Monitoring Data Streams – Classification under Uncertainty and Entropy-based Dependency-Detection on Streaming Data

Jonathan Boidol

München 2017

Monitoring Data Streams – Classification under Uncertainty and Entropy-based Dependency-Detection on Streaming Data

Jonathan Boidol

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Jonathan Boidol
aus Regensburg

München, den 04.05.2017

Erstgutachter: Prof. Dr. Volker Tresp
Zweitgutachter: Prof. Dr. Albert Bifet
Tag der mündlichen Prüfung: 01.08.2017

Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12.07.11, § 8, Abs. 2 Pkt. .5.)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

Jonathan Boidol

Name, Vorname

Ort, Datum

Unterschrift Doktorand/in

Formular 3.2

Contents

List of Figures

List of Tables

Acknowledgements

For the last three years, many persons worked to ensure the success of this thesis, lending their assistance and support, their advice and their experience, to all of whom I feel deeply grateful. First, I would like to thank Prof. Dr. Volker Tresp for his supervision of this thesis. His greatest talent is his ability to put things into perspective and always see the larger picture. I feel fortunate to have had him as my advisor.

At Siemens, my thanks go to Ariane Sutor, who made all of this possible through her interest and trust. Dr. Andreas Hapfelmeier tirelessly helped me through all challenges, scientific and prosaic, and was always the first with whom I could discuss new ideas. Listening to him earlier and more often would have saved me a headache or ten. It was a pleasure to work and publish with him. All the people at IBI made the days more interesting and more pleasant. Thank you for having me.

Felix and Viola, who have an open door for a tired person in need of a break, deserve to be mentioned here as well. See you on Thursday.

Thank you so much to my family, which always listened patiently to bad explanations. Especially to my grandfather, my example for character and determination, Raphael, always the big brother, and, of course, to my parents for, well, everything.

I am eternally grateful to Lisa, who no only read these pages despite her own demanding work in their first and most error-filled form and reigned in my more daring sentences, but also encouraged me whenever I needed it and supported me throughout.

Abstract

Stream monitoring is concerned with analyzing data that is represented in the form of infinite streams. This field has gained prominence in recent years, as streaming data is generated in increasing volume and dimension in a variety of areas. It finds application in connection with monitoring industrial sensors, "smart" technology like smart houses and smart cars, wearable devices used for medical and physiological monitoring, but also in environmental surveillance or finance.

However, stream monitoring is a challenging task due to the diverse and changing nature of the streaming data, its high volume and high dimensionality with thousands of sensors producing streams with millions of measurements over short time spans. Automated, scalable and efficient analysis of these streams can help to keep track of important events, highlight relevant aspects and provide better insights into the monitored system. In this thesis, we propose techniques adapted to these tasks in supervised and unsupervised settings, in particular Stream Classification and Stream Dependency Monitoring.

After a motivating introduction, we introduce concepts related to streaming data and discuss technological frameworks that have emerged to deal with streaming data in the second chapter of this thesis. We introduce the notion of information theoretical entropy as a useful basis for data monitoring in the third chapter.

In the second part of the thesis, we present Probabilistic Hoeffding Trees, a novel approach towards stream classification. We will show how probabilistic learning greatly improves the flexibility of decision trees and their ability to adapt to changes in data streams. The general technique is applicable to a variety of classification models and fast to compute without significantly greater memory cost compared to regular Hoeffding Trees. We show that our technique achieves better or on-par results to current state-of-the-art tree classification models on a variety of large, synthetic and real life data sets.

In the third part of the thesis, we concentrate on unsupervised monitoring of data streams. We will use mutual information as entropic measure to identify the most important relationships in a monitored system. By using the powerful concept of mutual information we can, first, capture relevant aspects in a great variety of data sources with different underlying concepts and possible relationships and, second, analyze theoretical and computational complexity.

We present the MID and DIMID algorithms. They perform extremely efficient on high dimensional data streams and provide accurate results, outperforming state-of-the-art algorithms for dependency monitoring.

In the fourth part of this thesis, we introduce delayed relationships as a further feature in the dependency analysis. In reality, the phenomena monitored by e.g. some type of sensor might depend on another, but measurable effects can be delayed. This delay might be due to technical reasons, i.e. different stream processing speeds, or because the effects actually appear delayed over time. We present LOGLAG, the first algorithm that monitors dependency with respect to an optimal delay. It utilizes several approximation techniques to achieve competitive resource requirements. We demonstrate its scalability and accuracy on real world data, and also give theoretical guarantees to its accuracy.

Zusammenfassung

Stream Monitoring bezeichnet die kontinuierliche Analyse von Daten die in unbegrenzter Menge in Form von Datenströmen auftreten. Diese Art von Analysen hat in den letzten Jahren an Bedeutung gewonnen und große Entwicklungssprünge erfahren. Sie wurden notwendig, da Datenströme in zunehmender Zahl und zunehmendem Umfang von verschiedensten, komplexen Systemen erzeugt werden. Die Analyse von Datenströmen ist nötig und findet praktische Anwendung im Betrieb von industriellen, sensorüberwachten Anlagen, bei der Vielzahl von smarten Technologien wie smarten, vernetzten Autos und intelligenten Häusern, tragbaren Geräten zur Überwachung der pesönlichen Gesundheit oder eigenen Fitness, aber auch in der Kontrolle von Ökosystemen zur Überwachung von Erosion oder Wasserreinheit oder den unzähligen Kennziffern und Kursen aus der Finanzwirtschaft.

Stream Monitoring ist in der Regel eine technische und algorithmische Herausforderung aufgrund der vielfältigen Art und wechselhaften Natur von Datenströmen und wegen des großen Datenvolumens, der Vielzahl an parallelen Strömen und oft auch hohen zeitlichen Frequenz von tausenden Sensoren die Datenströme mit Millionen von Messungen aus einer Anlage erzeugen. Die automatische, skalierbare und effiziente Analyse dieser Datenströme kann dabei unterstützen, wichtige Ereignisse zu erkennen, relevante Entwicklungen hervorzuheben und ein besseres Verständnis für das überwachte System zu gewinnen. Diese Arbeit beschäftigt sich mit dem Lernen auf Datenströmen mit und ohne Klassenlabel. Es werden neue Methoden vorgestellt, die sich zum Stream Monitoring besonders durch ihre breite Anwendbarkeit eignen. Dies sind Methoden zur Klassifikation von Datenströmen und zur Überwachung von Abhängigkeiten in Datenströmen.

Diese Arbeit gliedert sich in vier Hauptteile. Nach einer allgemeinen Einführung wird in den Kapiteln 2 und 3 das Konzept von Datenströmen vorgestellt und einige Frameworks diskutiert, die speziell zum Arbeiten mit Datenströmen entwickelt wur-

den. Zusätzlich werden wir die Informationsentropie und verwandte Ideen vorstellen, die eine theoretische Grundlage für unser Datenmonitoring liefern.

Im zweiten Teil stellen wir Probabilistic Hoeffding Trees vor, einen neuartigen Algorithmus zur Datenstromklassifikation. Wir zeigen, wie probabilistisches Lernen in Entscheidungsbäumen ihre Flexibilität und ihre Fähigkeit, sich auf Veränderungen der Datenströme einzustellen, deutlich erhöht. Die vorgestellte Technik eignet sich nicht nur für eine Vielzahl von Klassifikationsmodellen, sondern ist zusätzlich schnell und ohne signifikant höheren Speicherverbrauch zu berechnen. Wir zeigen, dass unsere Methode bessere Klassifikationsergebnisse auf verschiedenen, großen, synthetischen oder von Sensoren erzeugten Datensätzen erzielt als state-of-the-art Klassifikatoren mit Baummodellen.

Der dritte Teil dieser Arbeit konzentriert sich auf das Monitoring von Datenströmen ohne Klassenlabel. Wir benutzen die Transinformation, ein entropisches Maß für gegenseitige Abhängigkeit, um die wichtigsten Beziehungen in einem überwachten System zu identifizieren. Ein vielseitiges Maß wie die Transinformation ist notwendig, um die jeweils relevanten Aspekte in verschiedensten Datenquellen mit unterschiedlichen Entstehungsmodellen und Beziehungen zueinander abzubilden. Dies bringt aber auch theoretische Schwierigkeiten und zusätzliche Berechnungskomplexität mit sich. Wir stellen MID und DIMID vor, Algorithmen, die diese Komplexitäten elegant bewältigen. Sie arbeiten sehr effizient auf hochdimensionalen Datenströmen und erzielen genaue Ergebnisse, welche die von state-of-the-art Algorithmen zur Überwachung von Abhängigkeiten übertreffen.

Im vierten Teil dieser Arbeit nehmen wir Abhängigkeiten mit zeitlicher Verzögerung in die Analyse von Abhängigkeiten mit auf. In praktischen Anwendungsfällen können die gemessenen Größen zwar voneinander abhängen, aber die messbaren Effekte erscheinen erst verzögert. Diese Verzögerung kann technische Gründe haben, beispielsweise unterschiedliche Verarbeitungsgeschwindigkeiten in verschiedenen Sensoren. Sie kann aber auch erscheinen, weil eine Ursache messbare Effekte erst mit Zeitverzögerung auslöst. Wir stellen Loglag vor, den ersten Algorithmus der Abhängigkeiten unter Berücksichtigung einer optimalen Verzögerung überwacht. Er zieht mehrere Approximationstechniken heran um den Verbrauch an Computerressourcen zu minimieren. Wir demonstrieren die Skalierbarkeit und Zuverlässigkeit an mehreren Datensätzen und geben theoretische Garantien für seine Genauigkeit.

Chapter 1

Introduction

Data Mining and Machine Learning are the predominant techniques we use today to teach computers to learn from data, make data-based inferences or draw conclusions from large amounts of data. Over the last decades however, there has been a fundamental shift going on in the way we collect and receive this data. It is no longer just (manually or automatically) gathered in batches of limited size and fed into an algorithm for processing, analysis or training. Instead, data is continuously created by computers which are connected directly to other computers. In this new situation, data becomes available piece by piece and over time. We may of course still store it in a persistent way and process it as we are used to, once sufficient data has accumulated. But in many applications it is more appropriate to think of the data as a *data stream*, practically infinite in size and duration and without persistence on a computer.[45]

1.1 Learning from Streaming Data

Most machine learning and data mining algorithms are designed with a scenario in mind, where all data is available in full from the beginning of the learning process and drawn independently from a stationary distribution. In this scenario, it is possible and perfectly sensible to process the data several times, often independent of the order of the processed data.

The world of data streams on the other hand is more complex. If data is streamed over time, we need algorithms that deal with small but infinitely growing data sets, incorporate new data as it is received, and ensure optimal results at every step. The streams or rather the stream sources frequently will be dynamic and not stationary, meaning that important characteristics of the streams change over time. If algorithms

are applied in such a dynamic environment, they must reflect this dynamic as well, be able to adapt their internal model over time and discard outdated information.

Stream monitoring is concerned with continuously analyzing data that is represented in the form of infinite streams of data. This field has gained prominence in recent years, as streaming data is generated in increasing volume and dimension in a variety of areas. "Smart" technologies like smart houses and smart cars, wearable devices used for medical and physiological monitoring become more common. But also systems on a larger scale like the smart energy grid currently under development, environmental surveillance systems for erosion or pollution, financial and industrial facilities are equipped with sensors that produce data streams in quantities undreamed of before. Automated systems are needed that cope with this wealth of data appropriately and in accordance with its streaming, dynamic nature.

