ensemble of tree-based models*, such as FASTXML, where the number of models to train is logarithmic in the number of labels (Prabhu and Varma 2014). While these approaches are accurate and fast at *testing* time, they require significant resources at *training* time. Moreover, each of these methods needs to tune many *hyperparameters* for optimal performance.

Related to multi-label classification is the field of *recommender systems*. Here, the task is to rank items a user might click, often based on past preferences. Two well-known recommender systems are user-based (Resnick et al. 1994; Breese et al. 2013) and item-based collaborative filtering (Sarwar et al. 2001). An advantage of both approaches is that the results can be *explained*, i.e., using “people who liked this item also liked" type of explanations. Applying these techniques for multi-label classification is a major goal of this work.

User-based collaborative filtering is a memory-based learning algorithm where we need to compute the *nearest neighbours*. The problem of finding the *exact* set of k-nearest neighbours is studied under different names: all pairs similarity search (Bayardo et al. 2007; Awekar and Samatova 2009), top-k set similarity joins (Xiao et al. 2009), k-nearest neighbour graph construction (Anastasiu and Karypis 2016) and top-k queries (Fagin et al. 2003; Ding and Suel 2011; Fontoura et al. 2011). We propose a new algorithm to compute the exact k-nearest neighbours using pruning. Similar to search problems in information retrieval, we want to find a set of instances in our training dataset that is the most similar to a test instance for which we wish to predict labels. A key difference with information retrieval is that our test instances, or queries, typically have many more nonzero feature values than is usual in search, which has a severe impact on performance. Therefore, we adapt research from information retrieval and create a new *top-k query* algorithm. Technically, we combine *term-at-a-time* and *document-at-a-time* traversal using *Weak-AND* pruning (Broder et al. 2003). Different from the previous work in

information retrieval by Fontoura et al. (2011), our primary reason for first traversing the instances using term-at-a-time, is based on finding proper constraints such that more candidate instances get pruned using a tighter upper-bound.

In this work, we make the following contributions. First, we implement *instance-based k-nearest neighbours*, an adaptation of user-based collaborative filtering, for multi-label classification. Next, we implement the *feature-based k-nearest neighbours* method, an adaptation of item-based collaborative filtering, that computes the nearest labels for each feature in a column-wise manner. Finally, we combine both instance- and feature-based neighbourhood predictions using a linear ensemble. Second, we make the k-nearest neighbours search scalable for sparse datasets with an extremely high number of labels, features and instances. The baseline method uses an *inverted index* and organises computation so that we only perform nonzero similarity term computations, which we improve with a top-k query algorithm.

We validate the accuracy of our method on 10 real-world datasets and compare with multi-label classification methods such as multi-label k-nearest neighbours (Zhang and Zhou 2007), instance-based logistic regression (Cheng and Hüllermeier 2009), binary relevance with support vector machines as a base learner and FastXml (Prabhu and Varma 2014) on different evaluation metrics. We also compare the pruning ability and runtime performance of our k-nearest neighbours algorithm with state-of-the-art top-k query retrieval methods, such as term-at-a-time traversal, in-memory document-at-a-time traversal with Weak-AND pruning (Broder et al. 2003; Fontoura et al. 2011) and Fagin et al. (2003) threshold algorithm. Compared to the original version of this paper (Feremans et al. 2017b), we improve our algorithms for making predictions in different ways and propose a new algorithm to compute the nearest neighbours more efficiently based on top-k queries.

The remainder of this Chapter is organised as follows. In Section 4.2 we define the problem setting. In Section 4.3 we describe our algorithm for multi-label classification. In Section 4.4 we describe our method for finding the k-nearest neighbours more efficiently. We experimentally validate our method and compare it with existing state-of-the-art methods in Section 4.5 and discuss related and future work in Section 4.6. Finally, we conclude in Section 4.7.

4.2 Problem Setting

Definition 4.1 (Multi-label dataset). *Let $X \in \mathbb{R}^{N \times M}$ denote the set of training points and let $Y \in \{0,1\}^{N \times L}$ denote the set of labels. The training dataset $\mathcal{D}$ consists of N instances $\mathcal{D} = \{(x_1, \mathbf{y_1}), \ldots, (x_N, \mathbf{y_N})\}$ where each M-dimensional feature vector $x_i \in \mathbb{R}^M$ is associated with an L-dimensional label vector $\mathbf{y_i} \in \{0,1\}^L$.*

We use $x_{i,j}$ to denote the value of feature j for instance i and $y_{i,j}$ to denote the (binary) value for label j of instance i. In the multi-label datasets we encountered, feature values are real numbers higher than or equal to zero. We have not experimented with negative values but, theoretically, this should be possible. Alternatively, we can transform feature values to positive numbers, i.e., using minmax normalisation. For datasets consisting of categorical attributes we use one-hot encoding.

Definition 4.2 (Cardinality). *We define feature cardinality as the average number of nonzero features for each instance. Analogously, we define label cardinality as the average number of labels for each instance, that is:*

$$fcard(\mathcal{D}) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} \delta(x_{i,j})$$

$$lcard(\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{L} y_{i,j},$$

where $\delta(x_{i,j})$ is 1 if $x_{i,j} > 0$ and 0 otherwise.

Definition 4.3 (Column)**.** *We use $f_j = \{x_{1,j}, \ldots, x_{N,j}\} \in \mathbb{R}^N$ to denote the column of values for feature j. Likewise we use $l_j = \{y_{1,j}, \ldots, y_{N,j}\} \in \{0,1\}^N$ to denote the column of values for label j.*

Definition 4.4 (Density)**.** *We define feature and label density as*

$$fdens(f_j) = \frac{|\{x_{i,j} \mid x_{i,j} \in f_j \wedge x_{i,j} \neq 0\}|}{N}$$

$$ldens(l_j) = \frac{|\{y_{i,j} \mid y_{i,j} \in l_j \wedge y_{i,j} \neq 0\}|}{N}.$$

For sparse datasets, we observe that feature and label cardinality are small compared to M and L, and that there is a skewed distribution where only a few features (or labels) have a high density, and most features (or labels) have a density close to 0.

Problem 4.1 (Multi-label classification)**.** *The task for multi-label classification is to predict a subset of labels for each test instance $x_q \in X_{test}$ for which the set of labels $\mathbf{y_q}$ is unknown. Formally we have to learn a function $h : X \to \{0,1\}^L$ that optimises a selected evaluation metric.*

The function h can be implemented as $h(x) = t(f(x))$ where f produces a confidence (or probability) score for each label and t is a threshold function. For instance, we can employ the *binary relevance* method and learn a binary classifier $h_\lambda : X \to \{\neg\lambda, \lambda\}$ for each label $\lambda \in L$ (Spyromitros et al. 2008). We can split this classifier into $f_\lambda : X \to [0,1]$ to compute a confidence score for each label and $t_\lambda : [0,1] \to \{\neg\lambda, \lambda\}$ to compute the final decision. In essence, the proposed method is a binary relevance method were we first compute scores for each label and then apply a single threshold, e.g. only if $f_\lambda(x_q) \geqslant 0.5$ we predict label λ.

4.3 Linear Combination of Instance- and Feature-based kNN

Our classification method consists of instance-based k-nearest neighbours (kNN), feature-based kNN and the linear combination of both predictions.

4.3.1 Instance-based kNN

The algorithm begins by searching for the k-nearest neighbours x_i in the training data for each test (or query) instance x_q using *cosine similarity*.

Definition 4.5 (Instance-based cosine similarity)**.** *The cosine similarity between two feature vectors x_q and x_i is defined as*

$$sim_{\text{INS}}(x_q, x_i) = \frac{x_q \cdot x_i}{\|x_q\|_2 \cdot \|x_i\|_2} = x_q \cdot x_i = \sum_{j=1}^{M} x_{i,j} x_{q,j},$$

where we ensure that all feature vectors are normalised to unit length during preprocessing.

In most multi-label datasets feature values are real numbers higher than or equal to zero and the cosine similarity is between 0 and 1. For dealing with negative features values we can compute the cosine similarity between -1 and 1, where a negative similarity is indicative that vectors are opposite. Normalisation has the advantage that we correct against the length of each instance, e.g., a long and short document are given equal weight. By normalising all feature vectors during preprocessing we ensure that the norm is computed once for each training instance, instead of computing the norm each time we compare a pair of instances during k-nearest neighbours search.

Definition 4.6 (Instance-based confidence score)**.** *To compute the confidence score for instance x_q for (a single) label y_j we define the following function:*

$$\hat{y}^{\text{INS}}_{q,j} = \frac{\sum_{x_i \in \text{KNN}(x_q)} y_{i,j} \cdot sim_{\text{INS}}(x_q, x_i)^{\alpha}}{\sum_{x_i \in \text{KNN}(x_q)} sim_{\text{INS}}(x_q, x_i)^{\alpha}}.$$

where

$$\text{KNN}(x_q) = \{x_i \mid x_i \in \mathcal{D} \wedge \nexists x_1, \ldots, x_k \in \mathcal{D} : \forall j \in \{1, \ldots, k\} :$$

$$sim_{\text{INS}}(x_j, x_q) < sim_{\text{INS}}(x_i, x_q)\}$$

This function is an adaption of user-based collaborative filtering (Wang et al. 2006), where the similarity in feature values replaces the similarity between user preferences, and we do not recommend an item but a label. Also, we apply a power transformation to the similarities. For example, if we apply the power $\alpha = 2$, we give similarities closer to 1 more weight compared to similarities closer to 0.01. Vice versa, $\alpha = 0.5$ has the reverse effect.

Algorithm

We retrieve the k-nearest neighbours by using an *inverted index* (IID) and compute only nonzero terms for similarity. CREATEINDEX is shown in Algorithm 4.1. We associate each feature with a set of (training) instances and their nonzero feature value.

In INSTANCEKNNSEARCH, shown in Algorithm 4.2, we compute the cosine similarity between x_q and all instances incrementally thereby only computing nonzero terms of each dot product. We first loop over each nonzero feature $x_{q,j}$, then fetch all candidates x_i that have a (nonzero) $x_{i,j}$ value from the IID and then increment the partial dot product $x_{q,j} \cdot x_{i,j}$. Finally, we use *partial sort*, i.e., using heap sort, to maintain the top k instances with the highest cosine similarity.

Algorithm 4.1: CREATEINDEX($\mathcal{D}$) Creates an inverted index for instance-based kNN baseline

Input: A dataset $\mathcal{D}$
Result: An inverted index (IID) of the dataset

```
1 IID ← EMPTY_HASH_MAP();
2 for x_i in X do                                  // For each instance
3 |   for x_{i,j} ≠ 0 in x_i do                    // For each nonzero feature value
4 |   |   IID[j] ← IID[j] ∪ ⟨x_i, x_{i,j}⟩;         // Add to inverted index
5 return IID;
```

Algorithm 4.2: InstanceKnnSearch(x_q, k, IID) Finds the k-nearest neighbours for x_q in $\mathcal{D}$

Input: A query instance x_q, number of neighbours k, IID
Result: k-nearest neighbours and their similarities

```
1 S ← EMPTY_HASH_MAP();
2 for x_{q,j} ≠ 0 in x_q do                    // For each nonzero feature value in x_q
3     for ⟨x_i, x_{i,j}⟩ in IID[j] do           // Get instances from IID
4         S_{q,i} ← S_{q,i} + x_{q,j} · x_{i,j};   // Compute similarity term
5 kNN ← HEAP_SORT_TOP_K(S, k);
6 return kNN;
```

We compute the prediction scores for each label using InstanceKnnPredict, shown in Algorithm 4.3. We only compute predictions for labels that are present in any of the k-nearest neighbours. As in InstanceKnnSearch, we organise the computation so that we only compute nonzero increments to each label score. Remark that in our implementation, we compute similarities and predictions in *parallel*. We initialise shared hash tables statically, so subsequent updates to partial scores (or similarities) from different threads can occur in a lock-free manner (Liu 2015). This results in performance gains almost *linear* with the number of processors.

Algorithm 4.3: InstanceKnnPredict(x_q, kNN, α) Computes instance-based confidence scores for labels

Input: A query instance x_q, kNN contains the k-nearest neighbours (and their similarities), α for the power transform
Result: Prediction scores for labels

```
1 ŷ ← EMPTY_HASH_MAP();
2 for x_i in kNN do                      // For each instance in kNN
3     for y_{i,j} ≠ 0 ∈ y_i do            // For each nonzero label
4         ŷ_j ← ŷ_j + S^α_{q,i};          // Compute confidence score term
5 normalise ŷ with Σ_{x_i∈kNN} S^α_{q,i};
6 return ŷ;
```

Complexity

For instance-based kNN search the complexity is $\mathcal{O}(N \times M)$, but in practice, we observe that the average runtime is closer to $\mathcal{O}(\tilde{n} \times \tilde{m})$ for sparse datasets. Here $\tilde{n}$ is proportional to the average number of *candidate* instances, i.e., instances fetched from the inverted index, and $\tilde{m}$ is proportional to the feature cardinality. We will analyse the runtime of this algorithm in Section 4.5. We remark that the expensive neighbourhood search is performed once and is *independent* of the number of labels L (Spyromitros et al. 2008).

Hyperparameter Optimalisation

An essential advantage of our method is that for optimising k using *grid search* we need to compute the nearest neighbours only *once*. First, we search for the nearest neighbours with a maximal value of k_{max}. For smaller values of k, we just take the first k values of the cached

k_{max}-nearest neighbours. We argue that this makes k a *virtual* hyperparameter, meaning that we optimise it efficiently on a validation set. This is important as we compute a weighted confidence score and often the optimal value of k is quite large, e.g., 100 or 200. We remark that for other kNN based methods the optimal value of k is quite small, e.g., 5 or 10 and this optimisation is less important.

Second-order Instance Variation

A possible disadvantage of the instance-based kNN method is that we ignore *inter-label dependencies*: the prediction for a given label is obtained independently of the values of other labels. We propose an extension that uses the second-order neighbourhood to handle this situation. Details on this variation of instance-based kNN and experimental results are discussed in Appendix C.

4.3.2 Feature-based kNN

Feature-based kNN is an adaptation of item-based collaborative filtering for multi-label classification (Sarwar et al. 2001).

Definition 4.7 (Feature-based cosine similarity)**.** *For feature-based predictions, we compute the cosine similarity between each feature column f_i and each label column l_j using,*

$$sim_{\text{FL}}(f_i, l_j) = \frac{f_i \cdot l_j}{\|f_i\|_2 \cdot \|l_j\|_2} = f_i \cdot l_j = \sum_{k=1}^{N} x_{k,i} y_{k,j},$$

where we make sure that all feature and label vectors are normalised to unit length during preprocessing.

Definition 4.8 (Feature-based confidence score)**.** *We compute the confidence score for a test instance x_q and a label y_j using:*

$$\hat{y}^{\text{FL}}_{q,j} = \frac{\sum_{i=1}^{M} x_{q,i} \cdot sim_{\text{FL}}(f_i, l_j)^{\beta}}{\sum_{i=1}^{M} x_{q,i}},$$

where we apply the power β to the similarities.

We compute the full similarity matrix between all pairs of feature and label columns at training time. When L is extremely large, we consider a variation to feature-based kNN, that only computes a (nonzero) prediction for labels that occur at least once in the neighbourhood, i.e.,

$$y_j \in \bigcup_{x_{q,i} \in x_q \wedge x_{q,i} \neq 0} \text{KNN}(f_i),$$

which is more scalable given an extreme number of labels.

Algorithm

For feature-based kNN we first compute a matrix containing the similarities between all features and labels in $\mathcal{D}$. We use sparse data structures as we assume most values for label column y_j and feature column f_k will be 0. We use a technique similar to INSTANCEKNNSEARCH and compute the feature-based cosine similarity incrementally. CREATESIMILMATRIX is shown in Algorithm 4.4. First, we create an index that associates each of the L labels with a set of positive

Algorithm 4.4: CREATESIMILMATRIX($\mathcal{D}$) Computes similarities between all features and labels for feature-based kNN

Input: A dataset $\mathcal{D}$
Result: Similarity matrix S

```
   /* Create inverted index for labels */
1  IID ← EMPTY_HASH_MAP();
2  for ⟨x_i, y_i⟩ in D do                          // For each instance
3  |  for y_{i,j} ≠ 0 in y_i do                    // For each nonzero label value
4  |  |  IID[j] ← IID[j] ∪ {x_i};
   /* Compute similarities */
5  S ← 0.0^{M×L};
6  for y_j ≠ 0 in IID do                           // For each label
7  |  for x_i in IID[j] do                         // Get instances from IID
8  |  |  for x_{i,k} ≠ 0 in x_i do                 // For each nonzero feature k
9  |  |  |  S_{j,k} ← S_{j,k} + x_{i,k} · y_{i,j}; // Compute similarity term
10 return S;
```

instances. After indexing, we fetch positive instances x_i for each label. For each instance x_i, we traverse over each nonzero feature $x_{i,k}$ and compute a nonzero term of the dot product between label y_j and feature f_k.

FEATUREKNNPREDICT is shown in Algorithm 4.5. We follow a similar approach as INSTANCEKNNPREDICT to only compute nonzero terms of each confidence score. We remark that we also experimented with an alternative confidence score that considers the k nearest features for each label, but in preliminary experiments, this did not increase average results while requiring an extra hyperparameter.

Algorithm 4.5: FEATUREKNNPREDICT(x_q, S, β) Computes feature-based confidence scores for labels

Input: A query instance x_q, a similarity matrix S, β for the power transform
Result: Prediction scores for labels

```
1  ŷ ← EMPTY_HASH_MAP();
2  for x_{q,i} ≠ 0 in x_q do                       // For each nonzero feature
   |  // For each label with nonzero similarity with feature
3  |  for S_{j,i} ≠ 0 ∈ S_{*,i} do
4  |  |  ŷ_j ← ŷ_j + x_{q,i} · S_{j,i}^β;          // Compute confidence score term
5  normalise ŷ with ∑ x_{q,i};
6  return ŷ;
```

Complexity

Computing the similarity matrix has a complexity of $\mathcal{O}(\frac{1}{2}M \times L \times N)$. However, in practice runtime is closer to $\mathcal{O}(\frac{1}{2}M \times \tilde{l} \times \tilde{n})$ for sparse datasets. Here $\tilde{l}$ is proportional to the average number of candidate labels, that is the number of labels sharing at least one instance with each feature and $\tilde{n}$ is proportional to the average number of nonzero feature values (or labels) column-wise. We observe that the similarity matrix can be computed once at *training time*

for all test instances, while at *test time* only prediction scores have to be computed. This makes feature-based kNN very efficient and is arguably one of the reasons why item-based collaborative filtering is so popular in real-world web applications.

4.3.3 Linear Combination

We introduce a straightforward ensemble method based on the Linear Combination of the confidence scores of the Instance- and Feature-based k-nearest neighbours (LCIF). Combinations of the two techniques have been studied in collaborative filtering research, but not in multi-label classification (Wang et al. 2006; Verstrepen and Goethals 2014).