However, stream monitoring is a challenging task due to the diverse and changing nature of the streaming data, its high volume and high dimensionality with thousands of sensors producing streams with millions of measurements. Automated, scalable and efficient analysis of these streams helps to keep track of important events, highlight relevant aspects, and provides better insights into the monitored system. In this thesis, we propose techniques adapted to streaming data, in particular Stream Classification and Stream Dependency Monitoring. These are examples for a supervised and an unsupervised task respectively.

Such supervised and unsupervised settings are one way to divide the fields of machine learning and data mining. They are distinguished by the available target we try to teach to an algorithm. In supervised settings we have some sort of label for every instance of the data that indicates one of several distinct classes or functions. Most often, the label is available during the algorithm training and the goal is to predict the label from as-yet unlabelled new data. Exemplary tasks are classification and regression, the prediction of discrete and real valued labels. Unsupervised data does not have such a well-defined label. The tasks are more general, to find structure or patterns in the data, for example clustering or anomaly detection.

Both settings are highly relevant to the streaming scenario and both are topic of this thesis. The central theme in the techniques in this thesis is an effort to quantify the information content inherent in the kinds of measurements that constitute data streams. We will show that taking the uncertainty of measurements into account greatly improves classification accuracy in dynamic streams. In unlabelled streams, we will use entropy, a measure of the order contained in a system, as a powerful tool to identify meaningful relationships.

1.2 Stream Mining Tasks

Inspired by challenges in network monitoring, web mining, sensor networks and many other areas that produce large amounts of streaming data, certain common problems and research issues have emerged. The following overview is by no means comprehensive but may serve as starting point for further enquiries.

Stream Prediction is likely the most well studied problem in recent stream mining literature. Numerous approaches have been published that deal with the different aspects of stream classification and regression. Some algorithms are direct adaptations of batch predictors, others deal with the specific problems of stream mining like the stationary distribution of data or load shedding, an issue of very high frequency streams that threaten to overload algorithms. One of the best known classifiers is the VFDT or Hoeffding Tree by Domingos *et al.* [38] and its many derived forms.

Stream Clustering is a particular difficult problem on data streams. Not only is it necessary to respect the development of clusters over time but the order of data in a stream might heavily influence the results. This has significant impact since we cannot randomly access and reorder streaming data. Nevertheless, there exist single pass or streaming clustering algorithms, the earliest being the hierarchical Leader algorithm [91], and streaming adaptations of the well-known k-means algorithm for example by Farnstrom *et al.* [41] and Domingos *et al.* [39].

Other algorithms concentrate on queries over the whole stream history, for example to answer the question after the most frequently appearing events or to count how many events of one type have occurred. These *frequency counting* or *frequent pattern* algorithms often use sub samples of the data stream or otherwise reduce the data volume. The first streaming algorithm to estimate item frequencies was the Count Sketch[26]. It uses several independent hash-tables to answer such queries space-efficiently. The famous Count-Min Sketch [31] works on similar principles but provides stronger theoretical guarantees for its results.

Many algorithms deal with more specialized problems which appear due to the particular nature of streaming data and often involve adapting a model to new developments. Among those are *Change Detection* itself and, closely related, *Novelty Detection* and *Anomaly Detection* where short-term, exceptional events have to be distinguished from normal behaviour or gradual changes.

Literature that provides a more thorough introduction into stream learning and discusses techniques and algorithms exhaustively and with respect to their technical details can be found for example by Domingos *et al.* [40], Aggarwal *et al.* [1] and Gama *et al.* [45, 46]. An excellent introduction on machine learning in general including techniques for stream learning are the books by Mitchell [71] or Witten [98].

This thesis deals with problems in stream classification and dependency detection in streams.

1.3 Contributions of this Work

In this thesis we study monitoring techniques of dynamic data streams and their application to large sets of streaming sensor data. Its contributions can be summarized as follows:

Uncertainty-aware Stream Classification We will describe a novel class of online decision trees, a classifier for streaming data. Based on the well known Hoeffding Trees, we develop Probabilistic Hoeffding Trees. They are capable of dealing with sudden concept changes in a data stream and provide fast, accurate classification results. The key idea is to incorporate the concept of uncertain data into the decision model which improves and speeds up the learning process on data streams with non-stationary distributions.

Entropy-based Dependency Monitoring of Data Streams The second contribution is a framework to monitor dependencies between data streams. The reason we look for dependencies is the simple assumption that different data streams that behave similarly over time, i. e. show similar, mutually predictable behaviour, indicate interesting subgroups in the whole system. We show how mutual information, an entropy-based measure for information shared between streams, can be used as a measure for dependency that is not only optimized for certain types of relationships. Calculating the mutual information in a carefully optimized way allows an incremental computation which allows excellent efficiency.

Entropy-based Lag Detection of Data Streams A further complication in monitoring dependencies are time delays in the data. Imagine, for example, relationships in weather data where temperature changes have delayed effects on humidity or precipitation or multiple sensors picking up changes caused by the same event, but from varying distances. We define the problem of lagged dependency as the analysis of two or more data streams for dependence and for the lag at which the dependence is the strongest. We develop an algorithm to efficiently calculate the lagged dependency which uses geometric sampling and adaptive compression of old data to provide very accurate results for smaller delays and a good approximation for large delays.

All significant contributions have been published in the conferences proceedings listed below as peer-reviewed articles. All of these were written largely by the author, with the editorial oversight and assistance of Andreas Hapfelmeier and Volker Tresp. Excerpts of the chapters that are taken verbatim from these publications are written by myself.

[22] J. Boidol, A. Hapfelmeier, and V. Tresp. Probabilistic Hoeffding trees. In *Industrial Conference on Data Mining, Best Paper Award*, pages 94–108. Springer, 2015

[19] J. Boidol and A. Hapfelmeier. Detecting data stream dependencies on high dimensional data. In *The 1st International Conference on Internet of Things and Big Data, IoTBD 2016*, pages 375–382. INSTICC, 2016

[20] J. Boidol and A. Hapfelmeier. Fast mutual information computation for dependency-monitoring on data streams. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2017

[21] J. Boidol and A. Hapfelmeier. Lagged dependencies in data streams. *In Review to: IEEE Transactions on Knowledge and Data Engineering*, 2017

The rest of this thesis is structured as follows: The second chapter serves as an introduction to data streams from a theoretical point of view. It also discusses some of the recently emerging streaming engines, technological frameworks that aim to follow the characteristics of streaming data and facilitate high-performance stream analytics in practice. The quantification of information is a central theme to this work. We heavily use the concept of entropy as defined by information theory in this regard, so Chapter 3 defines this and other related concepts to support the later chapters theoretically. Chapters 4 to 6 are centered around the aforementioned contributions and describe them in detail. They also include descriptions of our implementations and the design and results of experiments to evaluate each algorithm in practice. Chapter 7 summarizes this work and concludes with our thoughts on future research opportunities.

Chapter 2

Streaming Data

In this chapter, we will introduce the notion of streaming data and highlight differences to batch data. We discuss technological frameworks that have emerged to deal with streaming data.

We can imagine data streams as a series of events which occur continuously. Messages of these events are transmitted from their source to a processing computer. When we compare them to batch data, data streams need different management systems as well as a different data model. We will discuss data stream models in Section 2.1. Section 2.2 discusses the differences between learning from batch data and learning from streaming data. Section 2.3 introduces a uniform notation and terminology we use throughout this thesis. Not only algorithms but also the data management systems themselves need to follow a different logic to handle streaming data. We will discuss Stream Management Systems in Section 2.4. Section 2.5 discusses the sources for our streams.

2.1 Streaming Models

We can imagine data streams as a stochastic process that generates samples one after another, independently from each other, and for an arbitrary long time. This model explains the differences we see between batch data and stream data (cf. Table 2.1): Data arrives one by one and we have no control over the order it arrives in. There is also no known size of the total data, it is effectively infinite. This also means, we cannot store all data but have to discard (almost all of) it after processing.

We can distinguish several data stream models, here given by decreasing generality. Those are the turnstile model, the additive model and the time-series model.[72]

Table 2.1: Properties of Batch Data and Streaming Data.[45].

Batch Data	**Stream Data**
Complete data set	Arrival of data in increments
Random access to data	No control over order of arriving data
Fixed size	Unbound size
Permanent storage of data	Data not (completely) stored after processing

They each describe a series of items a_i characterizing a function A.

- In the *turnstile model*, every element represents an update of a counter in A. Each a_i can be thought of as a tuple (j, u_i) where the j-th counter is incremented by an amount u_i. Updates can be negative and for example cancel previous increments. This is the most general model as the current state of a counter gives no indication of the previous states. We can imagine this for example as the number of users currently logged onto a number of server in a network.
- The *additive model* restricts the turnstile model. Updates can now only be positive integers. It is useful to describe for example network traffic where we monitor the number of times an IP address accesses a server
- In the *Time-Series model*, the items a_i simply describe the current state of the process A. We can formulate it in the Turnstile model as $a_i = (i, a_i)$. In a sense it is the most restrictive model since every counter is used only once. However, it is also very expressive and widely used since it is appropriate to describe data streams from continuous measurements, like sensor measurements or stock prices.

Another way to look at streaming models is the type of information that is transmitted. The updates could contain either values or events.

- In event streams each stream element represents an event that happened at the specified time. We generally have only a limited number of possible events, for example alarms or discrete levels of a condition (low, medium, high for temperature, etc.).
- In real valued streams, the elements are real numbers, giving a potentially more fine grained view and allowing more complex processing.

We can draw a comparison to discrete and numeric features in machine learning to understand the differences between these types. Some classes of stream problems apply only to event streams, for example the aforementioned frequency counting algorithms. Of course the two types might be mixed together in a real system. Problems like classification and clustering frequently have to deal with both types of streams.

2.2 Stream Learning and Batch Learning

The traditional batch approach in machine learning assumes that all data, i. e. the whole data set we operate on, is fully accessible during the learning process. This requires that the whole data set fits into main memory or is at least quickly accessible on a high performing file system like HDFS, the distributed file system used by Hadoop. [5] It also assumes that the data collection is already completed and no other, new relevant data has to be taken into account. This approach is shown in Figure 2.1.

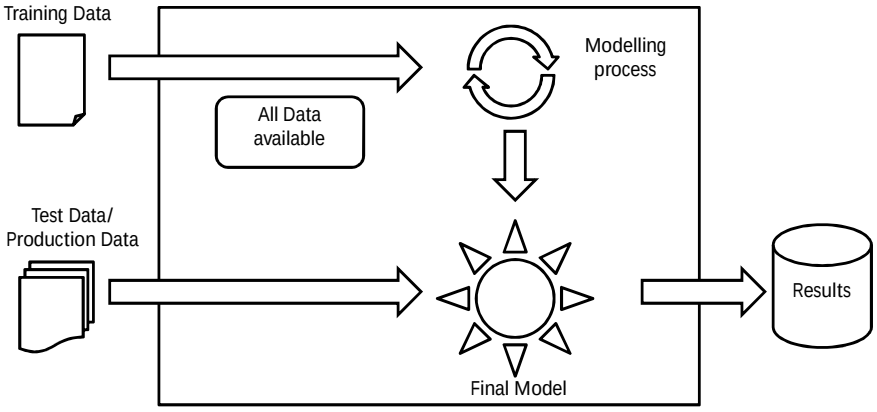


Figure 2.1: Batch learning setting.

The batch procedure obviously breaks down in the data stream scenario. We cannot wait for all data to arrive since the stream has no end. We also cannot relearn a full model whenever new data arrives since the computational cost will likely be too high for non-trivial models.[1] We could sample the incoming data down to a manageable load, for example if we keep randomly selected instances in a buffer and discard equally at random whenever our allotted storage capacity is reached. Sampling and periodic re-learning on this sample can be a valid approach to adapt batch algorithms

[1] We can keep simple aggregate statistics like mean, minimum and maximum values, or event counts. These will help us to build a model but are not sufficient themselves.

to data streams. However, this seems also wasteful and runs into another problem: Unlike a static data set, streams may undergo changes over time. In this case, older data does not reflect the current situation anymore and would not only slow down the model building but actually harm the model quality.

Instead of relearning repeatedly, it seems more appropriate to create new algorithms for streaming applications where the stream requirements have been taken into account from the start. Such algorithms are called online learners or incremental learners. The different process is illustrated in Figure 2.2.

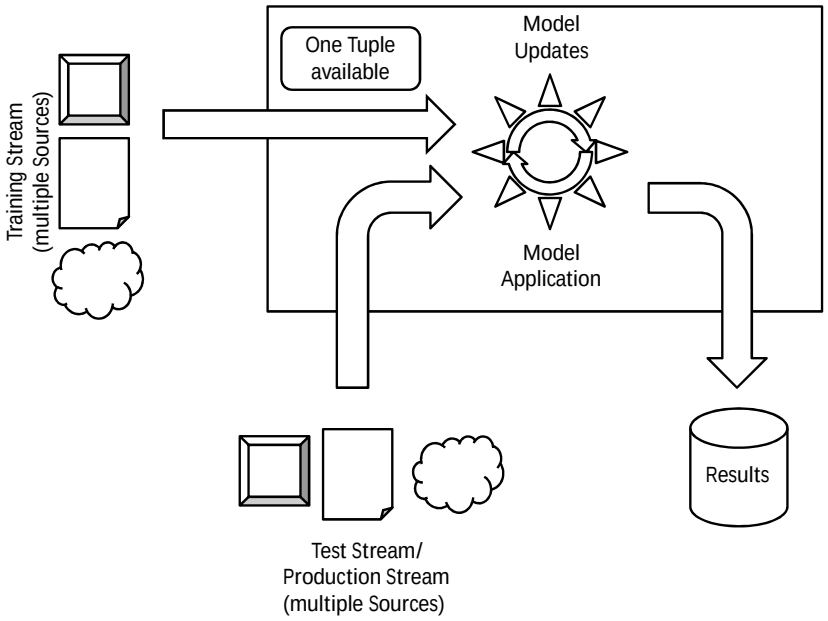


Figure 2.2: Online and incremental learning setting.

Four main requirements have emerged for online learning [40]

1. Iterative/Only once processing of a tuple
2. Constant time to process every instance
3. Constant limited memory
4. Anytime Readiness

The main idea is a type of model, that can be quickly updated instead of having to re-learn a model from scratch. Only one instance is processed at a time as it comes in. The algorithm also has no control over the order of the stream and simply processes one after the other. After the processing, the instance cannot be looked at again. This is the first property from the list above. We can stretch it slightly by selecting and storing some instances, for example those we consider typical or simply keep a random sample we replace over time. Nevertheless, we cannot randomly access all instances at will.

The processing and model update have to be done quickly enough to keep up with the stream. This means, there is a fixed time limit for every instance, the second property in our list. In particular there must be no connection between the number of instances seen so far and the current processing time. The first update should not take longer than the millionth update otherwise the processing time will grow without bound. Ideally, a single update will not take longer than the time it takes for the next instance to arrive. Otherwise, we have to skip over some instances – so called load shedding – or use some sort of buffer in the hope that we can catch up with the stream in the future.

An important limitation is the restriction of memory usage, either because it is actually infeasible to use that much memory or simply because it would be too expensive. Similar to the processing time in the second requirement, the processing of a new instance and model update may only use a fixed amount of memory or storage. The most relevant consequence of this is that we cannot store all or even a proportion of the stream instances for future reference. We can only keep statistics or summaries over the history of the stream. For classification, this could mean the distribution of an attribute per instance class. These statistics can be updated constantly and incrementally without an increase in memory, as the data stream continually delivers new data. A clever use of these statistics allows us to build more sophisticated models without explicitly resorting to stored instances.

The last property follows from the nature of the data streams: Since they have no end and we cannot wait for them to finish, the model has to be ready at any time during the lifetime of the stream to produce results, e. g. to classify an instance. There is no clear separation between training and test/production phases since there is always new data we want to incorporate. We also want the best possible model we can generate from the data seen so far. We should also expect an increase in model performance over time, since the model gets better as more data is used in the model.

All four of these requirements depend at least to some degree on each other. For example, if we expend some more memory for our algorithm we can store a certain number of data instances and process those together. This technique is sometimes

called mini-batching (cf. Section 2.4.2). On the one side, this can save processing time, on the other side it means our model is not up-to-date for the time between the mini-batch-updates. Those different components have to be balanced against each other.

The same is true for hardware and software. Hardware, processing engine and algorithm design have to fit together for the best results. For example, we can run an algorithm that supports a high degree of parallelization on hardware that is optimized for such use or run a very fast, streamlined algorithm on cheap hardware physically integrated with or near the data source to get immediate results.

There exists a large selection of both supervised and unsupervised online algorithms for different purposes. Count-Min sketch [31] allows to maintain a frequency table of events, CobWeb [42] is an online clustering technique. Stochastic gradient descent [23] is an adaptation of the gradient descent optimization, FIMTDD [60] allows online regression, to name only a few of the better established ones.

In this thesis, we will discuss two more classes of online algorithms: Hoeffding Trees for stream classification in Chapter 4 and dependency monitors in Chapters 5 and 6.

2.3 Notation in this Thesis

For the purpose of this thesis, we call a data set a collection of data streams that originate from sources, usually some sort of sensor, in the same system. The individual streams consist of a series of measurements as in the time-series model together with a time stamp. The measurements may be values for the same characteristic, e.g. temperature, recorded at different physical source locations, or different characteristics, e.g. temperature and humidity, at the same physical location, or a combination of the two. The measurements can be represented as rational numbers without unit. The terms measurement and value are largely interchangeably, but we will use *measurement* when we talk about the underlying physical phenomenon and *value* when we talk about the more abstract processing, for example as input or instance in a classification algorithm.

We use $\mathbf{X} = (\mathbf{x_1}^T, \mathbf{x_2}^T, \ldots, \mathbf{x_d}^T)$ to identify a data set with d streams and $\mathbf{X^t}$ for values at time t.[2] This is somewhat similar to matrix notation for static batch data sets but since a data stream has no defined start or end, the full matrix cannot ever be realized, it actually doesn't fully exist at any time. It cannot be realized because it has theoretically infinite size, and it is never complete because new data is created

[2]Please note that, following the usual convention, the capital ^{T} always indicates a transposed vector. A lowercase ^{t} on the other hand always indicates an index of time.

and is continuously arriving. We cannot access this future data and we can store only a limited amount of historic data. A row $\mathbf{X^t}$ of the data set contains the values from all streams at one particular time stamp. Such a row is also called a tuple of the data stream. In the context of a learning algorithm, we also speak of rows as the instances of the data set.

We can select a window of all values of the data streams between time t_1 and t_2 as $\mathbf{X^{t_1,t_2}} = (\mathbf{X}^{t_1}, \mathbf{X}^{t_1+1}, \dots, \mathbf{X}^{t_2})$ (with each $\mathbf{X}^{t_i}$ a row of $\mathbf{X}^{t_1,t_2}$) or for a single stream as $\mathbf{x}^{t_1,t_2} = (x_i^{t_1}, x_i^{t_1+1}, \dots, x_i^{t_2})^T$. In reality the time stamps of measurements may not be perfectly synchronized due to technical imprecision. In this case, the timestamps in a row of $\mathbf{X}$ might not coincide absolutely but they are assigned the same index in the sequence. For example, measurements at two locations every 5 seconds might result in timestamps of $(0.001, 4.992, 10.004, \dots)$ and $(0.002, 5.002, 10.00, \dots)$. We can treat them slightly idealized as parallel measurements at times $(0, 5, 10, \dots)$ regardless of the small imprecision. For the purpose of this thesis we will assume that data streams are complete, i.e. have no missing values, and timestamps coincide. Timestamps then function in the same way as sequence indices.

2.4 Streaming Engines

Since volume and speed of streaming data in real world applications brings traditional computational paradigms to their limits, specialized stream processing frameworks have emerged. They often aim specifically towards the real-time processing of high volume data [93], usually based on an architecture that allows distributed and parallel computation. They offer different levels of performance and features – in terms of guarantees, fault tolerance, latency or throughput – and a way to implement applications more user-friendly and with more high-level code than in non-streaming frameworks. We will present an overview of three of the more mature and popular engines, Apache Storm, Apache Spark and Google Dataflow. The code written for this thesis has been implemented mostly in Python and is compatible to PySpark, the python interface of Apache Spark Streaming.

Table 2.2 summarizes some of the differences between traditional Data Base Management Systems and Data Stream Management Systems. They follow directly from the properties of streaming data we laid out in this chapter. Queries are no longer asked once but over the lifetime of a stream. The streamed data can only be accessed as it comes in and not in a random order, since we do not (completely) store the data at all. We also cannot rely on precomputed results since the characteristics of the stream might change over time. Data Streams Management Systems and Stream engines are designed with these characteristics in mind.

Table 2.2: Comparison between Database Management Systems and Data Stream Management Systems. Taken from [45].

Data Base Management Systems	Data Streams Management Systems
One-time queries	Continuous queries
Random access	Sequential access
Persistent relations	Transient streams (and persistent relations)
Access plan determined by query processor and physical DB design	Unpredictable data characteristics and arrival patterns

2.4.1 Apache Storm

Apache Storm [8] was first released in 2011 and attracted great interest. It was used in the technology stack of Twitter and Spotify, popular social media applications with huge data traffic. Storm defines a topology of "Spouts" that connect to external data sources and "Bolts" that act as computational nodes. Spouts and Bolts can be connected in a directed acyclic graph to emit one or more result streams (see Figure 2.3).

Storms API has a native streaming model (as opposed to micro-batching) meaning a stream tuple is processed without abstraction on top. This allows very low latency but limits throughput and makes failure recovery more expensive. Storm guarantees at-least-once processing where a tuple is sent again after a failed computation. That way, nodes higher in the topology above a failure or earlier in the processing pipeline might receive a tuple more than once.

Storm is one of the earliest streaming engines with significant popularity and has been considered as a de-facto standard. [84] For a higher abstraction, Storm offers an alternative interface called Trident. It expands the capabilities of Storm for stateful operations, similar but less developed to the feature set of Apache Spark, which we will discuss in the next section.