Definition 4.9 (LCIF confidence score)**.** *We compute the confidence score for test instance x_q for label y_i using*

$$\hat{y}_{q,i} = \lambda\ \hat{y}^{\text{INS}}_{q,i} + (1-\lambda)\ \hat{y}^{\text{FL}}_{q,i},$$

where $\lambda \in [0,1]$ is a hyperparameter that is optimised on a validation sample for each evaluation metric.

For datasets with many labels we compute this score only for candidate labels, that is, labels i that have a nonzero score for either $\hat{y}^{\text{INS}}_{q,i}$ or $\hat{y}^{\text{FL}}_{q,i}$. The main LCIF algorithm is shown in Algorithm 4.6.

4.3.4 Thresholding

To obtain a set of predicted labels, we apply a *single* threshold (Read et al. 2011; Triguero and Vens 2016).

Definition 4.10 (Single threshold)**.** *Given confidence scores $\hat{y}_{q,j}$ for instance x_q and each label y_j, we predict a set of labels using a single threshold t:*

$$h(x_q) = \{y_j \mid \hat{y}_{q,j} >= t\},\ \forall y_j \in L.$$

We determine t automatically by selecting the value of t that minimises the difference in *label cardinality* between the actual and predicted label set. That is,

$$\underset{t}{\text{argmin}} \left| \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{L} \delta(\hat{y}_{i,j} > t) - lcard(\mathcal{D}) \right|,$$

where $\delta(\hat{y}_{i,j} > t)$ returns 1 if the confidence score is higher than t and 0 otherwise. Alternatively, we make use of a *label-specific threshold* (Yang 2001; Draszawka and Szymański 2013). This allows us to lower the threshold for *minority* labels in imbalanced datasets.

Definition 4.11 (Label-specific threshold)**.** *We predict a set of labels using a separate threshold t_{y_j} for each label y_j:*

$$h(x_q) = \{y_j \mid \hat{y}_{q,j} >= t_{y_j}\},\ \forall y_j \in L$$

We remark that for many multi-label datasets, there is at least one label for every instance. Therefore, if the highest-scoring label is below the threshold, we ignore the threshold value (Spyromitros et al. 2008).

Algorithm 4.6: LCIF($\mathcal{D}, X_{test}, k, \alpha, \beta, \lambda, t$) Predicts labels based on a linear combination of instance- and feature-based weighted similarities

Input: A training dataset $\mathcal{D}$, one or more test instances in X_{test}, number of neighbours k, parameters α and β for the power transform, λ for the linear combination and t the single threshold

Result: Predicted labels for each instance in X_{test}

```
/* Train: create index and feature-based similarity matrix */
1  IID ← CreateIndex(D);
2  S ← CreateSimilMatrix(D);
   /* Predict: compute kNN and predictions for each test instance */
3  Ŷ ← ∅;
4  for x_q in X_test do
5  |   kNN_q ← InstanceKnnSearch(x_q, k, IID);
6  |   ŷ_j^INS ← InstanceKnnPredict(x_q, kNN_q, α);
7  |   ŷ_q^FL ← FeatureKnnPredict(x_q, S, β);
8  |   ŷ_q^LCIF ← λ ŷ_q^INS + (1 − λ) ŷ_q^FL;
9  |   ŷ_q ← {ŷ_{q,j} | ŷ_{q,j} ∈ ŷ_q^LCIF : ŷ_{q,j} ≥ t};
10 |   Ŷ ← Ŷ ∪ ŷ_q;
11 return Ŷ;
```

4.4 Fast kNN Search

A problem with the instance-based kNN search is that because of its inherent $O(N)$ complexity to search for the k-nearest neighbours, it does not scale to extreme datasets. In the previous section, we created a baseline algorithm optimised for sparse datasets. In this section, we improve on this baseline. This is important in interactive applications since we must perform the search at *test time* and every millisecond is important.

4.4.1 Indexing

The problem of instance-based kNN is similar to top-k queries algorithms in information retrieval. A key difference, however, is that search queries are typically much *shorter*, having less than 10 terms. Real-world search engines often limit search queries to 50 terms. In our case, each query is a test instance, that consists of many more nonzero dimensions on average. We will show that existing state-of-the-art top-k query algorithms are less efficient in this setting. Therefore, we propose a new top-k query algorithm for computing the exact set of k-nearest neighbours ranked on cosine similarity.

Top-k Query Algorithm

Our method is an extension of the work of Fontoura et al. (2011) that combines two ways to traverse the most relevant instances (or *documents*) given a certain test instance (or *query*): Document-at-a-time (DAAT) and Term-at-a-time (TAAT). Using this framework, Algorithm 4.2 computes cosine similarity following a TAAT strategy, that is, for every nonzero feature, or *term*, of a test instance we fetch all instances, or *documents*, from the inverted index and increment the partial similarity of each document with the nonzero weight in that dimension. In DAAT traversal we keep all documents in the inverted index sorted on document order and traverse

through all *posting lists* (the inverted index for each term) simultaneously similar to a merge join. Using this document-per-document manner, we can compute the complete similarity score for each document in turn. We propose to first traverse using TAAT and next using DAAT. We focus on *memory-resident* indexes, thereby assuming that memory in present-day is often large enough to maintain the complete index (Fontoura et al. 2011).

Partitioning

We observe that in real-world datasets feature values in most dimensions have a high standard deviation. For example, we can encode text documents using a bag-of-words encoding with term frequency-inverse document frequency and get a significant difference in values between terms that frequently occur in one document but seldom occur in others, and words that are infrequent in one document and frequent overall. This variation is a useful property that we exploit.

Definition 4.12 (Partition)**.** *First, we partition all instances into two disjoint sets:*

$$I_{taat} = \{x_i \mid x_i \in \mathcal{D} \land \exists\, x_{i,j} \in x_i : rank(x_{i,j}, f_j) \leq m\},$$

$$I_{daat} = \mathcal{D} \setminus I_{taat},$$

where we use $rank(x_{i,j}, f_j) \leq m$ *to denote that feature value* $x_{i,j}$ *is ranked before place* m *in the (descending) ordered posting list of feature* j*.*

The partition strategy has three useful consequences. Firstly, by using the partitioned and sorted inverted index, we encounter high feature values first during TAAT traversal and are more likely to find instances with a high cosine similarity early on. This is important since we prune instances during DAAT if the similarity cannot be higher than the kth candidate after TAAT traversal. Secondly, we use Weak-AND (WAND) for pruning during DAAT traversal proposed by Broder et al. (2003). The WAND upper bound depends on the maximum feature value in each dimension. Because of the partitioning, we guarantee that this maximum is smaller than the first m values. Thirdly, by having two disjoint partitions, the additional overhead for pruning and index creation is minimal.

Algorithm

The algorithm CREATEINDEXPARTITION for both TAAT and DAAT index creation is shown in Algorithm 4.7 and consists of two phases. In the first phase, we create an inverted index for all documents and compute the partition of all instances. For each feature or term, we find the instances with the m highest feature values and add these instances to I_{taat}. We can compute this efficiently using heap sort on the posting list for each term. In the second phase, we make a complete TAAT index by adding all feature values for all instances in I_{taat}. Remark that this index also includes feature values that are *not* in the top-m. The rationale for making the index complete is that we can compute the full similarity for each instance in I_{taat} without resorting to less efficient random access operations. Finally, we add all remaining documents to the DAAT index sorted on document ID. We also maintain the maximal feature value in each posting list. In our implementation, we keep the feature values local to the index as we want to avoid cache misses. We remark that in practice we set the parameter m to a small value, e.g., between 1 and 25 for large datasets and closer to 100 for extreme datasets. Alternatively, m could be defined relatively as the percentage of all documents in I_{taat}.

Algorithm 4.7: CREATEINDEXPARTITION($\mathcal{D}, m$) Partitions data and builds indexes for both TAAT and DAAT traversal

Input: A dataset $\mathcal{D}$, a parameter m that controls the partition
Result: Index structures for TAAT en DAAT traversal

```
1  IID ← CREATEINDEX(D);
   /* Compute partition */
2  I_taat ← ∅;
3  for f_j in D do
4  |   {⟨x'_1, x'_{1,j}⟩, ..., ⟨x'_m, x'_{m,j}⟩} ← HEAP_SORT_TOP_K(IID[f_j], m);
5  |   I_taat ← I_taat ∪ {x'_1, ..., x'_m};
6  I_daat ← D \ I_taat;
   /* Create indexes */
7  IID_taat ← CREATEINDEX(I_taat);
8  IID_daat ← CREATEINDEX(I_daat);
9  SORT IID_taat DESCENDING on feature value;
10 SORT IID_daat ASCENDING on document id;
11 MAX_daat ← MAX feature value for each f_j in I_daat;
12 Φ ← {I_taat, I_daat, IID_taat, IID_daat, MAX_daat};
13 return Φ;
```

4.4.2 TAAT and DAAT Traversal with Weak-And Pruning

In essence, DAAT is a merge join over the different posting lists sorted on document ID. We can, however, skip instances based on an upper bound using the Weak-And iterator first proposed by Broder et al. (2003).

Definition 4.13 (Upper bound cosine similarity). *Given a query x_q we compute an upper bound on each cosine similarity term:*

$$UB_{q,j} = max(\{x_{i,j} \mid x_i \in I_{daat}\}) \cdot x_{q,j},$$

where for any $x_i \in I_{daat}$ it holds that

$$sim_{\text{INS}}(x_q, x_i) \leq \sum_{x_{q,j} \in x_q} UB_{q,j}.$$

Therefore, we prune instances without computing the full similarity if

$$\sum_{x_{q,j} \in x_q \wedge x_{i,j} \neq 0} UB_{q,j} \leq \sum_{x_{q,j} \in x_q} UB_{q.j} \leq \theta,$$

where θ is the kth largest similarity of instances already visited during TAAT and the ongoing DAAT traversal. Remark that for most instances x_i only a small subset of the features of x_q will also be nonzero in x_i.

Algorithm

The main algorithm for finding the *exact* k-nearest neighbours is shown in Algorithm 4.8. INSTANCEKNNFAST consists of two phases. We start by computing the k nearest neighbours of all instances in I_{taat} using TAAT by calling INSTANCEKNNSEARCH. We then add these instances

to a *heap*. A heap is more efficient for managing the current instances as we want to access the current k^{th} maximal similarity efficiently. In the second phase, we traverse candidate instances using DAAT (line 3-20) and our extension to WAND pruning. We start by initialising the upper bound for x_q by computing the dot product between every nonzero feature value $x_{q,j}$ and the precomputed value $\text{MAX}_{daat}[j]$. Next, we iterate over instances in I_{daat} that have at least one feature shared with x_q (as determined by IID_{daat}). We start our while loop with the first document ($offsets_j = 0$) in each posting list and order these instances on ascending ID (line 8-11). If x_{k_1} is the document with the smallest ID for any posting list then for any other document x_{k_2}, with $k_2 > k_1$, we know that x_{k_1} has a zero value in that posting list. Therefore, we prune x_{k_1} if UB_{q,k_1} is smaller than θ. Likewise, we prune x_{k_2} if $UB_{q,k_1} + UB_{q,k_2}$ is smaller than θ, etc. We increment the upper bound UB_{cur} in a feature-by-feature manner and prune any documents that are below this accumulated value (line 12-17). We stop when a *pivot* document is identified, meaning the first document that is higher than the upper bound. We then compute the full similarity for the pivot instance and add it to the heap, where it will replace a candidate if the similarity is higher (line 19). Finally, we advance each posting list to the next document. If the next document identifier is larger than the pivot, we do not update the offset. Otherwise, we advance the posting list to point to a document with an identifier after the current pivot. In the worst case, the pivot document is always the first candidate, and we

Algorithm 4.8: INSTANCEKNNFAST(x_q, k, Φ) Finds the exact k-nearest neighbours from $\mathcal{D}$ based on two traversal strategies and pruning

```
   Input : A query instance x_q, number of neighbours k, partitioned inverted index
           structures Φ
   Result: k-nearest neighbours
   /* (i) TAAT traversal */
 1 kNN_taat ← INSTANCEKNNSEARCH(x_q, k, IID_taat);
 2 heap ← CREATE_HEAP(kNN_taat);
   /* (ii) DAAT traversal and pruning using WAND */
 3 UB_q ← 0.0^|x_q|;
 4 for x_q,j ≠ 0 in x_q do                              // Computer upper bound
 5 │  UB_q,j ← x_q,j · MAX_daat[j];
 6 offsets ← 0^|x_q|;
 7 while offsets ≠ [−1,...,−1] do
 8 │  next ← {} ;                                      // Enumerate next instances
 9 │  for x_q,j ≠ 0 in x_q and offsets_j ≠ −1 do
10 │  │  next ← next ∪ IID_daat[j][offsets_j];
11 │  next ← SORT ASCENDING on document ID;
12 │  pivot ← ∅ ;                             // Find the first candidate or pivot
13 │  UB_cur ← 0;
14 │  for ⟨x_i, x_i,j⟩ in next do
15 │  │  UB_cur ← UB_cur + UB_q,j;
16 │  │  if UB_cur > MIN_HEAP(heap) then
17 │  │  │  pivot ← x_i; break;
18 │  sim_q,p ← x_q · pivot ;                      // Compute similarity with pivot
19 │  heap ← PUSH_POP(heap, pivot, sim_q,p);
20 │  Advance offsets so next instances are after pivot
21 return heap;
```

advance by one document at a time. In the best case, however, there are $|x_q|$ documents and the pivot document is the last document. Then we can advance by $|x_q|-1$ documents, thereby pruning these documents without computing the full similarity.

4.5 Experiments

We now study the accuracy and efficiency of LCIF and INSTANCEKNNFAST.

4.5.1 Experimental Setup

Datasets

We have selected five *large* and five *extreme* datasets. Table 4.1 shows the most important characteristics of each dataset. The datasets are available in well known multi-label repositories (Tsoumakas et al. 2011; Read et al. 2016; Bhatia et al. 2016).

The *Medical* dataset consists of nearly 1 000 documents containing free clinical text, originally collected at a children's hospital medical centre's department of radiology. The problem is to assign one or more medical diagnoses or procedures coded using ICD-9-CM based on free clinical text. The documents are represented using a sparse *bag-of-words* encoding. The *Corel5k* dataset is a scene classification dataset. Labels represent familiar concepts such as sea, sky, cat or forest. The images are represented using 499 binary features. A feature value of 1 indicates that a certain segment in the image belongs to a certain cluster. The *Bibtex* dataset represents a tag assignment problem. The *Delicious* dataset is similar to *Bibtex*. *Wiki10* corresponds to 20 000 Wikipedia articles. For these three datasets, the labels (or tags) were assigned using the social tagging sites Bibsonomy and Del.icio.us. Note that the label cardinality with social tagging is higher. In the *IMDB-F* dataset, the task is to assign one or more of the 28 movie genres, based on movie summary texts from IMDB. This dataset is larger, containing more than 100 000 summaries. However, there are only 28 movie genres, and the total dictionary of terms is limited.

The *extreme Eurlex* dataset is a collection of documents about European law and has close to 4 000 categories. Reuters Corpus Volume I (*RCV1*) is a benchmark dataset for text categorisation

Table 4.1: Characteristics of five large and five extreme multi-label datasets

Dataset	Train N	Test N_{test}	Features M	Labels L	*lcard*	*fcard*	Avg. *ldens*	Avg. *fdens*
Medical	333	645	1 449	45	1.2	13.4	0.0277	0.0092
Corel5k	5 000	500	499	374	3.5	8.3	0.0094	0.0166
Bibtex	4 880	2 515	1 836	159	2.4	68.7	0.0151	0.0374
Delicious	12 920	3 185	500	983	19.1	18.3	0.0193	0.0366
IMDB-F	72 551	48 368	1 001	28	2.0	19.4	0.0714	0.0194
Eurlex	15 539	3 809	5 000	3 956	5.3	237.0	0.0013	0.0474
Wiki10	14 147	6 617	101 890	30 940	18.6	669.0	0.0006	0.0065
RCV1	623 847	155 962	46 672	2 456	4.8	74.0	0.0019	0.0016
AmazonCat	1 186 239	306 782	203 873	13 330	5.1	71.1	0.0004	0.0003
WikiLSHTC	1 778 352	587 085	1 617 899	325 056	3.3	42.5	0.0001	0.0001

containing more than 700 000 labelled news articles made available by the press agency Reuters. *AmazonCat* contains over a million instances of Amazon products with labels and reviews. We selected the version that has more than 13 000 labels. Finally, *WikiLSHTC* consists of more than a million instances and features and 325 000 categories. The source is Wikipedia. A large number of features is due to the large corpus size. For this dataset, there is also a hierarchy between the labels available which we ignore. The dataset originates from the large-scale hierarchical text classification challenge (Partalas et al. 2015). *Eurlex*, *Wiki10*, *RCV1*, *AmazonCat* and *WikiLSHTC* are extreme datasets since they have thousands of labels (Bhatia et al. 2016). We remark that although the extreme datasets contain more labels and features, they are also extremely *sparse* and the cardinality of labels and features remains comparable to the large datasets. A key advantage is that we optimised our method for sparse datasets.

State-of-the-art Methods

For the large datasets, we compare the performance of the instance- and feature-based kNN methods and their linear combination LCIF with the following state-of-the-art algorithms. Multi-label k-nearest neighbours (ML-KNN) (Zhang and Zhou 2007) and instance-based logistic regression (IBLR) (Cheng and Hüllermeier 2009) are two seminal instance-based multi-label algorithms. Binary relevance with support vector machines as a binary classifier (BR-SMO) is one of the best-performing algorithms (Zeng et al. 2008; Tsoumakas and Katakis 2006). For optimising the threshold for the other methods, we use OneThreshold, which optimises a single threshold on the selected evaluation metric (Read et al. 2008). For the extreme datasets, we compare the results of LCIF with the published results of FASTXML (Prabhu and Varma 2014), a fast tree-based method for extreme multi-label classification.

For the state-of-the-art methods, we use the implementations available in the Mulan library (Tsoumakas et al. 2011). We implemented LCIF in C++ and made the source code publicly available[2]. We use 64 threads to compute similarities and predictions in parallel on a test server from 2013, which has two 8-core processors (Intel E5-2690) and 64 GB RAM. Remark that we cannot use Mulan, or Meka (Read et al. 2016), for extreme datasets since methods like BR-SMO employ the binary relevance strategy for training L binary classifiers, which is not feasible.

Evaluation Metrics

Multi-label evaluation metrics can be organised in different ways. *Examplebased* evaluation metrics are averaged over all instances. *Label-based* evaluation metrics look at the different ratios between true-positive, false-positive and false-negative predictions for each label. Label--based *micro* scores give each instance the same weight, while *macro* scores give each label the same weight, giving equal weight to frequent and infrequent labels. Within the Example-based category, we make the distinction between metrics based on the bipartition between relevant and non-relevant labels, metrics based on the *ranking* of confidence scores, and metrics based on the individual score for each label.