2.4.2 Apache Spark

Apache Spark [7] is a framework for cluster computing designed for fault tolerance and parallelization. The included streaming module, called Spark Streaming [100], is built on top of the main Spark engine. Spark provides RDDs (resilient distributed data sets) an abstraction on the input data which may be distributed over several machines. They also provide fault tolerance since an RDD contains information about

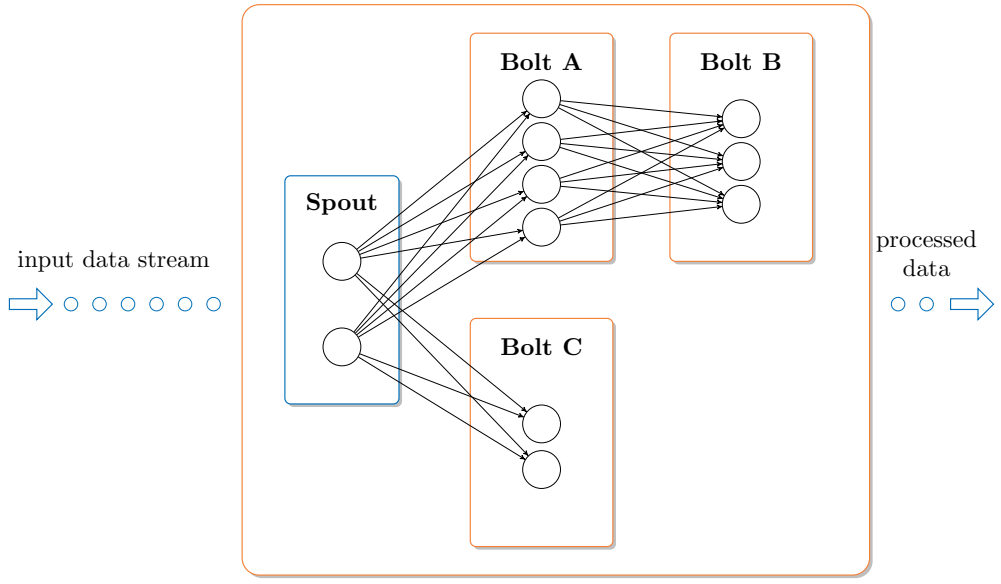


Figure 2.3: Apache Storm Topology with Spouts and Bolts as part of the stream processing. The spouts are sources of data, the bolts form different processing pipelines and transform the stream data, e.g. filter or aggregate it. Data is processed iteratively in single tuples, which allows low latency but limits throughput.

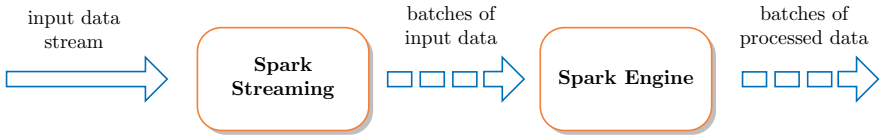


Figure 2.4: Micro-batched stream processing in the Apache Spark Streaming framework. Small sets of data are passed on to the Spark engine and processed together, which allows higher throughput at the cost of higher latency.

its creation, so this and only this RDD can be re-created in case of failure.

Spark Streaming introduces "discretized streams", stream processing as a series of batch computations on small time intervals. This is sometimes also called micro-batching (cf. Figure 2.4). In each interval, the data is processed and the output stored as RDD. Combining RDDs together with micro-batches makes failure recovery comparatively cheap. Spark therefore guarantees exactly-once processing where a tuple is recovered by recomputing the RDD after a failed computation.

The micro-batches also allow a higher throughput compared to single tuple stream like the Storm model [28] but cause much lower latency due to the coarse-grained intervals.

Stream analysis with Spark is also made easier since code written for batch-style analysis is more easily re-usable. The API even provides windowing operations as further abstractions on the stream.

As a measure for popularity, it may be interesting to note that as of December 2016, the Apache Storm repository on Github lists almost 250 contributors, while the Apache Spark project has over 1,000 contributors.

2.4.3 Google Cloud Dataflow

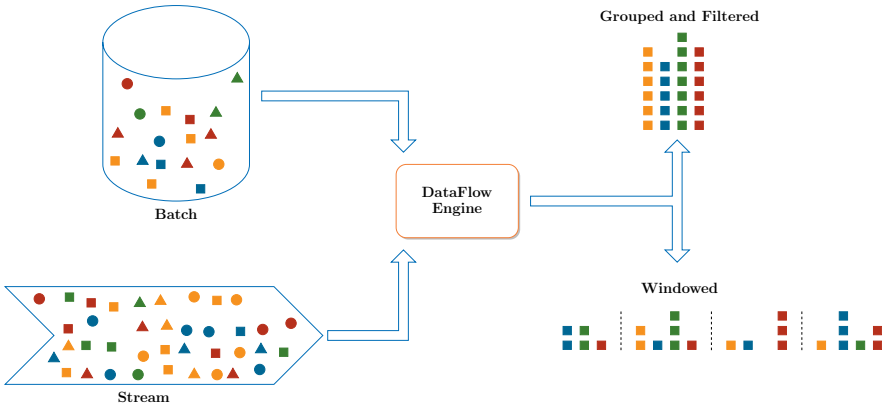


Figure 2.5: Batch and stream processing in the Google Dataflow model. Data may come as batch or from (potentially unaligned) data streams. It can be aggregated (for example filtered and grouped) or processed in windows over time.

A recent approach to stream analytics is the Cloud Dataflow Model by Google. [53] It descends from Flume [25] and Millwheel [3], two lesser known frameworks for distributed and streaming computation and aims towards data processing on massive datasets. [4] We mention it here as an example for a more service-oriented framework.

It concentrates on a windowing model supporting unaligned, i.e. phase-shifted, windows and out-of-order processing. This allows low latency and completeness even in the presence of late-arriving data.

As a framework, it also includes capabilities for data ETL (the process of extracting data from heterogeneous sources, transforming it in a suitable data format and loading it in a database), batch and real-time streaming analysis (cf. Figure 2.5). It has therefore the potential and is designed to replace both Hadoop and MapReduce frameworks for large scale analytic tasks. It is flexible enough to build a processing analysis pipeline at high data volume suited for use cases that balance completeness and latency. Unfortunately, so far it is not available as open source project but integrated in Google's Cloud Service platform.

2.5 Stream Data Sources

We use a variety of stream data sets for the experiments in this thesis. The data sets themselves are freely and publicly available from online repositories or databases. We use them to create our own reproducible data streams. A description of the data sets and their origins can be found in the respective chapters where they are first used.

All the streaming engines mentioned above can connect to a selection of data sources, like listening at a web socket, log files on a file system, or a dedicated streaming platform like Apache Kafka [6] and other stream managers. That way, we can feed our data into a stream process. However, they introduce a new class of sources for potential errors and deviation. Ultimately, it is our goal to test our designs independent from a particular streaming engine. Therefore, and to make experimental results truly reproducible, we created our own small Python streaming suite, where data is read from a hard drive and transformed into a constant stream. This was also the best way to ensure fixed window sizes for windowed applications and to guarantee independence from the processing environment.

Chapter 3

Entropy Analysis

In many practical scenarios, we face an analysis task where labels are unavailable. However, the research into unsupervised learning has developed techniques, such as clustering and anomaly detection, that deal with problems without labels. An essential element of unsupervised techniques is an alternative criterion beyond the labels which allows us to find structure and patterns in our data.

Entropy is a measure for the predictability and order within data and thereby a very powerful and versatile tool to infer structure in the data. It provides a criterion to separate random configurations and noise from structured behaviour. Even without prior knowledge of what to expect from our data, it allows us to separate interesting and useful parts from less meaningful ones. In this chapter, we will introduce entropy based on Shannon-information theory and the closely related concept of mutual information.

Concepts and quantities from information theory appear as answers to fundamental problems in statistics, data compression and transmission, hypothesis testing, probability theory and many more. To cite one example from machine learning: One of the earliest decision trees ID3 and many related ones use the information gain, the decrease in entropy, as the split criterion.[80, 79] Their properties also make them useful tools in the task of data monitoring. "Elements of Information Theory" by Cover and Thomas [33] can serve as a comprehensible and much more comprehensive introduction for the topic. An interesting alternative introduction that focuses on visual representations can be found by Olah [74].

3.1 Shannon-Entropy and Differential Entropy

Entropy, broadly speaking, is a measure for the predictability or conversely the uncertainty inherent in data, say a discrete random variable X with a known probability mass function (pmf) $p(x)$.

Definition 3.1 *The entropy H of X with probability mass function (pmf) $Pr(X = x) = p(x)$ is defined as*

$$H(X) = \sum_{x \in X} p(x) \log \left(\frac{1}{p(x)} \right). \tag{3.1}$$

It is often described as a measure for order or disorder in closed systems, appears as crucial quantity in our understanding of physics and chemistry, but is also useful for comparing written text [87] or even study the expressiveness of gestures and body movements [75].

The definition given above has originally been derived by Claude Shannon [86] following some simple requirements for the entropy function. It also arises naturally, for example as lower bound in data compression problems, and it is related to the definition of entropy in thermodynamics. For a strict derivation see [33, Chapter 2]. The choice of base for the logarithm is somewhat arbitrary. We will use the natural logarithm in the following. The unit for entropy in this case is called 'nat', for the binary logarithm it would be called 'bit'.

To give an example, let X be a discrete random variable over two types of person we might meet, $\{$*'witch'*, *'not a witch'*$\}$. Since witches are fairly rare, with $p($*'witch'*$) = 0.01; p($*'not a witch'*$) = 0.99$, we can be quite sure that any person we randomly meet is not a witch. The entropy of X is $H(X) \approx 0.056$ nats. In a world without witches, there is no surprise in the question and the entropy is appropriately $H(X) = 0$ nats[1]. The maximal surprise occurs in a world with equal numbers of witches and non-witches with $H(X) \approx 0.69$ nats.

The definition can be extended to more than one random variable, either jointly as the joint entropy, or one random variable conditioned on the second as conditional entropy. Keeping to the case of two variables, we get the following definitions:

Definition 3.2 *The joint entropy $H(X, Y)$ of random variables X and Y with joint distribution $p(x, y)$ is defined as*

$$H(X, Y) = \sum_{x \in X} \sum_{y \in Y} p(x, y) \log \left(\frac{1}{p(x, y)} \right). \tag{3.2}$$

[1] With the stipulation that $\log(0) \cdot 0 = 0$

Definition 3.3 *The conditional entropy $H(Y|X)$ of random variables X and Y is defined with the probability distribution $p(y|x)$ of Y conditioned on X as*

$$H(Y|X) = \sum_{x \in X} \sum_{y \in Y} p(x, y) \log \left(\frac{1}{p(y|x)} \right). \tag{3.3}$$

These definitions describe a relationship between two random variables. They are obviously more interesting, if the variables are not independent. If this is the case, we can learn something about the state of variable X from the state of variable Y (or vice versa).

To continue our example, we have it on good authority, that witches burn better than the average person, because they are made from wood. Let Y be a discrete random variable over $\{$*'wood'*, *'not wood'*$\}$ If most normal persons are not made out of wood, $p($*'wood'*$|$*'not a witch'*$) = 0.001$, and most witches are, $p($*'wood'*$|$*'witch'*$) = 0.999$, we get the situation we see in Figure 3.1.

The entropy, i.e. the surprise inherent in the question of whether or not someone is made of wood, is ~ 0.06 nats. (It is about as likely as meeting a witch, so we can just assume everyone we meet is not made of wood.) However, the two quantities are related and there is an easy test for woodenness[2], so we can reduce the surprise in meeting a witch further by administering this test. The conditional entropy for $H(X|Y) = 0.003$ is much smaller than $H(X)$ alone, since our test is quite precise.

From the example it should become clear that every additional test can only reduce the uncertainty about the random variable. We could also work from the definitions above and derive that conditioning on another variables never increases entropy:

$$H(X) \geq H(X|Y).$$

There is a direct relation between the three entropies we defined so far. The joint entropy of X and Y can be expressed as the sum of the entropy of X and the entropy remaining in Y after X is taken into account, i.e. Y conditioned on X:

$$H(X, Y) = H(X) + H(Y|X).$$

The connections between the different quantities is easier to understand if we visualize the information content of the variables as in Figure 3.2. While the joint

[2]Wood floats, as do ducks. A person lighter than a duck is made of wood and most likely a witch. We thank Terry Jones for the example.

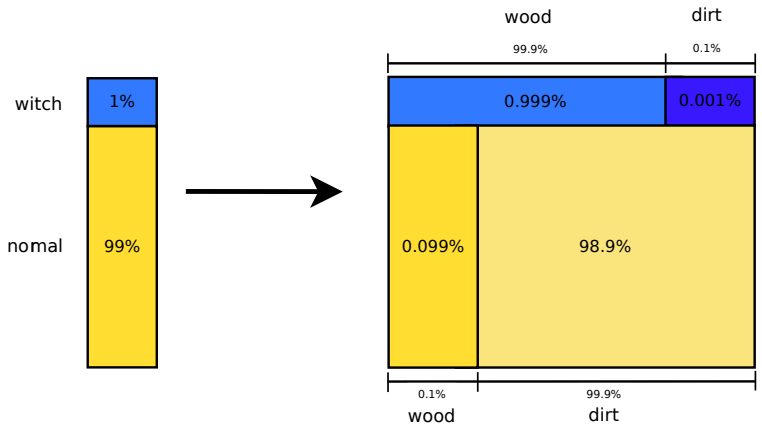


Figure 3.1: The joint probability distribution of witches and woodenness.

entropy is the union of information in both variables and the conditional entropy can be thought as the remaining information once the other variable is fully taken into account, we can also ask for the overlapping or shared part of the information. This mutual part is called the mutual information and can be defined analogous to the previous definitions via the probability mass functions of X and Y.

Definition 3.4 *The mutual information $I(X;X)$ of X and Y is defined as*

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) \log \left(\frac{p(x,y)}{p(x)p(y)} \right). \tag{3.4}$$

Looking at Figure 3.2, we could also derive it from the previous quantities, either by subtracting the joint entropy from both 'singular' entropies, or we decrease one of the singular entropies by the information content it does not share, the conditional entropy:

$$\begin{aligned} I(X;Y) &= H(Y) - H(Y|X) && (3.5)\\ &= H(X) - H(X|Y) && (3.6)\\ &= H(X) - H(X,Y) + H(Y). && (3.7) \end{aligned}$$

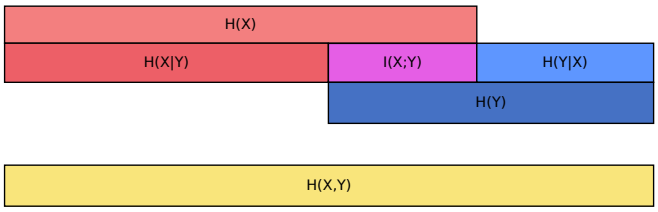


Figure 3.2: The relation between entropy, joint entropy, conditional entropy and mutual information of two discrete random variables X, Y.

The mutual information is, amongst other properties, a symmetric measure for the shared information content. The larger the mutual information, the more knowledge we can transfer from one variable to the other. It can therefore be thought as a measure for mutual predictability, derived from the underlying probability distributions.

In our example, the mutual information between woodenness and witchiness is about 0.052 nats, over 80% of the joint entropy. This not only shows us again that our test for witches is a good one, but that it could also work in reverse: Knowing either that a person we randomly meet is a witch or knowing that they are made of wood, allows a prediction about the other with reasonable certainty.

We can turn this idea around and now claim the following: A high mutual information is an indicator for a strong relationship between two random variables. It can be used as a measure of dependence between them. We will make use of this property later in Chapter 5 and Chapter 6.

Differential Entropy

We presented the case of entropy for discrete random variables so far. The continuous case is largely analogous to the discrete case, with probability density functions (pdf) for the random variables and integrals over the probability space taking the place of the sums.

There are some important differences, most notably we have to take care that the density functions and integrals exist. However, the same intuitions and relationships between entropy, conditional and joint entropy, and mutual information apply to the discrete case of entropy as well as to the continuous. For example, we can even prove that the entropy of a quantization of a continuous variable X in n bins is very close to the differential entropy plus a constant, $h(X) + n$. [33, Theorem 9.3.1]

The continuous form for entropy is called either differential entropy or simply continuous entropy.

Definition 3.5 *The differential entropy h of X with pdf $f(x)$ is defined as*

$$h(X) = \int_S f(x) \log \left(\frac{1}{f(x)} \right) dx, \tag{3.8}$$

where S is the sample space or support set of X

Definition 3.6 *The joint continuous entropy $h(X, Y)$ of random variables X and Y with joint pdf $f(x, y)$ is defined as*

$$h(X, Y) = \int_S f(x, y) \log \left(\frac{1}{f(x, y)} \right) dx\, dy. \tag{3.9}$$

Definition 3.7 *The conditional continuous entropy $h(Y|X)$ of random variables X and Y with $f(y|x)$ as probability distribution of Y conditioned on X is defined as*

$$h(Y|X) = \int_S f(x, y) \log \left(\frac{1}{f(y|x)} \right) dx\, dy. \tag{3.10}$$

Definition 3.8 *The continuous mutual information of continuous random variables X and Y is defined as*

$$I(X; Y) = \int_S f(x, y) \log \left(\frac{f(x, y)}{f(x) f(y)} \right) dx\, dy. \tag{3.11}$$

From the definitions we again get a partition of the mutual information into its entropy components:

$$\begin{aligned} I(X; Y) &= h(Y) - h(Y|X) && (3.12)\\ &= h(X) - h(X|Y) && (3.13)\\ &= h(X) - h(X, Y) + h(Y). && (3.14) \end{aligned}$$

3.2 Correlation and Dependency in Streams

In Chapter 2, we discussed data streams from a technical angle. In practice, they occur if events are observed over a long time and new observations are made available continuously as they arise. This scenario is common in live monitoring through specialized sensors. Important areas are for example the large field of wearable devices that measure physiological functions or track movement and activities for medical and other health purposes. Ecological and geological monitoring of ecosystems, precipitation, erosion, animal behaviour, flooding, etc. create large amounts of data continuously as well. A third area is the supervision of industrial facilities, monitoring power plants, factories or complex machines in great detail with sensors for a variety of parameters.

The connection to online learning and online algorithms appears obvious. We wish to draw conclusions from the new data immediately without having all data available at all times.

One way to monitor a large system is to look for close associations between pairs of parameters. We could achieve this by calculating a measure of dependency for each pair, identifying for example the relationships in the data and tracing changes as the continue.

The best known indicator for pairwise correlation is probably Pearson's correlation coefficient ϱ, essentially the normalized covariance between two random variables. We can calculate ϱ empirically from a sample of two variables as

$$\varrho = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}, \tag{3.15}$$

with sample means $\bar{x}$ and $\bar{y}$. To do so for a pair of streams, we can simply define a window size suitable for our purpose and treat the excerpt of the stream as sample of a random variable. The correlation as measured by Pearson's ϱ is, however, only one kind of dependency that might occur.

3.3 Entropy Measures for Similarity

If we are searching for significant patterns in our data, we naturally will find only those patterns we are actively looking for. The pattern of interest here is dependency or similarity between streams, and the measure we employ to define dependency limits our results from the start. Searching for very specific pattern, it may be the most

sensible approach to employ a specific dependency measure to detect the most relevant streams of interest.

However, to do so, we need to know the data set well enough to make several assumptions with confidence: We need a functional model of the processes we monitor, knowledge of the type of relationships or have to know the distribution of the data. In these cases, the task of detecting dependencies has already been done for us, or is made much easier, since we already have a model available. This last case is more relevant for anomaly detection in settings with stricter limitations. Here, we are interested in a more general monitoring task.

If we knew the type of relationship or the expected distribution, we could choose a specialized measure for the situation. Pearson's or Spearman's correlation coefficients have been designed for linear or monotone relationships, for example.

Many more interesting relationships though do not follow such simple functional forms or can be described in a single functional form at all. Especially non-linearity in time-series has been studied to some extent and may arise for a multitude of reasons. It occurs for example due to shifts in the variance over time. [37] Most likely however, non-linearity simply is found if the underlying processes that generate the data are determined by non-linear functional relationships.

As an example, Figure 3.3 shows twelve two-dimensional data sets with a simple relationship between the x and y-attributes we can instantly recognize visually. The strength of the relationship varies, but the data is clearly not random. A suitable measure for dependency should be able to differentiate these relationships from background noise.

We calculate two measures of dependency for all of these data sets and compare them in Table 3.1. Both measures are normalized between 0 and 1 if they are not already in this range by definition. The mutual information is calculated with the Kraskov-method we will discuss in later chapters. In the clear cases of correlation (subsets a) and e)), MI is slightly lower than ϱ, but still shows clearly a relationship. It is much higher than ϱ for the sets i) - l) with more unusual, non-linear relationships.

Our toy example makes clear, why a different approach than simple correlation analysis is useful to analyze our data. The type of significant signals in the noise cannot always be known in advance, so a measure for dependency should give weight to all sorts of forms of "interesting" behaviour.

Of course, we must not on the other hand fall in the trap of under-fitting and falsely recognize random behaviour as significant. Entropy-based measures have shown to do well as equipartitious general model to find all types of relations we are actually interested in.[82]

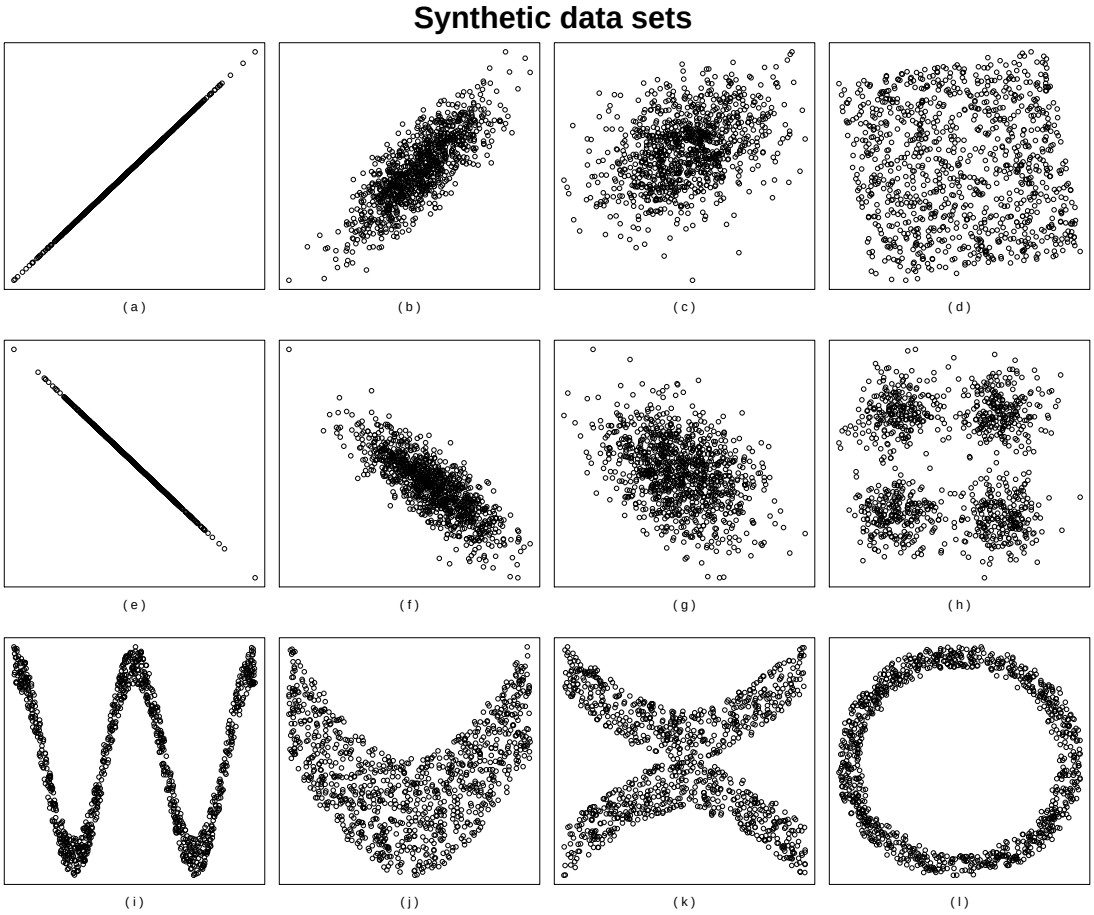


Figure 3.3: Synthetic data sets representing a variety of functional relationships. Mutual information and Pearson's ϱ show different behaviour due to their different definition of dependency.