Definition 4.14 (Example-based metrics)**.** *We report the following example-based multi-label evaluation metrics:*

$$\textit{Example-based Accuracy} = \frac{1}{N}\sum_{i=1}^{N}\frac{|\mathbf{y_i} \cap \hat{\mathbf{y}}_\mathbf{i}|}{|\mathbf{y_i} \cup \hat{\mathbf{y}}_\mathbf{i}|},$$

$$\textit{Hamming loss} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{L}|\mathbf{y_i} \triangle \hat{\mathbf{y}}_\mathbf{i}|,$$

[2] https://bitbucket.org/len_feremans/lcif

where N is the number of test instances, $\hat{\mathbf{y}}_\mathbf{i}$ *is the predicted set of labels and* $\mathbf{y_i} \triangle \hat{\mathbf{y}}_\mathbf{i}$ *is the symmetric difference (or XOR) of actual and predicted labels.*

Definition 4.15 (Label-based metrics)**.** *We define for each label* y_k *the number of true positives, false negatives, false positives and corresponding metrics as*

$$tp = \sum_{i=1}^{N} \delta(y_k \in \mathbf{y_i} \wedge y_k \in \hat{\mathbf{y}}_\mathbf{i}) \qquad fn = \sum_{i=1}^{N} \delta(y_k \in \mathbf{y_i} \wedge y_k \notin \hat{\mathbf{y}}_\mathbf{i})$$

$$fp = \sum_{i=1}^{N} \delta(y_k \notin \mathbf{y_i} \wedge y_k \in \hat{\mathbf{y}}_\mathbf{i}) \qquad precision = \frac{tp}{tp + fp}$$

$$recall = \frac{tp}{tp + fn} \qquad F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}.$$

For label-based evaluation metrics, we report both micro and macro F1 metrics. *Micro F1* is based on the previous definitions but based on totals of true positives, false negatives and false positives over all labels L. For *macro F1*, we first compute precision and recall for each label separately and then compute the average.

Definition 4.16 (Micro and macro precision)**.** *We define micro and macro precision (analogous for recall and F1) as:*

$$precision_{micro} = \frac{\sum_{j=1}^{L} tp_j}{\sum_{j=1}^{L} tp_j + \sum_{j=1}^{L} fp_j} \qquad precision_{macro} = \frac{1}{L} \sum_{j=1}^{L} \frac{tp_j}{tp_j + fp_j}.$$

For the extreme datasets, we omit hamming loss which was close to 0 given the extreme number of labels and is a less suitable metric in such settings (Jain et al. 2016). Instead, we report precision@k.

Definition 4.17 (Precision at k)**.** *We compute precision@k based on the k predictions with the highest confidence score, defined as:*

$$precision@k = \frac{1}{N \cdot k} \sum_{i=1}^{N} \sum_{y_j \in \mathbf{y_i}} \delta(rank(y_j, \hat{\mathbf{y}}_\mathbf{i}) \leq k),$$

where $rank(y_j, \hat{\mathbf{y}}_\mathbf{i})$ *returns the rank for label* y_j *in the list of predictions sorted on descending confidence score.*

Hyperparameter Tuning

For each method, we have to optimise several *hyperparameters*. The parameter k is often set to a fixed value in other research, or only iterated over a small set of possible values (e.g., 5, 10, 15). However, optimising k can have a significant effect on reported evaluation metric values. Therefore, we vary k for ML-KNN and IBLR between 1 and 59 in steps of 2. For instance-based kNN, we vary k in steps of 50, that is $k \in \{1, 5, 50, 100, \ldots, 350\}$. We remark that the large values of k are due to the similarity weighted scores and common within user-based collaborative filtering. We vary α and β for the power transform of instance- and feature-based kNN in $\{0.5, 1.0, 1.5, 2.0\}$. For LCIF, we vary λ between 0.0 and 1.0 in steps of 0.1. Remark that we re-use the neighbour (and similarity) matrix and only compute it once for the maximal value of k speeding up grid search considerably. For optimising the single threshold t, we perform two

passes: first, we vary t between 0.0 and 1.0 in steps of 0.1 to obtain a temporary optimum t_{pass1}, and then we take steps of 0.01 and vary between $t_{pass1} - 0.05$ and $t_{pass1} + 0.05$ to obtain the final value. For the state-of-the-art methods, this is implemented by OneThreshold in Mulan. We use the same procedure for LCIF but minimise the difference between predicted and actual label cardinality (see Section 4.3.4) instead of maximising a selected evaluation metric. Finally, for the extreme datasets, we employ *feature selection* and select the top s features using *entropy*. We search for the optimal value of $s \in M \times \{0.01, 0.1, 0.25, 0.5, 0.75, 0.99, 1.0\}$. Note that we do not perform a full grid search, but instead first find the optimal value of s, assuming default parameters for other values (i.e., $k = 100, \lambda = 0.5, \alpha = \beta = 1.0$). Next, we find the optimal value of k assuming $\alpha = 1.0$ and the value of s previously found, then for α using the optimal k and s, then for β, λ and finally for t using the previously computed parameters.

To have a stable estimate for hyperparameters selected using *grid search*, we perform 10-fold *cross-validation* for LCIF on the training set for the large datasets. After the 10-fold cross-validation search finishes, we choose the average parameter combination that optimises the selected evaluation metric on the training data. For ML-KNN and IBLR, we report the optimal hyperparameters optimised directly on the test set, thereby making the *Oracle* assumption for performance reasons. For BR-SMO, we keep default parameters for the (linear) kernel function. For the extreme datasets, we skip 10-fold cross-validation for performance reasons and instead perform grid search on a sample consisting of the first 10 000 instances (1 000 instances for *Eurlex* and *Wiki10*). We report results computed on the publicly available train-test splits.

4.5.2 Classification Performance LCIF

Here we discuss accuracy on different evaluation metrics for large and extreme datasets.

Large Datasets

We compare our algorithms based on a variety of evaluation metrics for multi-label classification. This is common practice since different methods have different biases towards each metric (Gibaja and Ventura 2014). Table 4.2 shows the results for each different evaluation metric on the large datasets for LCIF and the selected state-of-the-art multi-label classifiers. Results highlighted in bold perform best on the selected metric and dataset. Missing values for IBLR for the *Delicious* and *IMDB-F* datasets are due to time-out on our test server.

If we compare the ranking of all algorithms, we see that LCIF performs better than ML-KNN, IBLR and BR-SMO for both accuracy, micro F1 and macro F1. On hamming loss, BR-SMO performs best, but the difference with LCIF is small. We also see that instance-based kNN ranks second for both accuracy, micro F1 and macro F1 outperforming both ML-KNN and BR-SMO. The feature-based kNN, by itself, does not perform great, but is comparable with ML-KNN, while requiring no k-nearest neighbour search at test time. Note that the current version of feature-based kNN is significantly better than the preliminary version described in (Feremans et al. 2017b). Compared to the original version, we now compute the full similarity matrix and not only the top-k highest similarities and scale similarities using the parameter β.

Extreme Datasets

For extreme multi-label classification (XML) we compare with the published results of FASTXML (Prabhu and Varma 2014; Bhatia et al. 2016). Table 4.3 shows the results on the extreme datasets. Results of micro or macro F1 are generally not available for other XML methods and are thus provided for information. LCIF performs better than FASTXML on *Eurlex*, *Wiki10*

Table 4.2: Comparing the accuracy of LCIF with ML-KNN, IBLR and BR-SMO on the large datasets

Metric	Datasets	INS. KNN	FEAT. KNN	LCIF	ML-KNN	IBLR	BR-SMO
Accuracy ↑	*Medical*	0.563	0.601	0.636	0.421	0.442	**0.699**
	Corel5k	0.163	**0.174**	0.170	0.147	0.105	0.098
	Bibtex	**0.347**	0.218	0.341	0.208	0.174	0.321
	Delicious	**0.230**	0.122	**0.230**	0.193	*n/a*	0.130
	IMDB-F	**0.250**	0.235	**0.250**	0.244	*n/a*	0.005
	Avg. rank	*2.2*	*3.4*	***1.8***	*4.2*	*5.6*	*3.8*
Micro F1 ↑	*Medical*	0.622	0.645	0.690	0.505	0.506	**0.773**
	Corel5k	0.266	0.263	**0.274**	0.245	0.163	0.166
	Bibtex	0.426	0.276	**0.427**	0.315	0.250	0.416
	Delicious	0.368	0.214	**0.369**	0.322	*n/a*	0.224
	IMDB-F	0.341	0.337	0.346	**0.358**	*n/a*	0.014
	Avg. rank	*2.6*	*4.0*	***1.4***	*3.6*	*5.8*	*3.6*
Macro F1 ↑	*Medical*	0.339	0.447	**0.492**	0.245	0.247	0.457
	Corel5k	0.315	0.308	0.315	**0.326**	0.161	0.317
	Bibtex	0.321	0.129	**0.328**	0.170	0.147	0.315
	Delicious	0.180	0.067	**0.181**	0.088	*n/a*	0.098
	IMDB-F	0.090	0.055	**0.084**	0.056	*n/a*	0.011
	Avg. rank	*2.5*	*4.6*	***1.7***	*3.6*	*5.6*	*3.0*
Hamming loss ↓	*Medical*	0.025	0.026	0.021	0.024	0.029	**0.012**
	Corel5k	0.014	**0.010**	**0.010**	0.022	0.027	0.012
	Bibtex	0.017	0.017	**0.014**	0.020	0.021	0.016
	Delicious	0.025	0.030	0.024	0.021	*n/a*	**0.018**
	IMDB-F	0.094	0.083	0.082	0.101	*n/a*	**0.072**
	Avg. rank	*3.9*	*3.6*	*1.9*	*4.0*	*6.0*	***1.6***

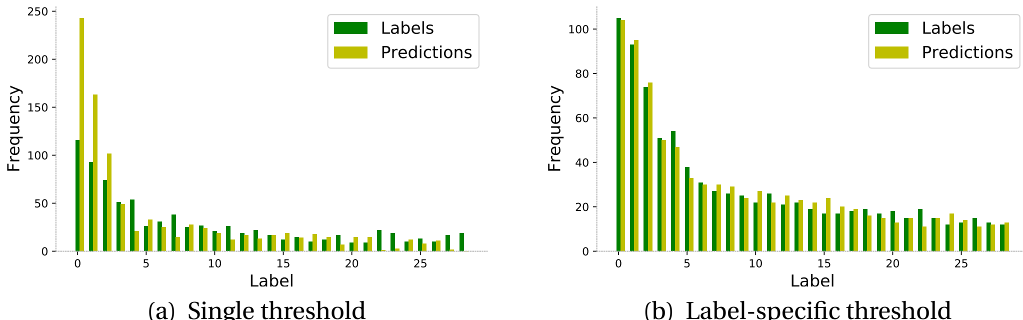
and *WikiLSHTC*, but not on *AmazonCat*. We remark that many other recent, rank-based optimised, XML methods exist (Bhatia et al. 2016). However, in classification benchmarks, the best results are reported by ensemble methods that combine many models, possibly using different algorithms and feature representations. Comparing a single simple model with a complex ensemble-based method would not be informative. We conclude that our method produces excellent results on extreme datasets.

Imbalanced Datasets

LCIF improves the accuracy on minority labels in imbalanced datasets. Firstly, we remark that feature-based cosine similarity is corrected for label imbalance since we normalise all label vectors to unit length during preprocessing. As such, weights for infrequent labels (and features) will be much higher. In Figure 4.1 we show the frequency of the top 30 most frequent labels (and predicted labels) on the *Corel5k* dataset where hyperparameters are optimised on the macro F1 metric during grid search. By using a single threshold, our method overestimates majority labels and underestimates minority labels. However, by using label-specific thresholds,

Table 4.3: Comparing the accuracy of LCIF with FASTXML on the extreme datasets

Metric	Dataset	INS. KNN	FEAT. KNN	LCIF	FASTXML
Micro F1 ↑	*Eurlex*	0.506	0.222	**0.517**	*n/a*
	Wiki10	**0.359**	0.274	0.358	*n/a*
	RCV1	**0.637**	0.450	0.632	*n/a*
	AmazonCat	0.611	0.418	**0.625**	*n/a*
	WikiLSHTC	0.329	0.144	**0.342**	*n/a*
Macro F1 ↑	*Eurlex*	0.509	0.381	**0.512**	*n/a*
	Wiki10	**0.324**	0.261	**0.324**	*n/a*
	RCV1	**0.152**	0.071	**0.152**	*n/a*
	AmazonCat	**0.463**	0.285	**0.463**	*n/a*
	WikiLSHTC	**0.130**	**0.130**	**0.130**	*n/a*
Precision@1 ↑	*Eurlex*	0.763	0.407	**0.776**	0.713
	Wiki10	**0.832**	0.722	**0.832**	0.830
	RCV1	0.830	0.762	**0.840**	*n/a*
	AmazonCat	0.775	0.657	0.811	**0.931**
	WikiLSHTC	0.491	0.258	**0.518**	0.497
Precision@3 ↑	*Eurlex*	0.611	0.302	**0.622**	0.599
	Wiki10	**0.724**	0.531	**0.724**	0.675
	RCV1	0.663	0.615	**0.672**	*n/a*
	AmazonCat	0.665	0.589	0.698	**0.782**
	WikiLSHTC	0.317	0.178	**0.336**	0.331
Precision@5 ↑	*Eurlex*	0.504	0.244	**0.514**	0.504
	Wiki10	**0.637**	0.469	**0.637**	0.578
	RCV1	0.480	0.449	**0.487**	*n/a*
	AmazonCat	0.547	0.504	0.577	**0.634**
	WikiLSHTC	0.238	0.141	**0.251**	0.244



(a) Single threshold (b) Label-specific threshold

Figure 4.1: Effect of thresholding on the distribution of actual versus predicted labels on the imbalanced dataset *Corel5k*

we get a better match in the distribution of actual versus predicted labels. Here, we base the class-specific thresholds on the prior distribution of classes in the training dataset. We report an average gain of 1.1% on macro F1 on the large datasets.

4.5.3 Runtime Performance INSTANCEKNNFAST

We now compare INSTANCEKNNFAST with the following state-of-the-art methods in top-k query retrieval: TAAT without pruning, Fagin et al. (2003) Threshold Algorithm (FAGIN TA) and Fontoura et al. (2011) in-memory variant of DAAT with Weak-AND pruning (M-WAND).

Analysis Term-at-a-time

First, we analyse the runtime behaviour of baseline kNN algorithm INSTANCEKNNSEARCH shown in Algorithm 4.2. This algorithm has three properties that make it efficient. We will use dataset characteristics from the extreme dataset *WikiLSHTC* for illustration (see Table 4.1). Firstly, we compute a *sparse dot product* between the query instance and each instance from the training dataset. In this dataset, there are $M \approx 1.6 \times 10^6$ features, however on average an instance has only 42 nonzero features, i.e., *fcard* is 42. Clearly, computing a million of zero multiplications for the naive full dot product is wasteful. Using the TAAT traversal strategy, we only compute $x_{q,j} \cdot x_{i,j}$ terms that have a nonzero value for feature value $x_{i,j}$. Secondly, the inverted index causes a form of *rudimentary pruning* by only considering candidate instances x_i having a nonzero feature in common with the query instance. We experimented on *WikiLSHTC* and computed the average number of candidates for 500 random test instances. We found that on average, only for 42% of instances the similarity is computed, thereby pruning about a million of instances from the training dataset (see Table 4.4). Thirdly, we also compute *confidence scores in a sparse manner*, thereby only computing nonzero terms for the confidence score for candidate labels, i.e., labels that occur for at least one neighbour. This is important since on average, each training instance has only 3 labels, i.e. *lcard* is 3.3, while $L \approx 0.3 \times 10^6$.

Pruning Performance

The runtime performance is dependent on pruning, i.e., the number of candidate instances for which we compute the cosine similarity. We compare the average *number of candidate instances* for each state-of-the-art method. We set k to 100 and use the first 1 000 test instances to compute this average. We remark that INSTANCEKNNFAST is identical to M-WAND if $m = 0$, and identical to INSTANCEKNN when m is set high (such that $\mathcal{I}_{daat} = \emptyset$). For INSTANCEKNNFAST, we vary the hyperparameter $m \in \{1, 5, 10, 15, 20, 25, 100\}$ (not $m = 0$) and assume an Oracle that selects the best parameter.

The results are shown in Table 4.4 where we report both the absolute value and relative percentage of the average number of candidates. For the large datasets, all features are binary, however, after normalisation to unit length there is more variation in feature values, which is beneficial for pruning with our method. On the large datasets, we see this effect, where INSTANCEKNNFAST evaluates fewer candidate instances compared to other techniques. For example, on the *IMDB-F* dataset, we compute cosine similarity for 46% of the training instances for INSTANCEKNNSEARCH, 39% for FAGIN TA, 28% for M-WAND and only 22% for INSTANCEKNNFAST. If we look at the extreme datasets, we find that M-WAND outperforms both FAGIN TA and INSTANCEKNNFAST on the majority of datasets. For small values of m, the number of instances in I_{taat} is large, leading to a higher number of full evaluations. By considering only instances with a maximum feature value for low-density features, this could be resolved, but we

Table 4.4: Pruning of INSTANCEKNNFAST and state-of-the-art top-k query retrieval methods

Dataset	FAGIN TA	M-WAND	INS. KNNFAST	INS. KNNSEARCH
Avg number of candidate instances ↓				
Medical	**197 (59%)**	214 (64%)	214 (64%)	224 (67%)
Corel5k	681 (14%)	641 (14%)	**455 (10%)**	719 (16%)
Bibtex	**4 404 (90%)**	4 553 (93%)	4 762 (97%)	4 860 (99%)
Delicious	3 412 (26%)	3 120 (24%)	**3 115 (24%)**	4 141 (32%)
IMDB-F	28 389 (39%)	20 716 (28%)	**16 170 (22%)**	33 509 (46%)
Eurlex	14 091 (91%)	**11 785 (76%)**	12 355 (79%)	15 528 (99%)
Wiki10	13 727 (97%)	**12 592 (89%)**	14 110 (99%)	14 118 (99%)
RCV1	305 002 (48%)	**176 038 (28%)**	196 453 (31%)	569 297 (91%)
AmazonCat	**195 488 (11%)**	227 326 (19%)	290 450 (24%)	741 380 (62%)
WikiLSHTC	462 939 (26%)	**367 645 (20%)**	594 107 (33%)	899 071 (50%)

will see in the next subsection, where we compare runtimes, that this is less important. Overall we conclude that INSTANCEKNNFAST for pruning is theoretically always better than or equal to M-WAND without partitioning and performs better than FAGIN TA and INSTANCEKNNSEARCH, under the assumptions of sparse datasets and high-dimensional queries.

Runtime Performance

We now compare our method with state-of-the-art-methods and report elapsed wall time. We do not report results for FAGIN TA since in our experiments we found that the random access cost and associated zero computations for computing full cosine similarity at each iteration caused much worse performance than the TAAT baseline without pruning. We report the number of milliseconds required for INSTANCEKNNSEARCH, M-WAND and INSTANCEKNN-FAST with m in $\{1, 20, 100, 500, 1000\}$. For each dataset, we take the first 1 000 test instances and report the average time it takes to retrieve the *exact* set of 100 nearest neighbours using each algorithm. From the timings (averaged over 10 runs) we excluded time needed to load the data and create the inverted indexes since this took less than 1 minute on *WikiLSHTC*.