Table 3.1: Mutual information and Pearson's ϱ on synthetic data sets representing a variety of functional relationships.

Data set	MI	ϱ	Data set	MI	ϱ
(a)	0.99	1.00	(g)	0.05	0.38
(b)	0.13	0.80	(h)	0.03	0.00
(c)	0.06	0.39	(i)	0.33	0.02
(d)	0.06	0.06	(j)	0.13	0.02
(e)	0.98	1.00	(k)	0.15	0.06
(f)	0.14	0.81	(l)	0.23	0.03

The mutual information in particular is a symmetric measure of the mutual predictability of two (or more) random variables, and can be interpreted as a distance measure where distance equates the strength of a relationship. It can be used for variables on different scales and in different domains, and works with small or large sample sizes suitable for practical purposes.

Therefore, we use a variant of the mutual information in streaming analysis, detailed in Chapter 5 and Chapter 6.

Chapter 4

Stream Classification

Stream classification as a topic in machine learning is a sub-class of the common classification task: We want to assign to every instance of a data set one out of several possible labels, based on the attributes of the instance. Stream classification tackles several additional complications compared to the pure classification problem. We operate on a stream of data instead of on a single batch of data and have to follow the requirements of online learning as described previously in Chapter 2 in Section 2.2.

Stream classification is a type of supervised learning where labels are available from the training data. In general, this makes the evaluation and comparison of different algorithms easier, but poses the crucial challenge to generalize towards data that has not seen before during the training. This is especially true with streaming data where not even the training data is fully available at once.

In this chapter, we briefly describe the idea of an online decision tree, a classifier for streaming data, and a novel class of such trees, the Probabilistic Hoeffding Tree. This class is capable of dealing with sudden concept changes in a data stream and provide fast, accurate classification results. Our own contributions in this chapter are published in:

[22] J. Boidol, A. Hapfelmeier, and V. Tresp. Probabilistic Hoeffding trees. In *Industrial Conference on Data Mining, Best Paper Award*, pages 94–108. Springer, 2015

Sections 4.1, 4.2, 4.3 and 4.4 have been taken from the chapters "Online Decision Trees", "Probabilistic Hoeffding Trees", "Experiments" and "Related Work" in the authors publication in [22] and edited to a minor extent. Newly added is Section 4.5.

4.1 Online Decision Trees

In this section, we quickly introduce an elementary notation and review the core steps of constructing an Online Decision Trees. For an in-detail description, we refer to the original publication of Domingos and Hulten [38] and the implementation of VFDT in the MOA framework [16]. We then present our approach PHT (Probabilistic Hoeffding Tree) as extension of those trees and outline how these changes can be implemented in a streaming fashion in the next section.

Frequently, data streams experience gradual or sudden changes in the distribution of input attributes and target label, because external conditions undergo changes that may be periodic and sudden like day and night cycles or incremental like wear and tear. In these cases, algorithms have to adapt to the changing conditions. In reference to the system or concept that generates the data, we call a gradual change *concept drift* and an abrupt and sudden change *concept change*. In this chapter, we consider only the case of concept change, which appears in several variants. Given measurements $\mathbf{x}$ and events $\mathbf{y}$ we discern three main cases. (i) There can be a change in the distribution of attributes $P(\mathbf{x})$, a sampling shift or sometimes called virtual concept change. (ii) There can be a change in the conditional probability of $P(y|x)$ or (iii) there can be changes in both the conditional probabilities and the distribution of attributes at the same time. [59] The more challenging part we will refer to in the following is the case of a changing conditional probability of attributes and class. This requires changes in the model itself while a virtual shift might be addressed with an appropriate sampling during training or classification.

For streaming scenarios, online learners for classification have been developed that should meet the following criteria (cf. Section2.2): Learning should operate iteratively, i. e. build a classification model incrementally without needing all the data before training starts. It must use every record in a single pass, i. e. look at every example only once. It needs to use only finite resources, i. e. the algorithm's training time and space requirements should not grow with the data size. It should exhibit anytime readiness, i. e. provide the best possible classification model at any time during execution.

Hoeffding Tree-based classifiers possess these desirable properties, and remain fairly easy to implement and analyze. They have been shown to be robust, highly scalable, fast and resource-efficient. We will use these tree structures and improve their classification power on data streams that undergo concept changes. We will do this be improving the way, Hoeffding Trees learn from new data they have to adapt to. We point out, that our algorithm design is based on the basic Hoeffding Tree, but is in principal also adaptable to any tree-like learner.

The Hoeffding Tree is a classifier that deals with changing, streaming data[38], also known as VFDT (Very Fast Decision Tree). Many state-of-the-art online classifier build on it, e. g. FIMTDD [59], CVFDT [58], VFDTc [47], iOVFDT [56], Hoeffding Option Trees [77]. Others use it as a component in ensemble algorithms[13].

VFDT and its derivatives incrementally build a decision tree and prune parts again as necessary without looking at any record more than once. Splits in the tree are introduced when sufficient examples have been seen to make a confident decision. This decision is guided by statistical bounds, e. g. the eponymous Hoeffding bound. To do so, we need only sufficient statistics stored in the tree. The nature of these statistics varies but typically allows to calculate the best split on promising attributes. They have a constant space requirement so the size of the tree is limited by the number of leafs, not by the total size of the data set used in training.

Let $A_i = (a_{i,1}, \ldots, a_{i,k})$ be an instance of the data stream with k single-valued attributes where the index i notes the position in the data stream. Like all decision trees, Hoeffding Trees consist of nodes and edges (V, E) where the nodes contain tests to decide which edge to follow towards a leaf of the tree. To build a Hoeffding tree, during the training phase leaf nodes are recursively replaced with decision nodes. The leaf nodes store statistics, decision nodes hold a split attribute and a split value. Each instance is assigned to one leaf node v after a series of decisions that determine the path from the root downwards. These decisions select the appropriate path based on the relevant split attribute of the instance in each node along the path. Thereby they determine the one branch of the tree the instance A_i falls into and the statistics stored in leaf v are updated with the information from A_i. In some versions, statistics in the nodes on the path to v are also updated. A decision to grow or prune the tree is then based on these updated statistics with a statistical bound. The Hoeffding bound for example uses the number n of observations, their mean $\bar{r}$, and the range of the attribute R. With probability $1 - \delta$ the true mean is then at least $\bar{r} - \epsilon$, with

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}. \tag{4.1}$$

The Hoeffding bound in particular is useful because it makes no assumptions on the distribution of the attributes. The unfortunate downside is that the bound is rather conservative and needs to see more instances than other bounds to reach the same guarantees. The bound is also crucial to detect changes in the data stream. If significant changes are detected, the tree is adapted to the new data via pruning and regrowing of leaves or subtrees. [15] We can formulate a simple pseudo code to build an incremental tree as in Algorithm 1 below.

Algorithm 1 Basic Online Tree Induction

Input: data stream s yielding records A_i
Output: decision tree t

```
1: procedure TreeInduction
2:     t ← empty leaf
3:     while A_i ← next_from(s) do
4:         v ← get_leaf(A_i)
5:         update(v, A_i)
6:         test_for_split(v)
7:         prune(t)
8:     end while
9: end procedure
```

In the code and discussion we will only explicitly consider two-way splits for numeric attributes. More branches are obviously possible, and necessary for categorical attributes. However, the conditions for multi-way splits and discrete distributions, i. e. counts for categorical attributes follow naturally in an analogous manner. The tree has no defined stopping criterion or final state. Whenever more instances are available from the stream, it continues to grow and adapt, and can at any time be trained further. The same is true for prediction with the tree. There is no training phase that has to be completed before predictions are possible. At any point in the lifetime of the tree it can be used for training and prediction.

Ordinarily, the prediction for a record A_i is based on whatever model is stored in the leaf to which A_i is assigned. In the simplest case this might be a single class-label or numeric value, more sophisticated versions store specific classification or regression models in the leaves.

4.2 Probabilistic Stream Classification

The main idea in our approach is to treat records not as simple vectors of values. Since the true, exact value for an attribute might be biased through imprecise measurements or inexact transformation and transmissions, the value we receive has an inherent uncertainty. We treat the attributes therefore as a probability density function (PDF) centered around the recorded value instead to address this uncertainty. We call the resulting class of Hoeffding-tree algorithms PHT (Probabilistic Hoeffding Trees).

We replace the single value of a_{ij} with a PDF $p(a_{ij})$ centered around a_{ij}. For numeric attributes a uniform or Gaussian distribution are standard choices, for cate-

gorical attributes any discrete distribution specified over the possible values of a_{ij} is acceptable [27, 89]. The training process is then adapted in the following way: We assume again an initial weight of 1 for every instance A_i. For every test A_i encounters in a node, e. g. $a_{ij} < t_m$, the integrals $w_l = \int_{-\infty}^{t_m} p(a_{ij})\,\mathrm{d}a_{ij}$ and $w_r = \int_{t_m}^{\infty} p(a_{ij})\,\mathrm{d}a_{ij}$ for the left and right branch are calculated. w_l and w_r simply determine, how much of the probability mass of the attribute falls in the left and right branch respectively. The values w_l and w_r are then interpreted as the weight of the branch. A_i follows every branch where w is larger than 0 simultaneously and may reach more than one leaf of the tree (cf. line 4 of Algorithm 2). The relative weight of a leaf v is then $w^{A_i,v} = \prod_{m \in M} w_{I,m}$, the product of all weights along the path to leaf v branching at nodes m, where $I \in \{l, r\}$ determines the branch taken at node m. The statistics in these leaves are then updated with the information from A_i, as in the original case, but down-weighted by $w^{A_i,m}$ (cf. line 6 of Algorithm 2). The total weight of A_i still sums to 1 but it promotes growth in more than a single leaf.

The modifications compared to the base algorithm are again given as pseudo code in Algorithm 2.

Algorithm 2 Incremental Uncertain Tree Induction

Input: data stream s yielding records A_i
Output: decision tree t

```
1:  procedure ProbabilisticTreeInduction
2:      t ← empty leaf
3:      while A_i ← next_from(s) do
4:          L ← get_leaves(A_i)
5:          for all v ∈ L do
6:              update(v, A_i, rel_weight(A_i, v))
7:              test_and_split(v)
8:          end for
9:          prune(t)
10:     end while
11: end procedure
```

We treat instances for prediction the same way as in training, see the modifications to the prediction process in algorithm 3. We do not need to change the prediction model used in the tree, but we do not limit the prediction to one of those models. Our algorithm filters one record down to several leaves instead, and averages the predictions from every leaf weighted by $w^{A_i,v}$.

The voting (cf. line 9 in Algorithm 3) has the additional advantage of giving a confidence for the prediction from which a confidence value can be inferred, even if the base algorithm does not provide one.

Algorithm 3 ProbabilisticTreePrediction

Input: tree t, instance A_i
Output: prediction $\tilde{x}$

```
1:  procedure ProbabilisticTreePrediction
2:      L ← get_leaves(A_i)
3:      V = ∅
4:      for all v ∈ L do
5:          vote ← predict(A_i, v)
6:          weight ← rel_weight(A_i, v)
7:          V = V ∪ (vote, weight)
8:      end for
9:      x̃ ← average(V)
10:     return x̃
11: end procedure
```

Run over a long enough time, the trees generated from vectors of values and of vectors of distributions – the instances during training – will naturally converge. This follows from the fact that the distributions are symmetric and therefore unbiased compared to the point-values. We also have to assume a stationary data stream, that is a stream without concept drift or change.

The advantages can be found in two areas. One, in the greater flexibility this allows during training and prediction. Training is less dependent on the order of the instances, and – as experiments will show – in the speed of the convergence towards the expected optimal tree. Two, if the stream is not stationary, the classifier becomes more reactive to changes in the data stream and faster convergence has an even greater impact on the performance over the lifetime of the stream.

4.2.1 Online Approximation of Density Functions

The PDFs for the attribute values have always been chosen as uniform distribution with mean equal to the original attribute point value and a standard deviation proportional to $(b - a) \times w$. Here a and b are the minimum and maximum values for the attribute that actually appear in the data set and w controls the width of the distribution and the 'fuzziness' of the attribute value. For the categorical attributes, the PDF

has been constructed in such a way that $1 - w$ of the probability mass is placed onto the original value and the rest spread uniformly on the possible attribute values. For the synthetic data set (with numerical attributes only), a and b have been chosen so that $P(x_a \in [a, b]) \geq 0.997$ or approximately within three standard deviations of the mean.

The notation as range of values is closely related to, but here more intuitive, than the standard deviation. If the attribute range is unknown, it can be estimated from the stream for example with a number of algorithms that incrementally calculate the variance of the attribute, e. g. [65]. The ranges follow easily from the variance, for example for uniform distributions $\sigma^2 = \frac{(b-a)^2}{12}$.

Representing the PDF $p(a_{ij})$ is simple if the attribute j is categorical. Then we need only the probability for every possible value of j which has a finite and in practice usually small domain. In principal, numeric attributes could be discretized in a number of bins and treated equivalently [70]. This, however, discards the ordinality of the attribute values, forces multi-way splits and is necessarily low grained. In a simple, non-analytic solution, which has been used for example in [95], the PDF can be represented numerically by storing a set of s sample points drawn from $p(a_{ij})$ which approximates any function with a discrete distribution. Conveniently, this works equally well for numeric and categorical attributes and for all types of distributions. We chose $s = 100$ which provided a balance between approximation quality and performance in our tests.

4.3 Experiments

To test our algorithm we used 4 large data sets collected from sensor readings or network streams and one synthetic data set. The real data sets are all available at the UCI machine learning repository and range from 5k to 580k in size. While these are suitable to gauge the algorithm behavior on real world data, we also use synthetic data to test performance in longer runs. For those experiments we used instance streams of 5 million instances. All test runs have been performed on a PC with an Intel Xeon 1.80GHz CPU, running Linux with a 2.6.32 x86_64 kernel, and with memory limitations set to 64 MB.

4.3.1 Implementation

We adapted three different tree induction algorithm to PHT: HoeffdingAdaptiveTree [15], iOVFDT [56] and HoeffdingOptionTrees [77] to HoeffdingAdaptiveTreePHT, iOVFDTPHT and HoeffdingOptionTreePHT. We implemented all three in the MOA

framework [16], where reference implementations of the aforementioned algorithms exist and the algorithms could easily be integrated.

For the evaluation of the experiments, we recorded accuracy, resulting tree size and training time measured in an interleaved test-then-train setting where every instance is first used for blind testing, and then to train the tree. [15] The standard deviation for each measure is computed over 10 repeated experiments with shuffled data sets or different initialization parameters for the synthetic data set. For the accuracy we use a fading average as described in [49] with a fading factor α of 0.99. The fading average $M_\alpha(i)$ is defined as

$$M_\alpha(i) = \frac{S_\alpha(i)}{N_\alpha(i)} \tag{4.2}$$

$$S_\alpha(i) = I_i + \alpha \times S_\alpha(i-1); \quad S_\alpha(1) = I_1 \tag{4.3}$$

$$N_\alpha(i) = 1 + \alpha \times N_\alpha(i-1); \quad N_\alpha(1) = 1, \tag{4.4}$$

where S_α is the fading sum of observations, N_α the fading increment and $I = 1$ for a correct prediction, 0 otherwise.

We report tree size in number of nodes rather than model size in bytes. The consumed memory depends not only on the implementation but also on the number and types of attributes in a data set. The number of nodes on the other hand allows an easier comparison of different tree models. We test our algorithms first on the static data sets to establish their performance and advance to time changing data streams in the following sections.

4.3.2 Data Sets

The data sets for our experiments are collections of sensor and other streams from different types of sources. Table 4.1 summarizes their characteristics.

Robot Movement Data (RM)

The RM data set is available since 2010. It contains 24 numeric attributes recorded from the a robot's sensors and four distinct classes, which determine the robot's course along a wall. The data set contains 5,456 instances [44].

Person Activity Analysis (PA)

The PA data set is available since 2010. It recorded the instances collected from four sensors placed on both ankles, belt and chest of five people. Each instance has five

numeric attributes, two categorical attributes and one of eleven classes. The classes distinguish human activities, e. g. walking, standing, falling, etc. The data set contains 164,860 instances [62].

Network Attack Detection (NA)

The NA data set has been published for the KDD CUP 1999. It describes network connections and is used to classify normal and abnormal connections, i. e. attacks. It contains 34 numeric and seven categorical attributes like duration, error rate and protocol type. The connection types are distinguished in 23 distinct classes. We use 10% of the full data with 494,021 instances [92].

Cover Type (CT)

The CT data set is available since 1999. It collects surveillance sensor data of forest-land. Each instance provides 42 categorical attributes and eleven numeric attributes like soil type, elevation, and hill shade. It distinguishes cover types in seven classes. The data set contains 581,012 instances [17].

Synthetic RBF Stream (RBF)

This type of synthetic stream uses a radial basis function to generate arbitrarily large data sets. Using different initialization parameters we can create different streams, each of arbitrary length. The streams for the experiments were initiated with fixed seeds to ensure reproducibility. We set the parameters to use 50 base functions that generate 15 attributes and 4 classes and limited stream size to five million instances.

Table 4.1: Data sets for the classification experiments with length n and dimension d of the data sets.

Data set description	**d**	**n**
Robot Movement (motion tracking data)	25	5,456
Network Attack Detection (network traffic data)	42	49,402
Personal activity (motion tracking data)	7	164,860
Cover Type (environmental data)	54	581,012
RBF Stream (synthetic data)	20	5,000,000

4.3.3 Classification on Static Data Streams

We implemented as PHT variants the following classifiers: HoeffdingAdaptivePHT, iOVFDTPHT and HoeffdingOptionPHT. We tested these on the five large data sets described in Section 4.3.2 and varied the values for the width w of the assumed distribution from 0 to 0.5. $w = 0$ means no uncertainty and is equivalent to the base classifiers our algorithms build upon. In general, we see an improvement for $w \leq 0.1$, with small to moderate (3.3%) improvement of accuracy. Accuracy drops for larger values of w that would imply major uncertainty and are not listed here.

There is not one base classifier that outperforms the others over all data sets. Taking the best performing setting for each classifier and data set, we see an improvement in 10 out if 15 cases (each significant with $p < 0.1$, in a one sided t-test) in Table 4.3. Figure 4.1 shows the final accuracy for the smallest (RM) and the largest (CT) UCI data set. The fading accuracy used gives less weight to the earlier test examples, giving an overall accuracy that favors the recent predictions.

Figure 4.2 shows the accuracy during the lifetime of the data stream of the RBF data set. We used the RBF data set to analyze the behavior of the algorithms on much longer lived data streams and see improvement over the base classifiers, especially for the Hoeffding Adaptive Tree.

The tree size on average stays within two nodes of the base classifier, with a few exceptions on the larger data sets. There is no clear correlation between changes in model size and improved performance, with five of the eight improved models being smaller, three larger than the base classifier. Tree size does, on the other hand, decrease slightly with increasing values of w since a flatter distribution makes splits in the tree less likely.

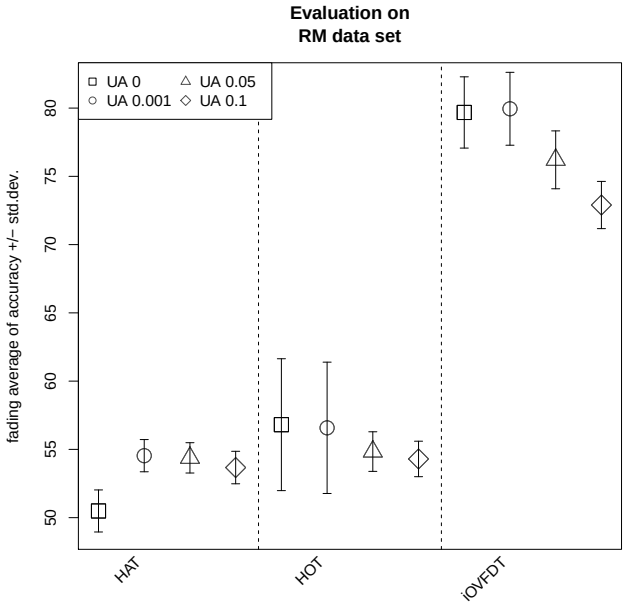
The most interesting observation here is, how even very small values for w can improve the classification without major cost to the model. Running time for the best performing models with $w \neq 0$ stays within a factor of 2 to the runtime of the base classifier. Our algorithms (with $w \leq 10\%$) hold equal to or considerably outperform the base algorithms.

Table 4.2: Classification accuracy on 4 UCI data sets with different width-options for the attribute PDFs. UA00 is equivalent to the base algorithm, UAmin to a PDF with 0.1% width of the attribute range, UA05 to a width of 5% of the attribute range, etc..