The results are shown in Table 4.5. We omitted the results for the large datasets since the differences in milliseconds are too small. First, we remark that there is no clear one-to-one correspondence between pruning and runtime performance. Because of its simplicity and

Table 4.5: Runtime of INSTANCEKNNFAST and state-of-the-art top-k query retrieval methods

Dataset	M-WAND	INSTANCEKNNFAST m=1	m=20	m=100	m=500	m=1000	INST. KNN SEARCH
Avg time (ms) to retrieving top kNN ↓							
Eurlex	17.4	12.8	1.4	**1.3**	1.4	1.4	1.5
Wiki10	62.4	1.8	1.6	**1.5**	1.6	1.7	1.7
RCV1	103.7	97.4	43.8	**32.6**	32.6	33.9	37.5
AmazonCat	215.4	174.1	45.1	**37.7**	38.7	45.1	45.2
WikiLSHTC	171.1	94.3	35.7	27.7	27.9	**27.6**	28.6

sparse optimisations, our current implementation of the INSTANCEKNNSEARCH method is far more efficient than M-WAND. M-WAND has overhead because of the computation and book-keeping required for computing and comparing with the upper bound, such as the sort on document ID required to find the pivot. Note that both M-WAND and INSTANCEKNNFAST use the same code. We see that INSTANCEKNNFAST, when m is set appropriately large, outperforms both state-of-the-art methods by a considerable margin on all extreme datasets. This is especially so for *AmazonCat*, where it is 6 times faster than M-WAND and 25% faster than INSTANCEKNNSEARCH. We find that, on the one hand, when only a few instances in the inverted index match the current query, the overhead of computing and verifying the upper bound is probably not justified. On the other hand, when only low similarity instances remain, and there is a high likelihood for pruning, pruning becomes the faster alternative. We see this in *AmazonCat*, where for $m = 100$, 98.6% of instances are indexed for TAAT traversal and 1.4% for DAAT traversal. However, during TAAT only 39% of instances are pruned (because of the inverted index), while during DAAT more than 99% of instances (in the remaining sample of about 16.000 instances) are pruned. We conclude that INSTANCEKNNFAST finds a *natural balance* between fast unpruned TAAT traversal and fast DAAT traversal with WAND pruning.

4.5.4 Runtime Performance LCIF

Large Datasets

We now compare the total runtime required for both training and applying the model of LCIF and each of the state-of-the-art algorithms for large datasets. The results are shown in Table 4.6. For the large datasets, the difference in wall time is quite large as LCIF takes seconds or minutes where other methods take minutes or hours to complete. For *Corel5k* the relatively long runtime of 34.3 minutes for IBLR is due to learning the optimal weights using logistic regression for each label: training the optimal weights takes about 10 seconds per label but must be repeated for 374 labels. Due to this reason, IBLR was not able to complete on *Delicious* and *IMDB-F*. BR-SMO does finish for *Delicious* and *IMDB-F* but runs for a full day when LCIF takes less than 1 minute. Table 4.6 also shows the runtime of instance- and feature-based kNN. The runtime of LCIF is approximately equal to the total runtime of the two components. We conclude that LCIF is *orders of magnitude* faster than ML-KNN, BR-SMO and IBLR.

Table 4.6: Runtime results of LCIF and the state-of-the-art multi-label classifications methods on the large datasets

Dataset	FEAT. KNN	INS. KNN	LCIF	ML-KNN	IBLR	BR-SMO
Total train and test time for classifier ↓						
Medical	0.1 s	0.0 s	**0.1 s**	0.5 s	1.5 s	6.5 s
Corel5k	0.0 s	0.1 s	**0.1 s**	15.5 s	34.3 m	6.5 m
Bibtex	1.9 s	0.3 s	2.6 s	**1.7 s**	8.2 m	10.1 m
Delicious	0.6 s	0.2 s	**1.3 s**	3.3 m	*n/a*	14.0 h
IMDB-F	13.5 s	33.8 s	**49.2 s**	2.1 h	*n/a*	29.4 h

Extreme Datasets

In Table 4.7 we show the total time required by LCIF on the extreme datasets. We report subtotals for running instance-based k-nearest neighbours search, feature-based similarity

matrix computation and feature-based predictions. For LCIF the total time is the sum of these steps. The remaining time needed for data loading, indexing, making instance-based predictions, combining predictions and computing and applying a single threshold is relatively small. Additionally, we report time to perform grid search for tuning hyperparameters. All experiments were run on a single test server with very moderate hardware specifications as described previously.

On *WikiLSHTC*, LCIF took less than 3 hours to finish on more than 500 000 test instances, including grid search using a validation set of 10 000 instances. Averaged over the number of test instances this means 22 ms per instance on average, of which the bulk is required for running the instance-based k-nearest neighbours search. Feature-based kNN predictions require less than 10 minutes in total and less than 1 ms per instance on average. In FASTXML the authors report wall times of 1.5 hours on *WikiLSHTC* for training only, depending on the hyperparameters (Prabhu and Varma 2014). However, some hyperparameters, such as the parameter that controls the number of iterations have a serious influence on both runtime and precision, and it is unclear how to optimise this and the 7 other hyperparameters efficiently. We conclude that our algorithm is very efficient and does hyperparameter tuning, training and predictions in a few hours on commodity hardware for extreme datasets, taking less than 22 ms per instance to predict labels.

Table 4.7: Runtime results of LCIF on the extreme datasets

Dataset	Grid search	INS. KNN search	FEAT. KNN simil	FEAT. KNN predict	LCIF total
Total grid search, train and test time for classifier ↓					
Eurlex	51.4 s	4.6 s	0.8 s	12.7 s	24.4 s
Wiki10	9.9 m	9.8 s	26.4 s	3.6 m	5.1 m
RCV1	8.2 m	58.9 m	6.0 s	2.9 m	63.0 m
AmazonCat	11.6 m	136.3 m	28.8 s	5.7 m	145.2 m
WikiLSHTC	11.8 m	154.8 m	31.1 s	8.3 m	167.9 m

4.6 Related Work

We have examined the most important related work in Section 4.1 and experimentally compared our method with existing state-of-the-art methods in Section 4.5. We now place our work into the wider context of multi-label classification.

Instance-based Learning

Several instance-based learning methods for multi-label classification have been developed. ML-KNN was one of the first methods. Zhang and Zhou (2007) first apply traditional kNN, using Euclidean distance. Next, they count the number of times each label occurs in the neighbourhood. Then they apply the *maximum a posteriori* principle for each label independently to determine if a label is relevant or not. They estimate prior probabilities by computing kNN for each training instance and then compute these probabilities for each label. In theory, ML-KNN could also adopt an inverted index and sparse computation of similarities, probabilities and

predictions. However, the complexity is worse than that of LCIF, which is $\mathcal{O}(N \times M)$ for the kNN search and $\mathcal{O}(N \times N \times M)$ steps for computing probabilities. The authors experimentally show that ML-KNN outperformed other techniques, such as RANK-SVM on different example-based evaluation metrics. A possible disadvantage of ML-KNN is that it does not take label dependencies into account, which was addressed by subsequent research into *dependent* multi-label k-nearest neighbours (Younes et al. 2008).

In combining instance-based learning and logistic regression for multi-label classification (IBLR), Cheng and Hüllermeier (2009) compute the k-nearest neighbours using Euclidean distance. Then the authors use label counts from the nearest neighbours as a feature vector and apply logistic regression to learn the optimal hyper-plane for each label. This approach comes down to *stacking* and does account with dependencies between labels since all label counts are used as input for the logistic regression. However, applying logistic regression for each label independently does take considerable resources, as shown in our experiments, where the implementation from Mulan was unable to finish on some large datasets. Both ML-KNN and IBLR are considered state-of-the-art methods (Gibaja and Ventura 2014).

Spyromitros et al. (2008) propose BRKNN, where kNN is combined with the binary relevance method. Like BRKNN, we compute the k-nearest neighbours once, independently of the number of labels. The authors also implement two extensions: BRKNN-a and BRKNN-b. The BRKNN-b variant minimises the label cardinality between predicted and actual label sets, while BRKNN-a returns the highest-scoring label as relevant, even if this label is below the threshold since for most benchmark multi-label datasets an empty set of labels is rare. Both ideas are implemented by our method. Compared to instance-based kNN, BRKNN uses a different scoring function, which is the fraction of labels found in the k-nearest neighbours. Also, BRKNN does not make use of an inverted index. In future work, it could be interesting to experimentally validate different variations of instance-based prediction functions.

Wang et al. (2011) propose the Enhanced kNN algorithm (EKNN), that uses a weighted prediction function similar to instance-based kNN, but based on *BM25 similarity* and a more elaborate thresholding scheme. EKNN scored first in the challenge on large scale hierarchical text classification on example-based accuracy and F1 (Partalas et al. 2015). However, EKNN has a larger range of hyperparameters (both for BM25 and thresholding) to tune and is only applicable for text categorisation. The implementation of EKNN is based on an inverted index, similar to INSTANCEKNNSEARCH.

Imbalanced Datasets

Different authors have studied how to improve the accuracy of imbalanced datasets. SMOTE is an algorithm for synthetic oversampling of multi-class instances with minority labels (Chawla et al. 2002). In preliminary experiments, we tried to adopt SMOTE for oversampling instances with minority labels together with downsampling of majority labels to generate balanced datasets. However, this did not significantly improve results on macro F1. Moreover, SMOTE has additional parameters for each minority label, making adoption challenging.

Tan (2005) proposes NWKNN, a neighbour-weighted kNN algorithm that achieves a performance improvement for text categorisation on imbalanced datasets. Like NWKNN, our instance-based kNN method performs distance weighting and a power transform. Unlike NWKNN, we do not take the size of the membership of labels into account in the instance-based confidence score.

Liu et al. (2014) present a hybrid coupled k-nearest neighbour classification algorithm (HC-KNN) for mixed-type data. They employ feature weighting proportional to the number of feature-label co-occurrences and inversely proportional to the global label frequency. This is

related to feature-based cosine similarity since we measure the feature-label co-occurrences by computing the cosine similarity between each feature and label column-wise and by normalising label vectors to unit length, we are dividing by the global label frequency (assuming binary positive data). The instance-based cosine similarity is related to the inter-coupled similarity measure if we employ one-hot-encoding of categorical features during preprocessing.

There is no related work on instance-based multi-label classification optimised for imbalanced extreme datasets. For example, Liu et al. (2014) propose an optimisation procedure to learn the correspondence between each feature value and label, but this has a complexity of $O(M^3 \cdot L)$. Therefore, we adopted a label-specific threshold to improve results on macro F1 (Draszawka and Szymański 2013). However, we acknowledge that given a long tail of minority labels (e.g. occurring less than 5 times), high precision and recall remain challenging.

Other Similarity Measures

In general, many other similarity measures exist, e.g., centred cosine similarity is often used in collaborative filtering where the idea is that some users are more generous in giving rating, and we normalise against user bias in giving ratings. A more recent technique proposes soft cosine similarity (Sidorov et al. 2014), where we consider the similarity of two features, i.e., the word “play” and “game” are different but related. We remark that many similarity measures that - when computed on sparse datasets - consist mostly of zero similarity terms can also be computed efficiently by LCIF. For instance, it would not be hard to adjust LCIF to handle BM25, centred cosine, Dice or Jaccard similarity. Additionally, we have not investigated other domain-specific feature extraction techniques, i.e., different word (or sentence) embeddings for text datasets or different mid-level feature extraction methods for scene classification.

Fast Nearest Neighbours

We did not consider combining TAAT traversal with pruning (Fontoura et al. 2011). For example, we could prune entire dimensions using max_score pruning. While this technique is useful for pruning more instances, it remains uncertain if this would decrease the overall wall time for high-dimensional sparse datasets as is the case with INSTANCEKNNFAST. We also did not consider *approximate* kNN strategies used by other authors in extreme multi-label classification (Zadeh and Goel 2013; Tagami 2017). Since our algorithm can compute the *exact* set of k-nearest neighbours efficiently on extreme datasets, approximations are of less interest.

Predicting Drug Side Effects and Drug-target Interactions

Recently, Zhang et al. (2015) proposed to apply multi-label classification for predicting drug side effects. Here, the authors proposed an advanced method to select feature subsets with high information gain and use ML-KNN to make final predictions. Given that their dataset consists of thousands of labels, suffers from label skew and that in our experiments LCIF outperforms ML-KNN, it would be interesting to validate if performance would increase using LCIF in this application. In another study Zhang et al. (2017) proposed to predict interactions between drugs and target proteins. Interestingly, they also compute the similarity between instances, that is, the similarity between drugs, and predict protein targets, which are labels. In general, we argue that there are many more applications where the efficiency and accuracy of LCIF on (extreme) multi-label classification could be beneficial.

Collaborative Filtering

In this Chapter we were inspired by baseline algorithms from collaborative filtering. However, many recent methods from recommender systems could also be adapted for extreme multi-label classification. Specifically it would be of interest to adapt SLIM that improves results compares to state-of-the-art recommender algorithms and can be efficiently computed on large sparse datasets (Ning and Karypis 2011). Like item-based collaborative filtering, SLIM employs a similarity matrix between all items (or features and labels in our case). However, SLIM learns a similarity matrix that optimizes the difference with the original user-item matrix and combines L1 and L2 regularisation. The recommender systems community has also investigated the *cold-start problem*, where the aim is to improve results for new users (or items) with only limited information on past preferences (Bobadilla et al. 2012). It would also be of interest to adapt solutions to the cold-start problem to online multi-label classification, i.e., to improve results for new labels.

4.7 Conclusion

In this Chapter we propose LCIF, a new algorithm for multi-label classification inspired by recent work in recommender systems research, i.e., user-based and item-based collaborative filtering. Our predictions are based on the labels of the nearest neighbours in the training dataset. The instance-based method finds the top k instances that are most similar using the features of the current test instance. The feature-based method gives higher weight to labels that are the most similar to each feature, where similarity is defined column-wise over all instances. A linear combination of the similarity weighted instance- and feature-based neighbourhood is computed to make the final prediction.

We created an efficient algorithm for finding the k- nearest neighbours using an inverted index and efficient sparse computation of cosine similarities and predictions. We extend this algorithm and create an even faster k-nearest neighbours search algorithm, by partitioning instances and combining term-at-a-time and document-at-a-time traversal with a tighter upper bound for Weak-AND pruning. We validated that this method can be 25% faster than the baseline method, and up to 6 times faster than existing top-k query retrieval algorithms, assuming high-dimensional sparse datasets. LCIF requires only seconds to complete on large datasets, where classic methods take minutes or hours. For extreme datasets, we require less than 20 milliseconds per instance to predict labels on commodity hardware.

Experiments on ten real-world multi-label datasets from different domains, i.e., text categorisation, scene classification and social tagging domain, show that LCIF outperforms state--of-the-art algorithms such as multi-label kNN, instance-based logistic regression and binary relevance with support vector machines on accuracy, micro F1 and macro F1. LCIF also produces excellent results on extreme datasets compared to FASTXML. Because of its efficiency at both train and test time, the possibility to generate explainable results, and excellent evaluation accuracy, LCIF is interesting for any extreme multi-label application, especially when making a trade-off between model/computational complexity and performance improvement. The source code of LCIF is publicly available and enables end-users to perform accurate extreme multi-label classification without the need for expensive clusters.

In future work, we see potential to improve further extreme multi-label learning algorithms inspired by advances in the related field of collaborative filtering. We also see potential to boost the prediction accuracy of LCIF by creating ensembles using boosting or stacking and adopting

embedding algorithms, such as word or sentence embeddings learned using neural networks (Mikolov et al. 2013; Devlin et al. 2018).

"The more you know, the more you know you don't know."

- Aristotle

CHAPTER 5

Conclusion and Outlook

In this thesis, we proposed two new methods for mining interesting patterns in a sequence of events and an efficient method for multi-label classification that handles datasets with an extreme number of labels.

5.1 Main Contributions

- In **Chapter 2**, we proposed FCI$_{seq}$, an algorithm that finds different types of patterns in event sequences. We evaluate the quality of the discovered itemsets using cohesion, a measure of how far apart the items making up the itemset are on average. We discover strong patterns that are not necessarily very frequent in the data, that existing state-of-the-art methods fail to find. We use an upper bound of the cohesion, thereby generating fewer candidate patterns, and prove theoretically that this bound is sound. Based on the discovered cohesive itemsets, we then search for sequential patterns and episodes, which offer additional information about the order in which the pattern items occur. Furthermore, we mine association rules, with a confidence measure based on the cohesion of the antecedent and consequent, rather than the frequency-based definition common in literature. We integrate the mining process of all four pattern types into a single efficient algorithm. Experiments on text datasets show that we rank interesting patterns highly, including patterns with high frequency, while avoiding spurious patterns that consist of unrelated items that often co-occur purely because they all occur very frequently. Moreover, on the Zimmermann (2014a) benchmark, we validate that cohesion is more robust to a variety of artificially induced types of noise that occur in real-world settings and that we outperform both frequency-based and Compact Minimal Windows (Tatti 2014a) on this benchmark.

- In **Chapter 3** we proposed QCSP, an algorithm for mining all quantile-based cohesive sequential patterns. We define quantile-based cohesion as the proportion of occurrences of the pattern that are cohesive, i.e., where the minimal window is small. This measure is easy to interpret and reports both frequent and less frequent, but always strongly correlated, sequential patterns, that other methods often fail to find. Since quantile-based cohesion is not an anti-monotonic measure, we rely on an upper bound to prune candidate sequential patterns and all possible supersequences. We prove theoretically that this bound is sound. We empirically show that QCSP is very efficient, that is, about an order of magnitude faster than FCI$_{seq}$. FCI$_{seq}$ needs to scan the sequence for computing the

minimal windows for each candidate pattern. In contrast, QCSP uses prefix-projected pattern growth. Therefore, we evaluate the bound on quantile-based cohesion only on the - monotonically decreasing - projection of each candidate pattern. Compared to FCI_{seq}, quantile-based cohesion is also robust in the presence of outliers and finds all sequential patterns, including patterns with repeating items. Additionally, we compared with two recent algorithms that find interesting sequential patterns in multiples sequences proposed by Petitjean et al. (2016) (SKOPUS) and Hoang et al. (2014) (GOKRIMP). Compared to SKOPUS, quantile-based cohesion is not biased towards shorter patterns or patterns consisting of frequent items. Compared to GOKRIMP, QCSP seems to discover the same patterns GOKRIMP reports; however, the reverse is not true. Additionally, both methods require a sliding window of fixed length during preprocessing to create multiple sequences.

- In **Chapter 4**, we propose LCIF, a new algorithm for extreme multi-label classification inspired by user-based and item-based collaborative filtering. We make a prediction using a linear combination of similarity weighted prediction scores using both instance- and feature-based neighbourhoods. The instance-based method finds the top-k instances that are most similar using the features of the current test instance. The feature-based method gives higher weight to labels that are the most similar to each feature, where similarity is defined column-wise over all instances. The baseline algorithms use an inverted index for efficient sparse computation of cosine similarities and predictions. For the instance-based method, we create an even faster k-nearest neighbours search algorithm inspired by recent advances in information retrieval. INSTANCEKNNFAST partitions the training database and combines term-at-a-time and document-at-a-time traversal with a tighter upper bound for Weak-AND pruning. We experimentally show that INSTANCEKNNFAST can be 25% faster than the baseline method, and up to 6 times faster than existing top-k query retrieval algorithms by Fagin et al. (2003) and Fontoura et al. (2011), assuming high-dimensional sparse datasets and queries with a higher number of non-zero features than typical in information retrieval. In experiments on ten real-world multi-label datasets from different domains, we show that LCIF outperforms multi-label kNN (Zhang and Zhou 2007), instance-based logistic regression (Cheng and Hüllermeier 2009) and binary relevance with support vector machines on both accuracy, micro F1 and macro F1. Additionally, LCIF has excellent results on precision@k on extreme datasets compared to FASTXML (Prabhu and Varma 2014). To the best of our knowledge, LCIF is the only publicly available method that enables end-users to perform accurate extreme multi-label classification and requires less than 20 milliseconds per instance to train and predict labels on a single computer.

5.2 Outlook

As research evolves, there are always more things to investigate. In this section, we discuss several ideas and observations that provide some perspective for future research. We discuss possible avenues for improving the proposed algorithms, but also consider applications of pattern mining and multi-label classification. For instance, applying quantile-based cohesive sequential patterns for prediction, classification or anomaly detection within temporal data.

5.2.1 Future of Pattern Mining In Sequences

Anytime, Streaming and Big Data Cohesive Pattern Mining

In future work, we could study optimising the proposed algorithms for different settings.

First, we can study *anytime* algorithms. By prioritising more likely candidates and relaxing the problem of finding the exact set of most cohesive patterns, we can mine cohesive patterns more quickly. That is, instead of enumerating all possible candidates that can theoretically satisfy the threshold on minimal cohesion, we can employ greedy- or A* search. We can use the upper bounds on cohesion as a heuristic and stop mining after a fixed number of iterations (or stop if there is no change in the top-k most cohesive patterns after several iterations).

Second, we can investigate *streaming* cohesive pattern mining algorithms. That is, to mine the cohesive candidate patterns and then update the candidate patterns *incrementally* using periodic batch updates. Similar to Cheung et al. (1996) we can make a distinction between removing existing patterns (losers) and mining new patterns (winners). We remark that quantile-based cohesion can be computed incrementally by splitting it in two parts: occurrences of the pattern in the already mined dataset and occurrences of the pattern in the new batch. Therefore, we can efficiently update quantile-based cohesion for existing patterns and only compute the number of minimal windows satisfying the threshold on α in the new batch. For finding new candidate sequential patterns satisfying the top k threshold this becomes more difficult. Naively we could re-run QCSP on the whole updated datasets, however, it would be interesting to study re-using initial computations instead of computing everything from scratch. The following example indicates that there is potential to prune more patterns in an incremental setting. Assume a candidate sequential pattern occurring 20 times in past batches and 4 times in the new batch. If we assume a threshold of $1/4$ on minimal quantile-based cohesion on the entire dataset, we require a total of $24 \times 1/4 = 6$ cohesive occurrences. If the pattern was not cohesive enough in the original batch, we can deduce that at most 4 occurrences were cohesive. Therefore, we require that 2 additional occurrences must be cohesive in the new batch. This implies that the constraint on quantile-based cohesion should be at least $1/2$, and not $1/4$, local to the new batch. Further study is required, but the given example is indicative that by keeping summary statistics from the previous batches, such as the frequency of all items, we can further tighten constraints and efficiently filter candidate patterns based on the new batch.

Finally, we can study *distributed* algorithms for cohesive pattern mining on different servers for handling Big Data sequences. Moens et al. (2013) have proposed to scale frequent pattern mining using *MapReduce*. For cohesive pattern mining, we could process different branches of the search space in parallel. However, further study is required for balancing the search tree and ways to minimise inter-server communication costs.

Next-generation Interestingness Measures

In our qualitative experiments, we found that different interesting measures have different biases towards patterns. FCI$_{seq}$ and QCSP directly mine cohesive patterns and find less frequent cohesive patterns other methods fail to discover. SKOPUS ranks interesting, shorter and more frequent patterns using leverage. CMW also finds longer patterns and was also able to find some patterns with high locality that were missed by FCI$_{seq}$. Finally, GOKRIMP finds a pattern set that best compresses the data. Arguably, the perfect algorithm would find the *union of patterns* discovered with different interestingness measures.

One area of attention for a next-generation of algorithms is balancing *redundancy* in the output pattern set. If a very long pattern is fully cohesive, FCI$_{seq}$ and QCSP, in practice, will enumerate most subpatterns. Both SKOPUS and CMW report many highly overlapping patterns,

i.e., with patterns such as (*paper*, *show*), (*paper*, *result*), etc. for SKOPUS and superpatterns of (*stock*, *market*, *hit*, *high*) for CMW. In contrast, GOKRIMP does not report any patterns sharing items. While this produces satisfactory results on textual datasets, this seems a somewhat restrictive constraint in many domains, especially when the vocabulary is limited, such as DNA sequences. For FCI_{seq} and QCSP a possible solution would be to define *closed* or *maximal* cohesive patterns. We argue that, in general, each method would benefit from taking measures that penalise, to a degree, too many overlapping patterns.

Related to the previous issue, is that all methods ignore, to some extent, the temporal context in very long sequences can miss *locally* interesting patterns. CMW, SKOPUS and GOKRIMP ignore temporal context entirely by relying on a smaller fixed sliding window. FCI_{seq} and QCSP rely on minimal windows, that can span the entire sequence, but in essence we also count occurrences and ignore the actual timestamp of each occurrence. For example, both (*Captain*) and (*Captain*, *Ahab*) occur frequently in Moby Dick. An interesting local pattern is (*Captain*, *Peleg*) which occurs 30 times, but only in chapter 16. A remaining challenge is thus to efficiently measure cohesion local to a larger subsequence of a very long sequence, i.e., a chapter-specific interesting pattern. A possible solution that is available in the open-source code of FCI_{seq} is to partition a single long sequence into medium long sequences for mining. Then we can mine each medium long sequence using any of the techniques above and aggregate these pattern sets. Unfortunately, we would then miss interesting patterns scattered over the entire sequence. Further study is required to find a non-redundant set of patterns that are both locally and globally interesting.

Sequence Classification

Zhou et al. (2016) proposed mining cohesive patterns and association rules for *sequence classification*. They first discover interesting patterns, with interestingness defined as the product of cohesion and frequency, then mine confident associations rules and finally build a *rule-based classifier*. They also studied the usage of patterns as *embedding*, that is, using frequent and cohesive pattern occurrences as a feature vector and applying various *machine learning* algorithms.

It would be of interest to investigate if the work of Zhou et al. (2016) for classification of sequences can be improved. For improving *runtime efficiency*, we remark that we have provided two efficient algorithms that directly mine the most cohesive patterns. Additionally, we also presented a very efficient algorithm for mining cohesion-based rules. In contrast, Zhou et al. first mine all frequent patterns and filters patterns on their cohesion-based interestingness measure during post-processing, which is *less efficient*.

For improving *accuracy*, we first remark that our definition of rule-confidence, based on the extended average minimal window size, is different from the traditional frequency-based definition. Second, we remark that since quantile-based cohesion is more robust to outliers, it is possible to define a more *robust definition of rule confidence* based on quantile-based cohesion of the antecedent and consequent. For example, given a rule $a \Rightarrow b$, with minimal windows sizes $\{2,2,2,2,100\}$ of (a,b) for each instance of a, the confidence is quite low because of the single *outlier*. Intuitively, the confidence should be 0.8, which would be the case with confidence based on quantile-based cohesion. Additionally, we would like to study an embedding based on quantile-based cohesive patterns, investigate discriminative patterns and combine it with machine learning. Remark that we also proposed (frequent) pattern-based embeddings in the context of anomaly detection (Feremans et al. 2019a), which is also relevant in this context. Therefore, we believe that integrating our research with the rule-based classifier proposed by Zhou et al. (2016) could result in further improvements.

Mining Complex Patterns on Complex Event Sequences

In Chapters 2 and 3 we have proposed relatively simple definitions of temporal sequences and patterns. It is not hard to find examples of real-world datasets that are more complex.

For instance, a journal publishes a new volume each year, and each volume consists of a series of abstracts. A new problem setting is to mine patterns that summarise all abstracts over all years. Here, we should consider the order between words in each abstract and the order of each volume. That is, we could represent the dataset as a sequence of sequential databases. For example, we could report interesting patterns trending each decade, such as "pattern mining" that was trending in 2000-2009, and "deep-learning" in 2009-2020, or temporal chains between patterns such as "cohesive pattern mining" is precipitated by "frequent pattern mining". In this setting, Shahaf and Guestrin (2010) have proposed an algorithm for extracting useful knowledge from large datasets. We believe that it would be original to start from cohesive patterns and study interesting non-redundant definitions of patterns and pattern links in this context.

A second problem setting is the study of interesting patterns in multiple long sequences with *mixed-type* attributes. This type of sequential data occurs naturally, for instance, in wind turbine datasets. Here, also *contextual* attributes and *different levels of temporal granularity* are essential. For example, a hot day in the winter is unusual. We proposed a framework for pattern mining and anomaly detection (Feremans et al. 2019b). In this framework, we consider patterns spanning multiple sequences, i.e., finding patterns such as {*low_power_day*, *high_windspeed_hour*, *stop_turbine*}. However, *reducing* all data to a single discrete event sequence might lead to suboptimal performance for anomaly detection or sequence classification. Therefore, we suggest further study of pattern mining algorithms that can process multiple time series and event logs directly. That is mining complex patterns - representing multi-sequence and multi-granular events - that are non-redundant and interesting without ad hoc preprocessing.

Combining Deep-learning and Word Embeddings with Mining Interesting Patterns

Very recently, Li et al. (2017) proposed a method that mines discriminative patterns from a huge number of image patches that outperforms state-of-the-art methods on the task of mid-level visual element discovery. They first train a *convolutional neural network* and then create items based on the activation of each image patch in the final layers of the neural network and learn association rules. We are also interested in extending this idea to other applications, such as sequence classification, and in combining deep-learning with mining interesting patterns. For example, we could train a convolutional neural network on labelled sequences. After training, activation weights in the final layers of this neural network will be discriminative towards each label. Next, we can transform the raw sequences to a sequence of activation weights and mine higher-order cohesive itemsets and association rules. Thus, further study might yield advances in accuracy and explainable sequence, or time series, classification.

Likewise, for finding patterns in text data, we can study adopting more advanced feature extraction than transforming words to lower case and stemming. It could be interesting to use feature extraction algorithms for natural language processing, such as word2vec (Mikolov et al. 2013). For example, we can replace each word with the highest value in the embedding space (or top-k highest values), and then mine cohesive patterns after transforming each word in the sequence to a latent feature. For example, assuming both "King" and "Queen" have a latent feature, roughly corresponding with "royalty", we can mine sequential patterns, where "royalty" co-occurs with another latent feature and discover more semantically interesting patterns.

5.2.2 Improving Extreme Multi-label Classification

Extending Feature-based Nearest Neighbourhood

In Chapter 4 we have adapted the item-based collaborative filtering technique of Sarwar et al. (2001) for multi-label classification. By itself, feature-based kNN is already quite useful: it outperforms IBLR and is comparable with ML-KNN. From a performance perspective, feature-based kNN requires no nearest neighbour search at test time and can make highly explainable predictions within one millisecond on extreme datasets. More recently, Ning and Karypis (2011) proposed SLIM, an extension to item-based collaborative filtering that learns a sparse similarity matrix by combining L2 and L1 regularisation. SLIM achieves significant improvements both in run time performance and recommendation quality over the best existing methods. Therefore, we consider adapting SLIM for extreme multi-label classification.

Boosting the Performance of LCIF

LCIF is a straightforward method that is extremely efficient at test and, especially, *training* time. Recently many extreme multi-label methods have been proposed that compete on benchmark datasets on precision@k (Bhatia et al. 2016). Some methods reduce the *dimensionality of the label space*, while others build a hierarchical *ensemble of tree-based models*. We compared LCIF with FASTXML (Prabhu and Varma 2014) , but not with ANNEXML (Tagami 2017), that uses an approximate nearest neighbour search, SLEEC (Bhatia et al. 2015), that computes sparse local embeddings, or DISMEC (Babbar and Schölkopf 2017), which is an efficient parallel algorithm for learning one-versus-rest linear classifiers.

We argue that in classification benchmarks, the best results are reported by ensemble methods that combine many models, possibly using different algorithms and feature representations. Therefore, comparing LCIF as-is, with the aforementioned ensemble-based methods would not result in superior results. However, we see potential to improve results on precision@k by creating ensembles using *boosting* or *stacking* and adopting *embeddings*. For text categorisation, we consider that the results of multiple runs of LCIF using different feature extraction algorithms from natural language processing, such as word or sentence embeddings learned using neural networks (Mikolov et al. 2013; Devlin et al. 2018) and different similarity measures, such as BM25, could improve results. In general, we can compute predictions on different - possibly randomised - subsets of features, different samples of datasets, using different settings for the hyperparameters, and then use voting or machine learning, to make final predictions based on different runs of LCIF. Finally, we can also study creating an ensemble that combines predictions from LCIF with any of the aforementioned techniques.

APPENDIX A

Additional Material for FCI$_{seq}$

A.1 Computing Minimal Windows for Sequential Patterns

For mining sequential patterns we must adapt SUM_MIN_WINS. For example, given a sequence $a_1\ b_2\ a_3$ we need to take into account that both sequential patterns (a_1, b_2) and (b_2, a_3) occur at time stamp 2. Algorithm SUM_MIN_WINS$_{seq}$ is an adaptation of the original SUM_MIN_WINS and is shown in Algorithm A.1. Remark that we omit unmodified code for brevity. The main difference compared to the original algorithm, is that we now return a list of *minimal windows*, where for each occurrence t we maintain possibly *multiple instances* of the minimal windows at each occurrence. The list of *final windows* is first initialised (line 2) and returned (line 21) together with the sum of minimal windows. As a result, within the inner loop, we now not only update the minimal width of active windows but maintain, where necessary, multiple instances of windows with the same minimal length (lines 7 to 13). If a new window is found with a lower minimal width, we *reset* the instances of active windows (line 10). If a new window is found with the same minimal width, we *add* that window to the instances (line 12). For example, given sequence $\ldots bxaba\ldots$, for itemset $\{a, b\}$, we will find one minimal window for the two occurrences of a, and the first occurrence of b, but we will find two minimal windows for the second occurrence of b. Note that, compared to Algorithm 2.3, we omitted the condition of removing windows where $window.width == |X|$ from the list of active windows (line 13), because we now want to enumerate all instances of the minimal window.

Algorithm A.1: SUM_MIN_WINS$_{seq}(\mathcal{S}, X, Y)$ Compute the sum and the set of minimal windows of each occurrence of X in $\mathcal{S}$

```
   Input : An event sequence S, candidate itemset X, set of items Y
   Result: Sum and set of minimal windows (possibly with multiple instances)
   /* Initialisation as in SUM_MIN_WINS */
 1 ...;                               // Unmodified code is omitted for brevity
 2 final_windows ← ∅;
 3 for index in N(X) do
       /* Abandon if running sum too high to be cohesive */
 4     ...;
       /* If new minimum in current window */
 5     if minpos ≠ −∞ and minpos > prev_min then
           /* Inner loop over previous non-final windows */
 6         for window in active_windows do
               /* Manage multiple minimal window instances */
 7             newwidth ← maxpos − min(minpos, window.pos) + 1;
 8             if newwidth < window.width then
 9                 window.width ← newwidth;
10                 window.instances ← { INSTANCE(minpos, maxpos) };
11             if newwidth = window.width then
12                 window.instances ← window.instances ∪
                    { INSTANCE(minpos, maxpos) };
               /* Check if minimal window is final */
13             if window.pos < minpos or
                window.width < (maxpos − window.pos + 1) then
14                 active_windows ← active_windows \ {window};
15                 final_windows ← final_windows ∪ {window};
16                 smw ← smw + window.width;
17     active_windows ← active_windows ∪ { WINDOW(current_pos,
        maxpos − minpos + 1, INSTANCE(minpos, maxpos))};
18     prev_min ← minpos;
19 smw ← smw + ∑_{window ∈ active_windows} window.width;
20 final_windows ← final_windows ∪ active_windows;
21 return ⟨smw, final_windows⟩;
```

A.2 Compute Support of an Episode

For discovering dominant episodes, we must compute the support for each candidate episode. Algorithm COMPUTE_SUPPORT_EPISODE is shown in Algorithm A.2. We first loop over each occurrence (or minimal window) and then over each minimal window instance. As discussed in section 2.4, it is important to loop over multiple minimal window instances since there can be more than one minimal window instances for a single occurrence of an item. For example, sequence $a_1\ b_2\ a_3$ has two minimal window instances for itemset $\{a, b\}$ at timestamp 2, namely $a_1\ b_2$ and $b_2\ a_3$. We optimise our computation in two cases: When any edge is not covered by the current instance (line 10) we do not check remaining edges for the current instance, and when an instance is covered we do not check other instances (line 11).

Algorithm A.2: COMPUTE_SUPPORT_EPISODE($\mathcal{S}, X, G$) Compute the support of a candidate episode in a sequence based on a cohesive itemset X

```
  Input : An event sequence S, frequent cohesive itemset X, episode G where V(G) = X
  Result: Number of occurrences of G, that is |occ_po(G)|
1 ⟨smw, min_windows_seq⟩ ← SUM_MIN_WINS_seq(S, X, ∅);
2 support_G ← 0;
3 for win in min_windows_seq do
      /* Check partial order holds in minimal window */
4     covers_window ← false;
5     for win_ins in win do
6         covers_instance ← true;
7         pos ← {⟨i, t⟩| i ∈ X ∧ win_ins.min ≤ t ≤ win_ins.max};
8         for ⟨i1, i2⟩ ∈ E(G) do
9             covers_edge ← ∃⟨i1, t1⟩, ⟨i2, t2⟩ ∈ pos : t1 < t2;
10            if not covers_edge then
                  covers_instance ← false; break ;
11        if covers_instance then
              covers_window ← true; break ;
12    if covers_window then
          support_G ← support_G + 1 ;
13 return support_G;
```

A.3 Top 25 Patterns Discovered by FCI$_{seq}$ on Species

Tables A.1, A.2 and A.3 show the top 25 itemsets, sequential patterns and association rules discovered by FCI$_{seq}$ and the state-of-the-art methods on *Species*. Patterns for FCI$_{seq}$ in bold are not discovered by any other state-of-the-art method in the top 1000; likewise, patterns in bold for other methods are not discovered by FCI$_{seq}$ in the top 1000. As discussed in Section 2.8, FCI$_{seq}$ produced fewer than 25 *sequential patterns* due to the usage of the minimal occurrence ratio threshold. A lower threshold would naturally result in more patterns. Still, we argue that these patterns are better omitted from the output, since they are not representative for the occurrences of the underlying itemset.

Table A.1: Top 25 *itemsets* discovered by FCI_{seq} and the other methods on *Species*

FCI_{seq}	WINEPI	LAXMAN	MARBLES_{W}	CMW
del, fuego, tierra	natur, select	natur, select	natur, select	**absenc, island, mammal, ocean, terrestri**
facit, natura, saltum	**speci, varieti**	form, speci	**speci, varieti**	**altern, glacial, north, period, south**
del, fuego	form, speci	**speci, varieti**	distinct, speci	**bat, island, mammal, ocean, speci, terrestri**
del, tierra	natur, speci	natur, speci	form, speci	**bat, island, mammal, ocean, terrestri**
facit, saltum	distinct, speci	distinct, speci	condit, life	**cross, fertil, hybrid, mongrel, offspr, varieti**
facit, natura	gener, speci	gener, speci	natur, speci	**differ, endow, incident, special, steril**
fuego, tierra	**differ, speci**	**differ, speci**	**anim, plant**	**ag, earli, inherit, success, superven, variat**
natura, saltum	case, speci	case, speci	**genu, speci**	**glacial, northern, period, southern, temper**
ithomia, leptali	condit, life	case, natur	**differ, speci**	**ag, earli, inherit, period, superven, variat**
leptali, mimick	**genu, speci**	**natur, organ**	**be, organ**	**inhabit, island, mainland, nearest, relat**
ithomia, leptali, mimick	**anim, plant**	**select, speci**	**group, speci**	**fertil, mongrel, offspr, univers, varieti**
ithomia, mimick	**group, speci**	**group, speci**	gener, speci	**cross, fertil, mongrel, offspr, varieti**
hexagon, sphere	**number, speci**	**number, speci**	**genera, speci**	**mountain, northern, southern, temper**
forcep, urchin	case, natur	**genera, speci**	**cross, speci**	**ag, earli, inherit, superven, variat**
rufescen, sanguinea	**genera, speci**	**charact, speci**	**individu, speci**	**absenc, island, ocean, terrestri**
pyramid, rhomb	**cross, speci**	**plant, speci**	case, speci	**crop, fantail, pouter, tail**
natur, select	**be, organ**	**form, varieti**	**alli, speci**	**cross, differ, incident, system, unknown**
sur, tom	**close, speci**	**anim, plant**	**number, speci**	**exist, geometr, increas, ratio, struggl**
natur, speci	**charact, speci**	form, natur	**form, life**	**absenc, mammal, ocean, terrestri**
prism, pyramid, rhomb	**descend, speci**	condit, life	**descend, speci**	**connect, exist, intermedi, lesser, number, varieti**
busk, chela	**individu, speci**	**natur, variat**	**alli, close**	**fittest, man, natur, select, surviv**
form, speci	**form, varieti**	**organ, speci**	case, natur	**absenc, island, mammal, terrestri**
gener, speci	**natur, variat**	**individu, speci**	**exist, speci**	**altern, glacial, north, period**
matthew, vol	**natur, organ**	natur, select, speci	**close, speci**	**cross, differ, incident, reproduct, system**
saint, sur	**select, speci**	**anim, speci**	**produc, speci**	**endow, incident, special, steril**

Table A.2: Top 25 *sequential patterns* discovered by FCI_{seq} and other methods on *Species*.

FCI_{seq}	WINEPI	LAXMAN	MARBLES_{W}	CMW
tierra, del, fuego	natur, select	natur, select	natur, select	**struggl, exist, geometr, ratio, increas**
natura, facit, saltum	**varieti, speci**	**varieti, speci**	**distinct, speci**	**variat, superven, earli, ag, inherit**
del, fuego	**speci, varieti**	**speci, form**	**varieti, speci**	**form, life, chang, simultan, world**
tierra, del	**distinct, speci**	**speci, varieti**	**condit, life**	**variat, superven, earli, inherit, ag**
facit, saltum	**speci, form**	**form, speci**	**speci, varieti**	**steril, speci, cross, hybrid, offspr**
natura, facit	**form, speci**	**speci, natur**	**organ, be**	**wide, diffus, speci, larger, genera, vari**
tierra, fuego	**condit, life**	**natur, speci**	**speci, genu**	**success, variat, superven, earli, inherit**
natura, saltum	**speci, natur**	**distinct, speci**	**speci, form**	**inhabit, island, nearest, mainland**
natur, select	**speci, genu**	**speci, gener**	**form, speci**	**chapter, geolog, success, organ, be**
vol, matthew	**natur, speci**	**case, speci**	**individu, speci**	**varieti, exist, lesser, number, intermedi**
avicularia, vibracula	**organ, be**	**gener, speci**	**close, alli**	**glacial, period, north, south**
eject, foster, brother	**speci, gener**	**speci, differ**	**speci, natur**	**incident, differ, reproduct, system**
eject, foster	**differ, speci**	**differ, speci**	**anim, plant**	**natur, system, genealog, arrang**
oviger, frena	**speci, differ**	**speci, case**	**group, speci**	tierra, del, fuego
movabl, zooid	**case, speci**	**condit, life**	**speci, genera**	**seiz, place, economi, natur**
inter, se	**speci, genera**	**select, natur**	**alli, speci**	**superven, earli, ag, inherit**
sphere, prism	**gener, speci**	**speci, distinct**	**differ, speci**	natura, facit, saltum
foster, brother	**individu, speci**	**number, speci**	**form, life**	**instinct, slave, make, ant**
sown, mix	**group, speci**	**speci, genu**	**speci, gener**	**varieti, exist, lesser, number, connect**
sphere, hexagon, prism, rhombic	**number, speci**	**speci, genera**	**natur, speci**	**direct, action, extern, condit**
hexagon, prism	**speci, distinct**	**group, speci**	**speci, differ**	**ocean, island, terrestri, mammal**
	anim, plant	**speci, group**	**number, speci**	**natur, select, extinct, diverg, charact**
	alli, speci	**case, natur**	**speci, group**	**exist, geometr, ratio, increas**
	speci, group	**charact, speci**	**fresh, water**	**revers, long, lost, charact**
	speci, case	**individu, speci**	**case, speci**	**form, naturalist, rank, distinct, speci**

Table A.3: Top 25 *rules* discovered by FCI_{seq} and other methods on *Species*

FCI_{seq}	WINEPI	$\text{MARBLES}_{\text{M}}$	$\text{MARBLES}_{\text{W}}$
fuego, tierra → del	**divis, kingdom → anim**	**hive → bee**	**divis, kingdom → anim**
del, fuego → tierra	**averag, genera → speci**	**mivart → mr**	fuego, tierra → del
del, tierra → fuego	**cuckoo, lai, nest → egg**	**case, select, structur → natur**	**independ, ordinari → view**
facit, natura → saltum	**varieti, zone → inter-medi**	**candol → de**	natura, saltum → facit
natura, saltum → facit	**genera, present, varieti → speci**	**case, organ, select → natur**	**inherit, superven → earli**
facit, saltum → natura	**cape, hope → good**	**distinct, rank, varieti → speci**	**accumul, act, natur → select**
fuego → del	**accumul, act, natur → select**	**breed, rock → pigeon**	**rang, vari → speci**
del → fuego	**inherit, superven → earli**	**diverg, select → natur**	**economi, seiz → place**
tierra → del	fuego, tierra → del	**wallac → mr**	**ag, inherit, superven → earli**
del → tierra	**differ, genera, varieti → speci**	**humbl → bee**	**exist, select, theori → natur**
fuego → del, tierra	**genu, greater → speci**	**function, select → natur**	**inherit, superven, variat → earli**
tierra → del, fuego	**select, structur, theori → natur**	**independ, select → natur**	**act, natur, sole → select**
del → fuego, tierra	**genera, smaller, varieti → speci**	**life, physic → condit**	**newli, varieti → form**
facit → saltum	**independ, ordinari → view**	**charact, secondari → sexual**	**exist, varieti, zone → intermedi**
saltum → facit	**charact, secondari, speci → sexual**	**favour, select, speci → natur**	**ask, distinct → speci**
natura → facit	**creat, independ, view → speci**	**select, speci, theori → natur**	**ag, inherit, superven, variat → earli**
facit → natura	**inherit, superven, variat → earli**	**charact, diverg, select → natur**	**bottom, rest → side**
natura → facit, saltum	**genera, larger, number → speci**	**malai → archipelago**	**end, mean → gain**
facit → natura, saltum	**exist, select, theori → natur**	**genu, manner → speci**	**rang, vari, wide → speci**
saltum → facit, natura	**action, diverg, natur → select**	**genu, produc → speci**	**incident, reproduct → differ**
prism → hexagon	**action, diverg, select → natur**	**fittest → surviv**	**english, face → short**
fuego → tierra	natura, saltum → facit	**incipi, varieti → speci**	**economi, natur, seiz → place**
tierra → fuego	**ag, inherit, superven → earli**	**cell, hive → bee**	**larger, relat → genera**
natura → saltum	**rang, vari → speci**	**genera, larger, varieti → speci**	**manner, mivart → mr**
saltum → natura	**bird, cuckoo, lai, nest → egg**	**fritz → muller**	**life, organ, physic → condit**

A.4 Top 25 Patterns Discovered by FCI_{seq} on Trump

Tables A.4, A.5 and A.6 show the top 25 patterns discovered on *Trump*. For more details concerning reporting see Section A.3.

Table A.4: Top 25 *itemsets* discovered by FCI_{seq} and other methods on *Trump*

FCI_{seq}	WINEPI	LAXMAN	MARBLES_W	CMW
puerto, rico	fake, new	fake, new	fake, new	**ab, japan, minist, prime**
hunt, witch	cut, tax	cut, tax	cut, tax	**high, hit, market, stock, time**
harbor, pearl	america, great	america, great	america, great	**abc, cnn, fake, nbc, new**
lago, mar	great, make	great, peopl	america, make	**alabama, big, luther, strang, vote**
arabia, saudi	america, make	great, make	great, make	**lowest, market, stock, unemploy, year**
jong, kim	great, peopl	great, job	**state, unit**	**base, immigr, merit, system**
davo, switzerland	america, great, make	countri, great	**great, honor**	**high, hit, job, market, stock**
davo, wef	great, job	great, todai	korea, north	**abc, cb, cnn, fake, new**
davo, switzerland, wef	**great, honor**	america, make	great, job	**greatest, histori, hunt, witch**
switzerland, wef	countri, great	**great, tax**	america, great, make	**alabama, great, luther, state, strang**
unga, usaatunga	fake, media	great, state	great, peopl	**high, hit, market, stock**
prstrong, ricar-dorossello	great, state	big, great	**media, new**	**donald, presid, proclaim, trump**
rex, tillerson	**state, unit**	great, new	**hard, work**	**korea, moon, presid, south**
fake, new	**great, tax**	america, great, make	fake, media	**bail, compani, democrat, insur**
minist, prime	great, todai	**great, honor**	market, stock	**fail, fake, media, new, nytim**
christma, merri	**great, work**	**great, work**	great, state	**high, market, stock, un-employ**
korea, north	**media, new**	fake, media	**hous, white**	**abc, cnn, nbc, new**
cut, tax	great, new	**cut, great**	**republican, senat**	**high, job, market, stock, time**
america, make	**dai, great**	**great, presid**	countri, great	**alabama, luther, strang, vote**
chain, migrat	big, great	**american, great**	**dai, great**	**high, market, record, stock, time**
great, peopl	**hard, work**	great, year	great, new	**big, luther, senat, strang**
market, stock	korea, north	**dai, great**	**great, meet**	**court, state, suprem, unit**
america, great	fake, media, new	**great, republican**	great, todai	**cnn, fail, fake, new, nytim**
fake, great, new	**cut, great**	**cut, great, tax**	**cut, reform**	**high, market, stock, time**
america, great, make	**american, great**	**fake, great**	minist, prime	**alabama, great, luther, senat, strang**

Table A.5: Top 25 *sequential patterns* discovered by FCI_{seq} and other methods on *Trump*

FCI_{seq}	WINEPI	LAXMAN	MARBLES_{W}	CMW
puerto, rico	fake, new	fake, new	fake, new	**stock, market, hit, time, high**
witch, hunt	tax, cut	tax, cut	tax, cut	**job, stock, market, time, high**
pearl, harbor	**america, great**	**america, great**	**america, great**	**greatest, witch, hunt, histori**
mar, lago	**make, america**	**make, great**	**make, america**	**stock, market, hit, high**
saudi, arabia	**make, great**	**make, america**	**unit, state**	**presid, moon, south, korea**
kim, jong	**make, america, great**	**great, peopl**	**make, great**	**presid, donald, trump, proclaim**
davo, switzerland	**unit, state**	**make, america, great**	**great, honor**	**fake, new, fail, nytim, cnn**
switzerland, wef	**great, honor**	**great, job**	north, korea	**market, hit, time, high**
usaatunga, unga	fake, media	**great, honor**	**make, america, great**	**stock, market, time, high**
ricardorossello, prstrong	**great, job**	**peopl, great**	**new, media**	**stock, market, hit, time**
rex, tillerson	**great, peopl**	**great, state**	stock, market	**stock, hit, time, high**
fake, new	**new, media**	fake, media	fake, media	**massiv, tax, cut, reform**
prime, minist	north, korea	**great, countri**	**white, hous**	**healthcar, tax, cut, reform**
merri, christma	stock, market	**great, tax**	**work, hard**	**fake, new, cnn, abc**
north, korea	**great, state**	**unit, state**	**great, job**	**republican, senat, work, hard**
tax, cut	**fake, new, media**	**tax, great**	**great, peopl**	**radic, islam, terror**
chain, migrat	**white, hous**	**great, todai**	**great, state**	**fake, new, cnn, nbc**
stock, market	**work, hard**	**great, work**	**republican, senat**	**new, media, fail, nytim**
luther, strang	**great, countri**	**countri, great**	prime, minist	**make, america, great, fake, new**
fake, media	**tax, reform**	north, korea	**tax, reform**	**greatest, witch, hunt**
berni, sander	**peopl, great**	**great, america**	**fake, new, media**	**joint, press, confer**
radic, islam	**republican, senat**	**great, american**	**crook, hillari**	**prime, minist, ab**
	great, tax	**great, presid**	**cut, reform**	**behalf, flotu, melania**
	great, american	**new, great**	**great, countri**	**stock, market, hit, high, great**
	great, meet	**new, media**	**men, women**	**m, gang, member**

Table A.6: Top 25 *rules* discovered by FCI_{seq} and other methods on *Trump*

FCI_{seq}	WINEPI	$\text{MARBLES}_{\text{M}}$	$\text{MARBLES}_{\text{W}}$
puerto → rico	**hit, stock → market**	**cut, reform → tax**	**honor, minist → prime**
rico → puerto	**high, hit, stock → market**	puerto → rico	**confer, joint → press**
hunt → witch	**bill, reform, tax → cut**	rico → puerto	**immigr, merit → base**
witch → hunt	**biggest, cut, histori → tax**	witch → hunt	**greatest, hunt → witch**
migrat → chain	**ab, prime → minist**	hunt → witch	**donald, proclaim → trump**
pearl → harbor	**abc, fake → new**	**great, white → hous**	**hit, record, stock → market**
harbor → pearl	**honor, minist → prime**	**high, market → stock**	**immigr, merit, system → base**
merri → christma	**behalf, melania → flotu**	**cut, dem → tax**	**law, offic → enforc**
saudi → arabia	**hit, stock, time → market**	**cut, pass → tax**	**liyuan, madam → peng**
arabia → saudi	**massiv, reform, tax → cut**	**biggest, cut → tax**	**donald, presid, proclaim → trump**
lago → mar	**abc, cnn, fake → new**	**market, record → stock**	**chain, visa → migrat**
mar → lago	**confer, joint → press**	**alabama, strang → luther**	**great, honor, minist → prime**
kim → jong	**immigr, merit → base**	**suprem → court**	**high, hit, record, stock → market**
jong → kim	**abc, fake, nbc → new**	**great, prime → minist**	**cut, massiv, work → tax**
sander → berni	**high, hit, stock, time → market**	**great, minist → prime**	**great, high, stock → market**
usaatunga → unga	**biggest, reform → tax**	**great, puerto → rico**	**histori, witch → hunt**
rex → tillerson	**biggest, reform → cut**	**hous, tax → cut**	**jame, leak → comei**
switzerland → davo	**biggest, reform → cut, tax**	**cut, hous → tax**	**bill, massiv, tax → cut**
wef → davo	**hit, record, stock → market**	**fake, nytim → new**	**famili, thought → prayer**
davo → switzerland	**biggest, reform, tax → cut**	**big, cut, great → tax**	**men, protect → women**
switzerland → davo, wef	**biggest, cut, pass → tax**	**birthdai → happi**	**jone, pelosi, puppet → schumer**
switzerland, wef → davo	**biggest, cut, reform → tax**	**xi → china**	**great, job, market → stock**
wef → davo, switzerland	**immigr, merit, system → base**	**premium → obamacar**	**american, cut, massiv → tax**
switzerland → wef	**alabama, big, strang → luther**	**great, strang → luther**	**high, stock, unemploy → market**
davo, switzerland → wef	**budget, cut → tax**	**north, presid → korea**	**anthem, great, stand → nation**

APPENDIX B

Additional Material for QCSP

B.1 Weighted Quantile-based Cohesion

While quantile-based cohesion is more robust to outliers than cohesion, it does not take the size of the minimal windows into account. In response to this observation, we propose the following variant, namely *weighted quantile-based cohesion*[1]:

Definition B.1 (Weighted quantile-based cohesion)**.**

$$C_{quan_weighted}(X_s) = \frac{\sum_{t \in cover(X_s) \wedge W_t(X_s) < \alpha \cdot |X_s|} |X_s| \cdot W_t(X_s)^{-1}}{support(X_s)}$$

Since $C_{quan_weighted}(X_s) \leq C_{quan}(X_s)$, $C_{quan}(X_s)$ is an upper bound for weighted quantile-based cohesion and we can re-use the QCSP algorithm and re-rank sequential patterns on $C_{quan_weighted}$ during post-processing. We have not performed experiments on re-ranking on this variant. For some scenario's (see the example below) it makes sense to rank patterns on weighted quantile-based cohesion. However, it could be that $C_{quan}(X_{s_1}) > C_{quan}(X_{s_2})$, but $C_{quan_weighted}(X_{s_1}) < C_{quan_weighted}(X_{s_2})$ and re-ranking might perform worse, e.g., if X_{s_1} have a minimal window of size 4 for 10 out of 20 occurrences and X_{s_2} has a minimal window of size 2 for 10 out of 30 occurrences, which is preferred? In this scenario, a simple solution is to first sort on $C_{quan}(X_s)$ and then break ties using $C_{quan_weighted}(X_s)$. Alternative, the final ranking can be chosen specifically to each application, i.e., giving more weight to the ratio between cohesive versus non-cohesive occurrences, more weight the absolute number of cohesive occurrences or more weight relative to the size of minimal windows. Therefore, further experimental validation is required to consider this variant.

Example B.1. For example, assume we have two sequential patterns $X_{s_1} = (a, b)$ and $X_{s_2} = (b, c)$ and the minimal window sizes are $\{2, 2, 2, 2, 100, 100, 100, 100\}$ for X_{s_1} and $\{4, 4, 4, 4, 100, 100, 100, 100\}$ for X_{s_2}. In this scenario, 4 occurrences of (a, b) are next to each other, while for (c, d) there are two items, or gaps, between c and d in 4 occurrences. However, quantile-based cohesion ignores the actual size of minimal windows, that is, w.r.t. $\alpha = 3$, $C_{quan}(X_{s_1}) = C_{quan}(X_{s_2}) = 0.5$. In contrast, weighted quantile-based cohesion would rank X_{s_1} first, that is, $C_{quan_weighted}(X_{s_1}) = \frac{4 \cdot 2/2}{8} = 0.5 > C_{quan_weighted}(X_{s_2}) = \frac{4 \cdot 2/4}{8} = 0.25$ □

[1] This variant was not discussed in the original publication (Feremans et al. 2018).

B.2 Top 20 Sequential Patterns Discovered by QCSP

Tables B.1, B.2 and B.3 show the top 20 patterns for the text datasets. Patterns not found either exactly or as sub-patterns in the top 250 of other methods are shown in bold. We see that QCSP finds many interesting patterns that other methods fail to rank highly. On the other hand, the patterns SKOPUS ranks highly and other methods do not are typically combinations of very frequent items. GOKRIMP, in general, produces few patterns and misses out on many interesting patterns altogether.

Table B.1: Top 20 sequential patterns discovered by QCSP and other methods on *Moby*

	mobi, dick	**um, um**	sperm, whale	**ginger, ginger**
	mrs, hussey	**seven, seventi**	town, ho	**ha, ha**
QCSP	**ii, octavo**	cape, horn	**ii, iii**	**jack, knife**
	crow, nest	o, clock	dough, boy	mast, head
	iii, duodecimo	**hither, thither**	**beef, bread**	**inclin, plane**
	sperm, whale	mobi, dick	said, i	d, ye
	white, whale	captain, ahab	**whale, him**	**ye, see**
SKOPUS	**though, yet**	right, whale	**look, like**	quarter, deck
	old, man	**whale, head**	cri, ahab	**ahab, him**
	mast, head	**whale, ship**	**one, side**	**now, whale**
	mobi, dick	captain, ahab	cri, ahab	
	sperm, whale	said, i		
GOKRIMP	mast, head	right, whale		
	white, whale	quarter, deck		
	old, man	d, ye		

Table B.2: Top 20 sequential patterns discovered by QCSP and other methods on *JMLR*

	mont, carlo	reproduc, hilbert	support, machin	**bayesian, network**
	nearest, neighbor	real, world	state, art	high, dimens
QCSP	support, vector	**belief, propag**	vector, machin	**collabor, filter**
	http, www	support, vector, machin	**messag, pass**	**naiv, bay**
	cross, valid	**plai, role**	data, set	learn, algorithm
	paper, show	**base, result**	**paper, propo**	**paper, set**
	paper, result	**paper, method**	vector, machin	support, machin
SKOPUS	**paper, experi**	**learn, result**	**paper, new**	**learn, data**
	paper, algorithm	**problem, experi**	**algorithm, result**	**problem, result**
	support, vector	**learn, experi**	**paper, base**	**algorithm, experi**
	support, vector, machin	high, dimens	well, known	
	real, world	neural, network	hilbert, space	
GOKRIMP	machin, learn	compon, analysi	experi, result	
	state, art	**supervi, learn**		
	reproduc, hilbert, space	support, vector		

Table B.3: Top 20 sequential patterns discovered by QCSP and other methods on *Trump*

QCSP	puerto, rico	**goofi, elizabeth, warren**	**luther, strang**	https, co
	witch, hunt	prime, minist	**stock, market**	fake, news
	elizabeth, warren	goofi, warren	self, fund	**novemb, 8th**
	las, vega	radic, islam	suprem, court	white, hous
	goofi, elizabeth	e, mail	mitt, romney	**common, core**
SKOPUS	make, america	america, again	thank, trump2016	crook, clinton
	make, great	great, again	**thank, makeamericagreatagain**	donald, trump
	crook, hillari	make, again	fake, news	last, night
	hillari, clinton	thank, you	2016, fals	**thank, co**
	america, great	https, co	ted, cruz	interview, enjoy
GOKRIMP	make, america, great, again	thank, you	trump2016http, co	north, carolina
	https, co	last, night	00, m	north, korea
	crook, hillari	hillari, clinton	new, york	make, america, great
	fake, news	2016, fals	donald, trump	white, hous
	ted, cruz	look, forward	south, carolina	work, hard

APPENDIX C

Additional Material for LCIF

C.1 Second Order Instance-based kNN

A possible disadvantage of the instance-based kNN method proposed in Section 4.3.1 is that *inter-label dependencies* are ignored: the prediction for a given label is obtained independently of the values of other labels. Additionally, in user-based collaborative filtering there is no distinction between features and labels and cosine similarity is defined using all items. This is not possible in the multi-label classification setting since for any test instance x_q, by problem definition, there is no label information. In this section, we propose a variant of instance-based kNN that does regard other labels in the prediction process and uses cosine similarity defined on both features and labels. That is, second order instance-based kNN does a *neighbours of neighbours search* where the similarity between features and labels for training instances is used.

Definition C.1 (Full cosine similarity)**.** *The cosine similarity using both features and labels between two training instances x_i and x_j in $\mathcal{D}$ is defined as*

$$sim_{\text{ALL}}(x_i, x_j) = \frac{x_i \cdot x_j + y_i \cdot y_j}{\sqrt{\sum_{k=1}^{M} x_{i,k}^2 + \sum_{k=1}^{L} y_{i,k}^2} \cdot \sqrt{\sum_{k=1}^{M} x_{j,k}^2 + \sum_{k=1}^{L} y_{j,k}^2}}$$

$$= x_i \cdot x_j + y_i \cdot y_j = \sum_{k=1}^{M} x_{i,k} x_{j,k} + \sum_{k=1}^{L} y_{i,k} y_{j,k}$$

where y_i and y_j are both vectors of size L containing the labels corresponding to instances x_i and x_j and we ensure that all instance vectors (consisting of both feature and label values) are normalised to unit length during preprocessing.

Definition C.2 (Second order label weight)**.** *Given a training instance x_i in $\mathcal{D}$ we define the re-weighted value for label $y_{i,j}$ as*

$$rw(y_{i,j}) = \frac{\sum\limits_{x_k \in \, kNN(x_i)} y_{k,j} \cdot sim_{\text{ALL}}(x_i, x_k)}{\sum\limits_{x_k \in \, kNN(x_i)} sim_{\text{ALL}}(x_i, x_k)}$$

where the k-nearest neighbours are computed using sim_{ALL}. Note that the first neighbour is the instance itself.

Definition C.3 (Second order instance-based confidence score)**.** *To compute the confidence score for instance x_q for (a single) label y_j we define the following function:*

$$\hat{y}^{\text{INS2}}_{q,j} = \frac{\sum_{x_i \in kNN(x_q)} rw(y_{i,j}) \cdot sim_{INS}(x_q, x_i)}{\sum_{x_i \in kNN(x_q)} sim_{INS}(x_q, x_i)}$$

The motivation behind changing each binary label in $\mathcal{D}$ to a re-weighted value is that we use the similarity function sim_{ALL}. By using a measure of similarity within the space of both features and labels, we expect that this weight better reflects local clusters of co-occurring labels. It is somewhat counter-intuitive to change the given binary labels, but this approach has the advantage that we do account for local neighbours of each training instance (like a micro-cluster around each training instance). We remark that the second order approach has some similarity with ML-KNN that also employs a neighbours of neighbours search thereby counting the labels of the neighbours for each training instance to make final predictions (Zhang and Zhou 2007). It is also related to the idea proposed by Breunig et al. (2000) in which they identify local outliers as points that have low local density, that is, where the (relative) distance to its nearest neighbours is high. Here, the re-weighted label values will be higher for labels with a high local density and lower for labels with low local density. Finally we propose a variation of the linear combination, named LCIF2, where we re-use Equation 4.9 but take a linear combination of the second order instance-based confidence score $\hat{y}^{\text{INS2}}_{q,j}$ with the feature-based confidence score $\hat{y}^{\text{FL}}_{q,j}$.

Limitation

A limitation of this variation is that in many benchmark multi-label datasets the *label cardinality is very small*, with on average fewer than 5 labels for each instance. In this case, it does not make sense to apply this variation since the effect on the second order neighbourhood computed using sim_{ALL} versus sim_{INS} will be minor. As a *rule of thumb* we apply it on datasets where the average number of labels is comparable to the average number of features.

Complexity

The main computational cost is that we have to perform a kNN search over all training instances requiring $\mathcal{O}(\frac{1}{2}N^2 \times (M+L))$ steps. We compute this neighbourhood efficiently using an inverted index (for features and labels) and only compute nonzero terms of the full cosine similarity between all training instance pairs. In practice, the number of instances N that actually share a single feature with other training instances is much smaller, and we observe an average runtime of $\mathcal{O}(\frac{1}{2}N \times \tilde{n} \times (\tilde{m} + \tilde{l}))$. Here $\tilde{n}$ is proportional to the average number of candidate instances fetched from the inverted index (sharing at least one 1 feature), and $\tilde{m}$ and $\tilde{l}$ are proportional to the average number of nonzero features and labels.

Classification Performance

We apply the second order instance variation (and the linear combination with feature-based kNN) on datasets where the average number of labels and features is comparable. Given our selection of datasets we find that the *Delicious* dataset has a label cardinality of 19.1 and a feature cardinality of 18.3 (see Table 3.1) and is applicable.

The classification accuracy for the second order instance variation on the *Delicious* dataset is shown in Table C.1. We see that the second order instance variation further improves the

result of the instance-based kNN. We remark that the effect of feature-based kNN in LCIF2 is negligible in this dataset except for Hamming Loss. The second order variation performs better than ML-KNN and BR-SMO on *Delicious* on all metrics except Hamming Loss. We did not include results on IBLR since it timed-out on our test server because it has to learn $L = 983$ different models for *Delicious*. Based on these results we argue that the second order instance-based kNN is a promising variation on datasets where the label cardinality is close to the feature cardinality.

Table C.1: Comparing the accuracy of second order instance-based kNN with instance-based kNN, ML-KNN and BR-SMO on *Delicious*

Metric	INS. kNN	SECOND INS. kNN	FEAT. kNN	LCIF	LCIF2	ML-KNN	BR-SMO
Accuracy ↑	0.227	**0.237**	0.101	0.227	**0.237**	0.193	0.130
Micro F1 ↑	0.362	**0.372**	0.189	0.363	**0.372**	0.322	0.224
Macro F1 ↑	0.182	**0.184**	0.050	**0.184**	**0.184**	0.088	0.098
Hamming Loss ↓	0.024	0.023	0.021	0.021	0.021	0.021	**0.018**

Runtime Performance

Table C.2 shows the runtime of feature-based kNN, instance-based kNN and second order instance variation in seconds. In contrast to Chapter 4 we used the previous implementation available in Java[1]. The difference between the second order instance-based kNN compared to instance-based kNN is clear. However, if we make a distinction between *training time* and *test time* we remark that the additional time required for second order instance-based kNN is only required at training. Finally, we remark that the total runtime of the second order variation is still excellent compared to ML-KNN, IBLR and BR-SMO which are still an order of magnitude slower (or time-out after a full day of computation).

Table C.2: Comparing the runtime of the second order instance-based kNN variation on large datasets

Dataset	FEAT. kNN	INS. kNN	SECOND INS. kNN
Total train and test time for classifier ↓			
Medical	**0.3 s**	0.3 s	0.3 s
Corel5k	**0.4 s**	0.4 s	1.6 s
Bibtex	**1.6 s**	2.6 s	12.4 s
Delicious	**1.5 s**	3.1 s	30.1 s
IMDB-F	**5.7 s**	58.0 s	658.7 s

[1] https://bitbucket.org/len_feremans/lcif_knn_pub

List of Figures

1.1 Fragment of the novel Moby Dick written by Herman Melville. We highlight 4 sequential patterns . . . 2
1.2 Data for 15 days of activity of a wind turbine. On the first day (W_5) pattern 6 occurs, meaning that events 203, 221, 224, 225, 309 and 310 co-occur and the wind turbine has stopped. On the second and fifth day (W_6 and W_9) pattern 122 occurs, meaning events 221, 224, 225, 230 and 310 co-occur and the turbine is remotely paused and re-started . . . 2
1.3 Frequent sequential patterns mined in the fragment of Moby Dick . . . 3
1.4 Cohesive sequential patterns mined in the fragment of Moby Dick . . . 4
1.5 Example of a cohesive itemset, a sequential pattern and an episode . . . 5
1.6 Illustrative example of k-nearest neighbours classification . . . 6
1.7 Illustrative example of finding the nearest neighbours using term-at-a-time and document-at-a-time with pruning . . . 7

2.1 Example of a cohesive itemset, a representative sequential pattern and a dominant episode . . . 21
2.2 Effect of *varying noise* on the discovery of a single episode by state-of-the-art methods and FCI$_{seq}$ . . . 39
2.3 Effect of *varying the alphabet size* on the discovery of a single episode by state-of-the-art methods and FCI$_{seq}$ . . . 40
2.4 Effect of *varying the probability of omitting events* of source episodes on the discovery of a single episode by state-of-the-art methods and FCI$_{seq}$ . . . 40
2.5 Effect of *varying maximum time delay* on the discovery of a single episode by state-of-the-art methods and FCI$_{seq}$ . . . 41
2.6 Effect of *varying the number of episodes* and the discovery of them by state-of-the-art methods and FCI$_{seq}$ . . . 42
2.7 Effect of *min_coh*, θ and *max_size* thresholds on the number of patterns and runtime 51
2.8 Effect of mining representative sequential patterns, dominant episodes and association rules on runtime on *Species* and *Trump* . . . 53

3.1 Fragment of the novel Moby Dick written by Herman Melville. We highlight 4 sequential patterns having a high value of quantile-based cohesion . . . 61
3.2 Illustrative example of QCSP, a prefix-projected pattern growth algorithm that employs pruning using mingap and an upper bound on quantile-based cohesion . . . 68
3.3 Number of candidates (log scale) visited by QCSP with and without pruning . . . 74

4.1 Effect of thresholding on the distribution of actual versus predicted labels on the imbalanced dataset *Corel5k* . . . 99

List of Tables

2.1 Example of computing minimal windows of itemset $\{a, b, c\}$ in sequence *aabcccccacb* 30
2.2 Top 5 *itemsets* discovered by FCI_{seq} and the state-of-the-art methods on *Species* and *Trump* . . . 44
2.3 *Overlap* in the top k itemsets discovered by FCI_{seq} and the state-of-the-art methods on *Species* and *Trump* . . . 45
2.4 Top 5 *sequential patterns* discovered by FCI_{seq} on *Species* . . . 45
2.5 Top 5 *sequential patterns* discovered by FCI_{seq} and the state-of-the-art methods on *Species* and *Trump* . . . 46
2.6 Top 5 *dominant episodes* discovered by FCI_{seq} on *Species* . . . 47
2.7 Top 5 *episodes* discovered by the state-of-art methods on *Species* . . . 48
2.8 Top 5 *rules* discovered by FCI_{seq} and the state-of-the-art methods on *Species* and *Trump* . . . 49
2.9 Top 5 *rules* discovered by FCI_{seq} on sequences of characters in different languages . 50
2.10 Effect of varying *max_size* on the *number of candidates* enumerated by FCI_{seq} on *Species* . . . 52

3.1 Characteristics of datasets for comparing the performance of QCSP versus state-of-the-art methods . . . 73
3.2 Runtimes for QCSP and state-of-the-art methods on all datasets . . . 74
3.3 Top 5 sequential patterns for QCSP and state-of-the-art methods on *Synthetic* . . . 75
3.4 Top 5 sequential patterns for QCSP and the state-of-the-art methods on *Moby*, *JMLR* and *Trump* . . . 76

4.1 Characteristics of five large and five extreme multi-label datasets . . . 94
4.2 Comparing the accuracy of LCIF with ML-KNN, IBLR and BR-SMO on the large datasets 98
4.3 Comparing the accuracy of LCIF with FASTXML on the extreme datasets . . . 99
4.4 Pruning of INSTANCEKNNFAST and state-of-the-art top-k query retrieval methods . 101
4.5 Runtime of INSTANCEKNNFAST and state-of-the-art top-k query retrieval methods . 101
4.6 Runtime results of LCIF and the state-of-the-art multi-label classifications methods on the large datasets . . . 102
4.7 Runtime results of LCIF on the extreme datasets . . . 103

A.1 Top 25 *itemsets* discovered by FCI_{seq} and the other methods on *Species* . . . 122
A.2 Top 25 *sequential patterns* discovered by FCI_{seq} and other methods on *Species*. . . . 123
A.3 Top 25 *rules* discovered by FCI_{seq} and other methods on *Species* . . . 124
A.4 Top 25 *itemsets* discovered by FCI_{seq} and other methods on *Trump* . . . 125
A.5 Top 25 *sequential patterns* discovered by FCI_{seq} and other methods on *Trump* . . . 126
A.6 Top 25 *rules* discovered by FCI_{seq} and other methods on *Trump* . . . 127

B.1 Top 20 sequential patterns discovered by QCSP and other methods on *Moby* . . . 130
B.2 Top 20 sequential patterns discovered by QCSP and other methods on *JMLR* . . . 130

B.3 Top 20 sequential patterns discovered by QCSP and other methods on *Trump* 131

C.1 Comparing the accuracy of second order instance-based kNN with instance-based kNN, ML-KNN and BR-SMO on *Delicious* 135

C.2 Comparing the runtime of the second order instance-based kNN variation on large datasets 135

List of Algorithms

2.1 $\textsc{Fci}_{seq}(\mathcal{S}, \theta, max_size, min_coh, min_or, min_por, min_conf)$ Mine cohesive itemsets, representative sequential patterns, dominant episodes and association rules in a single sequence . . . 23

2.2 $\textsc{Dfs}(\mathcal{S}, X, Y, max_size, min_coh, min_or, min_por, min_conf)$ Pruned depth-first search to find cohesive itemsets and post-processing for other types of patterns and rules . . . 23

2.3 $\textsc{Sum_Min_Wins}(\mathcal{S}, X, Y)$ Compute sum of minimal windows of each occurrence of X in $\mathcal{S}$ . . . 29

2.4 $\textsc{Find_Sequential_Patterns}(\mathcal{S}, X)$ Generate sequential patterns based on cohesive itemset X . . . 31

2.5 $\textsc{Find_Dominant_Episode}(\mathcal{S}, X, sps, min_por)$ Generate a dominant episode based on cohesive itemset X . . . 33

2.6 $\textsc{Find_Rules}(\mathcal{S}, X, min_conf)$ Generate confident association rules from cohesive itemset X . . . 36

3.1 $\textsc{Qcsp}(\mathcal{S}, k, \alpha, max_size)$ Mine top-k quantile-based cohesive sequential patterns in a single sequence . . . 65

3.2 $\textsc{Proj_Candidates}(\mathcal{S}, X_s, \mathcal{P}_{X_s})$ Compute candidate itemset Y based on projection 66

3.3 $\textsc{Project}(\mathcal{S}, Z_s, \mathcal{P}_{X_s}, \alpha, max_size)$ Computes pseudo-projection incrementally . . . 67

4.1 $\textsc{CreateIndex}(\mathcal{D})$ Creates an inverted index for instance-based kNN baseline . . . 85

4.2 $\textsc{InstanceKnnSearch}(x_q, k, \text{IID})$ Finds the k-nearest neighbours for x_q in $\mathcal{D}$ . . . 86

4.3 $\textsc{InstanceKnnPredict}(x_q, \text{kNN}, \alpha)$ Computes instance-based confidence scores for labels . . . 86

4.4 $\textsc{CreateSimilMatrix}(\mathcal{D})$ Computes similarities between all features and labels for feature-based kNN . . . 88

4.5 $\textsc{FeatureKnnPredict}(x_q, S, \beta)$ Computes feature-based confidence scores for labels 88

4.6 $\textsc{Lcif}(\mathcal{D}, X_{test}, k, \alpha, \beta, \lambda, t)$ Predicts labels based on a linear combination of instance- and feature-based weighted similarities . . . 90

4.7 $\textsc{CreateIndexPartition}(\mathcal{D}, m)$ Partitions data and builds indexes for both TAAT and DAAT traversal . . . 92

4.8 $\textsc{InstanceKnnFast}(x_q, k, \Phi)$ Finds the exact k-nearest neighbours from $\mathcal{D}$ based on two traversal strategies and pruning . . . 93

A.1 $\textsc{Sum_Min_Wins}_{\text{seq}}(\mathcal{S}, X, Y)$ Compute the sum and the set of minimal windows of each occurrence of X in $\mathcal{S}$ . . . 120

A.2 $\textsc{Compute_Support_Episode}(\mathcal{S}, X, G)$ Compute the support of a candidate episode in a sequence based on a cohesive itemset X . . . 121

List of Definitions, Problems and Theorems

2.1 Definition (Event sequence) 16
2.2 Definition (Itemset support) 16
2.3 Definition (Minimal window) 17
2.4 Definition (Cohesion) 17
2.1 Problem (Frequent cohesive itemset) 17
2.5 Definition (Sequential pattern) 18
2.6 Definition (Sequential pattern occurrence ratio) 18
2.7 Definition (Representative sequential pattern) 19
2.8 Definition (Episode) 19
2.9 Definition (Episode occurrence ratio) 19
2.10 Definition (Intersecting episode) 20
2.11 Definition (Dominant episode) 20
2.12 Definition (Extended average window size) 21
2.13 Definition (Rule confidence) 21
2.14 Definition (Upper bound cohesion) 24
2.1 Lemma (A ratio inequality) 25
2.2 Theorem (Upper bound on cohesion) 25
2.3 Theorem (Upper bound sum of minimal windows) 28
2.4 Theorem (Computing rule confidence based on single antecedents) 35

3.1 Definition (Event sequence) 62
3.2 Definition (Sequential pattern) 62
3.3 Definition (Minimal window) 62
3.4 Definition (Support sequential pattern) 62
3.5 Definition (Quantile-based cohesion) 63
3.1 Problem (Top-k quantile-based cohesive sequential patterns) 63
3.6 Definition (Supersequence) 64
3.7 Definition (Projection) 64
3.8 Definition (Candidate items) 64
3.9 Definition (Minimal gap) 68
3.1 Theorem (Limit quantile-based cohesion using mingap) 68
3.2 Theorem (Upper bound on quantile-based cohesion) 70

4.1 Definition (Multi-label dataset) 83
4.2 Definition (Cardinality) 83
4.3 Definition (Column) 84
4.4 Definition (Density) 84

4.1 Problem (Multi-label classification) 84
4.5 Definition (Instance-based cosine similarity) 84
4.6 Definition (Instance-based confidence score) 85
4.7 Definition (Feature-based cosine similarity) 87
4.8 Definition (Feature-based confidence score) 87
4.9 Definition (LCIF confidence score) 89
4.10 Definition (Single threshold) 89
4.11 Definition (Label-specific threshold) 89
4.12 Definition (Partition) 91
4.13 Definition (Upper bound cosine similarity) 92
4.14 Definition (Example-based metrics) 95
4.15 Definition (Label-based metrics) 96
4.16 Definition (Micro and macro precision) 96
4.17 Definition (Precision at k) 96

B.1 Definition (Weighted quantile-based cohesion) 129

C.1 Definition (Full cosine similarity) 133
C.2 Definition (Second order label weight) 133
C.3 Definition (Second order instance-based confidence score) 134

Bibliography

R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *International Conference on Very Large Data Bases*, volume 1215, pages 487–499, 1994.

D. C. Anastasiu and G. Karypis. Fast parallel cosine k-nearest neighbor graph construction. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*, pages 50–53. IEEE Press, 2016.

A. Awekar and N. F. Samatova. Fast matching for all pairs similarity search. In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, volume 1, pages 295–300. IEEE, 2009.

R. Babbar and B. Schölkopf. Dismec: Distributed sparse machines for extreme multi-label classification. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 721–729. ACM, 2017.

R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web*, pages 131–140. ACM, 2007.

K. Bhatia, H. Jain, P. Kar, M. Varma, and P. Jain. Sparse local embeddings for extreme multi-label classification. In *Advances in Neural Information Processing Systems*, pages 730–738, 2015.

K. Bhatia, K. Dahiya, H. Jain, Y. Prabhu, and M. Varma. The extreme classification repository: multi-label datasets & code. *http://manikvarma.org/downloads/XC/XMLRepository.html*, 2016.

J. Bobadilla, F. Ortega, A. Hernando, and J. Bernal. A collaborative filtering approach to mitigate the new user cold start problem. *Knowledge-based systems*, 26:225–238, 2012.

J. S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. *arXiv preprint arXiv:1301.7363*, 2013.

M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of data*, pages 93–104. ACM, 2000.

A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth International Conference on Information and Knowledge Management*, pages 426–434. ACM, 2003.

N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

W. Cheng and E. Hüllermeier. Combining instance-based learning and logistic regression for multilabel classification. *Machine Learning*, 76(2-3):211–225, 2009.

D. W. Cheung, J. Han, V. T. Ng, and C. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *IEEE International Conference on Data Engineering*, pages 106–114. IEEE, 1996.

K. W. Church and R. L. Mercer. Introduction to the special issue on computational linguistics using large corpora. *Computational linguistics*, 19(1):1–24, 1993.

B. Cule, B. Goethals, and C. Robardet. A new constraint for mining sets in sequences. In *Proceedings of the 2009 SIAM International Conference on Data Mining*, pages 317–328. Society for Industrial and Applied Mathematics, 2009.

B. Cule, N. Tatti, and B. Goethals. Marbles: Mining association rules buried in long event sequences. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 7(2):93–110, 2014.

B. Cule, L. Feremans, and B. Goethals. Efficient discovery of sets of co-occurring items in event sequences. In *Joint European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, pages 361–377. Springer, 2016.

B. Cule, L. Feremans, and B. Goethals. Efficiently mining cohesion-based patterns and rules in event sequences. *Data Mining and Knowledge Discovery*, 33(4):1125–1182, 2019.

P.-J. Daems, L. Feremans, T. Verstraeten, B. Cule, B. Goethals, and J. Helsen. Fleet-oriented pattern mining combined with time series signature extraction for understanding of wind farm response to storm conditions. In *Second World Congress on Condition Monitoring*, 2019.

C. Darwin. On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life (DATASET). 1859. URL http://www.gutenberg.org/ebooks/1228.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

C. Dickens. David copperfield (DATASET). 1850. URL http://www.gutenberg.org/ebooks/766.

S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and development in Information Retrieval*, pages 993–1002. ACM, 2011.

K. Draszawka and J. Szymański. Thresholding strategies for large scale multi-label text classifier. In *6th International Conference on Human System Interactions*, pages 350–355. IEEE, 2013.

A. Elisseeff, J. Weston, et al. A kernel method for multi-labelled classification. In *NIPS*, volume 14, pages 681–687, 2001.

R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

L. Feremans. Mining cohesion-based patterns and rules in event sequences (SOFTWARE). 2019. URL https://bitbucket.org/len_feremans/fci_public.

L. Feremans, B. Cule, C. Devriendt, B. Goethals, and J. Helsen. Pattern mining for learning typical turbine response during dynamic wind turbine events. In *ASME 2017 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers Digital Collection, 2017a.

L. Feremans, B. Cule, C. Vens, and B. Goethals. Combining instance and feature neighbors for efficient multi-label classification. In *2017 IEEE International Conference on Data Science and Advanced Analytics*, pages 109–118. IEEE, 2017b.

L. Feremans, B. Cule, and B. Goethals. Mining top-k quantile-based cohesive sequential patterns. In *Proceedings of the 2018 SIAM International Conference on Data Mining*, pages 90–98. Society for Industrial and Applied Mathematics, 2018.

L. Feremans, V. Vercruyssen, B. Cule, W. Meert, and B. Goethals. Pattern-based anomaly detection in mixed-type time series. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2019a.

L. Feremans, V. Vercruyssen, W. Meert, B. Cule, and B. Goethals. A framework for pattern mining and anomaly detection in multi-dimensional time series and event logs. In *Post-Proceedings of the International Workshop on New Frontiers in Mining Complex Patterns*. Springer, 2019b.

L. Feremans, B. Cule, C. Vens, and B. Goethals. Combining instance and feature neighbors for extreme multi-label classification. *International Journal of Data Science and Analytics*, 2020.

M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proceedings of the VLDB Endowment*, 4(12):1213–1224, 2011.

P. Fournier-Viger, J. C.-W. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, and H. T. Lam. The spmf open-source data mining library version 2. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 36–40. Springer, 2016.

J. Fowkes and C. Sutton. A subsequence interleaving model for sequential pattern mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 835–844. ACM, 2016.

E. Gibaja and S. Ventura. Multi-label learning: a review of the state of the art and ongoing research. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 4(6):411–444, 2014.

P. D. Grünwald. *The minimum description length principle.* MIT press, 2007.

J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, 2004.

T. Hoang, F. Mörchen, D. Fradkin, and T. Calders. Mining compressing sequential patterns. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 7(1):34–52, 2014.

H. Jain, Y. Prabhu, and M. Varma. Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 935–944. ACM, 2016.

J. S. Justeson and S. M. Katz. Technical terminology: some linguistic properties and an algorithm for identification in text. *Natural language engineering*, 1(1):9–27, 1995.

S. Laxman, P. Sastry, and K. Unnikrishnan. A fast algorithm for finding frequent episodes in event streams. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 410–419. ACM, 2007.

M. Leemans and W. M. van der Aalst. Discovery of frequent episodes in event logs. In *International Symposium on Data-Driven Process Discovery and Analysis*, pages 1–31. Springer, 2014.

Y. Li, L. Liu, C. Shen, and A. Van Den Hengel. Mining mid-level visual patterns with deep cnn activations. *International Journal of Computer Vision*, 121(3):344–364, 2017.

C. Liu, L. Cao, and S. Y. Philip. A hybrid coupled k-nearest neighbor algorithm on imbalance data. In *2014 International Joint Conference on Neural Networks*, pages 2011–2018. IEEE, 2014.

Y. Liu. *Crafting Concurrent Data Structures*. PhD dissertation, Lehigh University, 2015.

H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

C. D. Manning and H. Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

N. Méger and C. Rigotti. Constraint-based mining of episode rules and optimal window sizes. In *Joint European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, pages 313–324. Springer, 2004.

H. Melville. Moby-dick; or, the whale (DATASET). 1851. URL http://www.gutenberg.org/ebooks/2701.

T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

S. Moens, E. Aksehirli, and B. Goethals. Frequent itemset mining for big data. In *SML: BigData 2013 Workshop on Scalable Machine Learning*. IEEE, 2013.

X. Ning and G. Karypis. Slim: Sparse linear methods for top-n recommender systems. In *2011 IEEE 11th International Conference on Data Mining*, pages 497–506. IEEE, 2011.

I. Partalas, A. Kosmopoulos, N. Baskiotis, T. Artières, G. Paliouras, É. Gaussier, I. Androutsopoulos, M. Amini, and P. Gallinari. LSHTC: A benchmark for large-scale text classification. *arXiv preprint arXiv:1503.08581*, 2015.

J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1424–1440, 2004.

J. Pei, J. Han, and W. Wang. Constraint-based sequential pattern mining: the pattern-growth methods. *Journal of Intelligent Information Systems*, 28(2):133–160, 2007.

F. Petitjean, T. Li, N. Tatti, and G. I. Webb. Skopus: Mining top-k sequential patterns under leverage. *Data Mining and Knowledge Discovery*, 30(5):1086–1111, 2016.

Y. Prabhu and M. Varma. Fastxml: A fast, accurate and stable tree-classifier for extreme multi-label learning. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 263–272. ACM, 2014.

J. Read, B. Pfahringer, and G. Holmes. Multi-label classification using ensembles of pruned sets. In *2008 Eighth IEEE International Conference on Data Mining*, pages 995–1000. IEEE, 2008.

J. Read, B. Pfahringer, G. Holmes, and E. Frank. Classifier chains for multi-label classification. *Machine learning*, 85(3):333–359, 2011.

J. Read, P. Reutemann, B. Pfahringer, and G. Holmes. Meka: a multi-label/multi-target extension to weka. *The Journal of Machine Learning Research*, 17(1):667–671, 2016.

P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, pages 175–186. ACM, 1994.

B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, pages 285–295. ACM, 2001.

R. E. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions. *Machine learning*, 37(3):297–336, 1999.

D. Shahaf and C. Guestrin. Connecting the dots between news articles. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 623–632, 2010.

G. Sidorov, A. Gelbukh, H. Gómez-Adorno, and D. Pinto. Soft similarity and soft cosine measure: Similarity of features in vector space model. *Computación y Sistemas*, 18(3):491–504, 2014.

E. Spyromitros, G. Tsoumakas, and I. Vlahavas. An empirical study of lazy multilabel classification algorithms. In *Hellenic conference on Artificial Intelligence*, pages 401–406. Springer, 2008.

R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology*, pages 1–17. Springer, 1996.

Y. Tagami. Annexml: Approximate nearest neighbor search for extreme multi-label classification. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 455–464. ACM, 2017.

S. Tan. Neighbor-weighted k-nearest neighbor for unbalanced text corpus. *Expert Systems with Applications*, 28(4):667–671, 2005.

N. Tatti. Discovering episodes with compact minimal windows. *Data Mining and Knowledge Discovery*, 28(4):1046–1077, 2014a.

N. Tatti. Mining closed strict episodes, includes implementation of WINEPI, LAXMAN and MARBLES$_W$ (SOFTWARE). 2014b. URL https://users.ics.aalto.fi/ntatti/software.shtml.

N. Tatti. Ranking episodes using a partition model. *Data Mining and Knowledge Discovery*, 29 (5):1312–1342, 2015.

N. Tatti and B. Cule. Mining closed strict episodes. *Data Mining and Knowledge Discovery*, 25 (1):34–66, 2012.

N. Tatti and J. Vreeken. The long and the short of it: summarising event sequences with serial episodes. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 462–470. ACM, 2012.

N. Tax, N. Sidorova, R. Haakma, and W. M. van der Aalst. Mining local process models. *Journal of Innovation in Digital Ecosystems*, 3(2):183–196, 2016.

I. Triguero and C. Vens. Labelling strategies for hierarchical multi-label classification techniques. *Pattern Recognition*, 56:170–183, 2016.

D. Trump. Tweets of president donald trump (DATASET). 2017. URL http://www.trumptwitterarchive.com.

G. Tsoumakas and I. Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining*, 3(3), 2006.

G. Tsoumakas, E. Spyromitros-Xioufis, J. Vilcek, and I. Vlahavas. Mulan: A java library for multi-label learning. *Journal of Machine Learning Research*, 12:2411–2414, 2011.

W. M. Van Der Aalst, H. A. Reijers, A. J. Weijters, B. F. van Dongen, A. A. De Medeiros, M. Song, and H. Verbeek. Business process mining: An industrial application. *Information Systems*, 32 (5):713–732, 2007.

C. Vens, J. Struyf, L. Schietgat, S. Džeroski, and H. Blockeel. Decision trees for hierarchical multi-label classification. *Machine Learning*, 73(2):185–214, 2008.

K. Verstrepen and B. Goethals. Unifying nearest neighbors collaborative filtering. In *Proceedings of the 8th ACM Conference on Recommender systems*, pages 177–184. ACM, 2014.

J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *IEEE International Conference on Data Engineering*, pages 79–90. IEEE, 2004.

J. Wang, A. P. De Vries, and M. J. Reinders. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *Proceedings of the 29th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 501–508. ACM, 2006.

X.-l. Wang, H. Zhao, and B. Lu. Enhanced k-nearest neighbour algorithm for largescale hierarchical multi-label classification. In *Proceedings of the Joint ECML/PKDD PASCAL Workshop on Large-Scale Hierarchical Classification, Athens, Greece*, volume 5, 2011.

G. I. Webb. Self-sufficient itemsets: An approach to screening potentially interesting associations between items. *ACM Transactions on Knowledge Discovery from Data*, 4(1):3, 2010.

C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *IEEE International Conference on Data Engineering*, pages 916–927. IEEE, 2009.

Y. Yang. A study of thresholding strategies for text categorization. In *Proceedings of the 24th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 137–145. ACM, 2001.

Z. Younes, F. Abdallah, and T. Denœux. Multi-label classification algorithm derived from k-nearest neighbor rule with label dependencies. In *2008 16th European Signal Processing Conference*, pages 1–5. IEEE, 2008.

R. B. Zadeh and A. Goel. Dimension independent similarity computation. *Journal of Machine Learning Research*, 14(1):1605–1626, 2013.

M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.

M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42 (1-2):31–60, 2001.

Z.-Q. Zeng, H.-B. Yu, H.-R. Xu, Y.-Q. Xie, and J. Gao. Fast training support vector machines using parallel sequential minimal optimization. In *2008 3rd International Conference on Intelligent System and Knowledge Engineering*, volume 1, pages 997–1001. IEEE, 2008.

M.-L. Zhang and Z.-H. Zhou. Ml-knn: A lazy learning approach to multi-label learning. *Pattern Recognition*, 40(7):2038–2048, 2007.

W. Zhang, F. Liu, L. Luo, and J. Zhang. Predicting drug side effects by multi-label learning and ensemble learning. *BMC bioinformatics*, 16(1):365, 2015.

W. Zhang, Y. Chen, and D. Li. Drug-target interaction prediction through label propagation with linear neighborhood information. *Molecules*, 22(12):2056, 2017.

C. Zhou, B. Cule, and B. Goethals. Pattern based sequence classification. *IEEE Transactions on Knowledge and Data Engineering*, 28(5):1285–1298, 2016.

A. Zimmermann. Understanding episode mining techniques: Benchmarking on diverse, realistic, artificial data. *Intelligent Data Analysis*, 18(5):761–791, 2014a.

A. Zimmermann. Generate event sequences (SOFTWARE). 2014b. URL https://zimmermanna.users.greyc.fr/software.html.