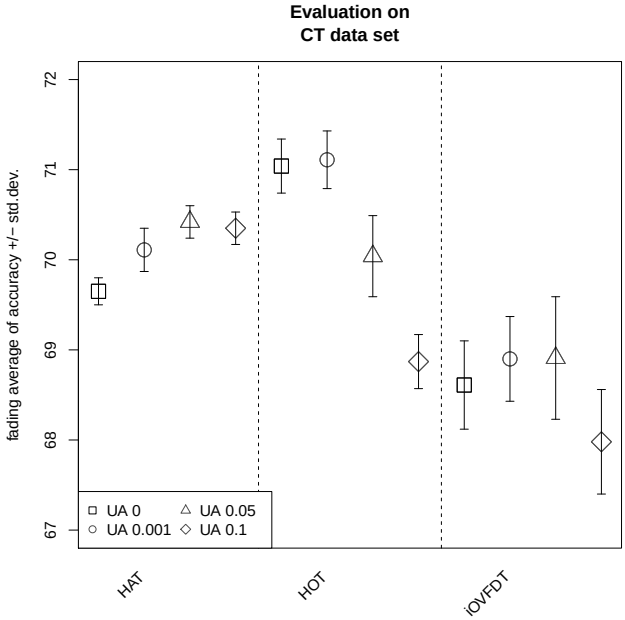
Accuracy in %						
RM data set						
	HOTPHT		HATPHT		iOVFDTPHT	
UA00	**56.81**	± 4.83	50.49	± 1.54	79.68	± 2.61
UAmin	56.58	± 4.81	**54.54**	± 1.18	**79.95**	± 2.67
UA05	54.84	± 1.45	54.38	± 1.11	76.21	± 2.12
UA10	54.30	± 1.30	53.67	± 1.19	72.90	± 1.73
PA data set						
UA00	50.12	± 0.29	46.84	± 0.15	**39.43**	± 0.22
UAmin	**50.21**	± 0.28	**48.10**	± 0.44	39.35	± 0.34
UA05	49.80	± 0.36	47.90	± 0.53	39.22	± 0.31
UA10	48.75	± 0.21	47.19	± 0.66	38.40	± 0.43
NA data set						
UA00	**99.70**	± 0.04	98.34	± 0.02	**98.89**	± 0.17
UAmin	99.38	± 0.17	99.08	± 0.23	98.79	± 0.15
UA05	99.35	± 0.17	**99.11**	± 0.20	98.84	± 0.15
UA10	99.04	± 0.15	98.75	± 0.38	98.41	± 0.22
CT data set						
UA00	71.04	± 0.30	69.65	± 0.15	68.61	± 0.49
UAmin	**71.11**	± 0.32	70.11	± 0.24	68.90	± 0.47
UA05	70.04	± 0.45	**70.42**	± 0.18	**68.91**	± 0.68
UA10	68.87	± 0.30	70.35	± 0.18	67.98	± 0.58
RBF data set						
UA00	91.73	± 1.21	91.72	± 0.04	**88.68**	± 5.21
UAmin	92.14	± 1.70	**93.33**	± 0.02	83.75	± 4.55
UA05	**92.89**	± 1.49	92.52	± 0.03	84.29	± 0.42
UA10	92.45	± 1.86	90.85	± 0.07	79.62	± 0.08

Table 4.3: Tree size of final tree on 4 UCI data sets with different width-options for the attribute PDFs. UA00 is equivalent to the base algorithm, UAmin to a PDF with 0.1% width of the attribute range, UA05 to a width of 5% of the attribute range, etc..

Tree size in #nodes						
RM data set						
	HOTPHT		HATPHT		iOVFDTPHT	
UA00	0.60	± 1.26	0.00	± 0.00	5.90	± 0.57
UAmin	0.60	± 1.26	0.00	± 0.00	6.10	± 0.74
UA05	0.20	± 0.63	0.00	± 0.00	6.90	± 0.88
UA10	0.40	± 0.84	0.00	± 0.00	6.80	± 0.63
PA data set						
UA00	3.30	± 0.48	4.00	± 0.00	3.80	± 0.42
UAmin	3.20	± 0.42	3.20	± 0.42	3.90	± 0.32
UA05	3.10	± 0.32	3.40	± 0.52	3.90	± 0.32
UA10	3.00	± 0.00	3.10	± 0.57	3.70	± 0.48
NA data set						
UA00	4.60	± 1.26	2.10	± 0.32	3.80	± 0.92
UAmin	3.70	± 0.48	4.30	± 0.67		
UA05	3.20	± 0.42	4.20	± 0.42	3.60	± 0.70
UA10	6.50	± 1.18	3.70	± 0.67	5.20	± 0.42
CT data set						
UA00	10.20	± 1.40	8.33	± 1.12	6.30	± 1.16
UAmin	10.00	± 1.70	13.00	± 1.94	7.30	± 0.82
UA05	8.90	± 0.74	9.56	± 0.73	7.60	± 1.07
UA10	7.80	± 0.63	9.78	± 1.39	7.60	± 0.84
RBF data set						
UA00	22.00	± 7.07	27.00	± 0.00	9.50	± 0.71
UAmin	22.00	± 2.83	26.00	± 0.00	10.00	± 0.00
UA05	23.50	± 12.02	32.00	± 0.00	9.50	± 0.71
UA10	21.00	± 7.07	26.00	± 0.00	8.50	± 0.71

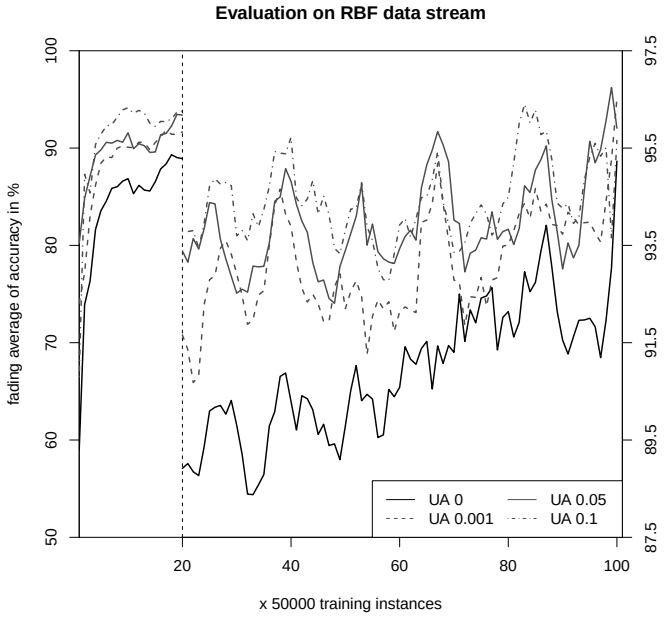


(a) RM data set with 5k instances.

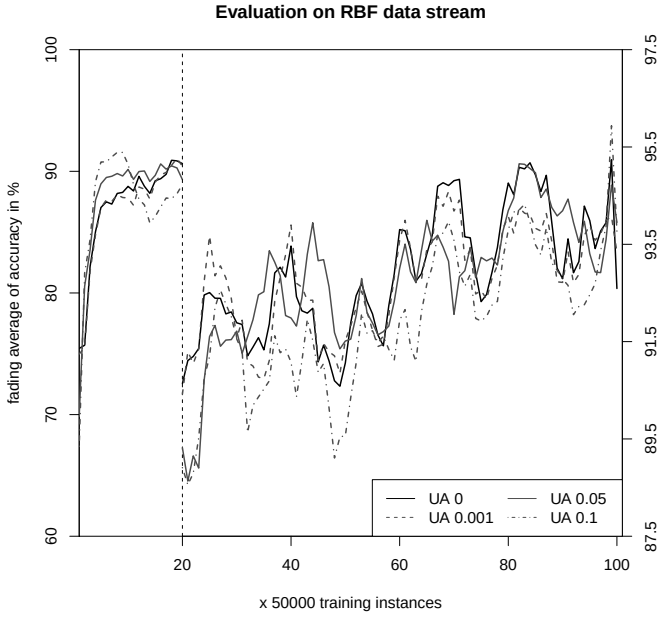


(b) CT data set with 580k instances.

Figure 4.1: Comparison of different flavors of Hoeffding-Tree based classifiers (iOVFDTPHT (iOVFDTPHT), HoeffdingOptionTreePHT (HOTPHT), HoeffdingAdaptiveTree (HATPHT)) on large UCI data sets. Shown here are only the smallest and the largest of the used data sets. UA gives the width-parameter of the PDF replacing the attribute values. Accuracy is the accuracy with a fading factor of 99%. The standard deviation is calculated from 10 shuffled runs.



(a) HoeffdingAdaptiveTreePHT on RBF data stream with 5M instances.



(b) HoeffdingOptionTreePHT on RBF data stream with 5M instances.

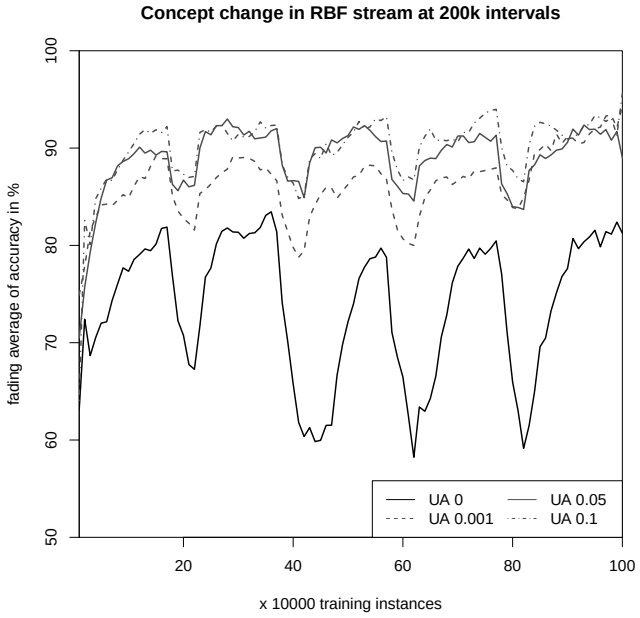
Figure 4.2: Evaluation on RBF stream. The plots shows the accuracy over a the lifetime of a stream with 5 million instances exemplary for the Hoeffding Adaptive Trees and Hoeffding Option Trees. The vertical axis is enlarged from the 20%-mark on.

4.3.4 Classification on Changing Data Streams

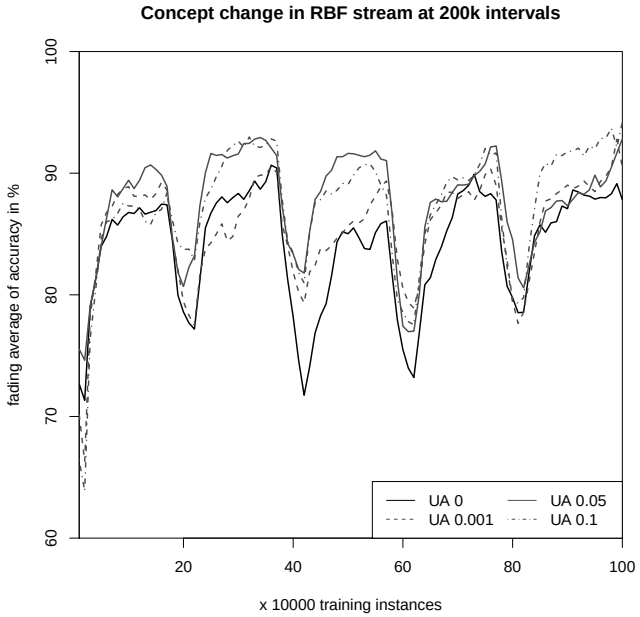
While stationary streams are much easier to deal with, we expect both gradual and sudden changes in real life streams. We therefore especially studied the effects of concept change, i. e. changes in the conditional probability of the classes given attribute vectors, and the improvements our algorithm achieves in such a setting. This occurs if an observed stream/the underlying system undergoes different phases in its lifetime like seasonal changes, day-night cycles in ecological systems, or different production phases in industrial machinery. Normal behavior might look completely different before and after these changes and the classification algorithm has to adapt accordingly. While HoeffdingAdaptiveTrees and HoeffdingOptionTrees have the capability to detect changes and adapt, iOVFDT does not have a mechanism to adapt to dynamic streams and is not included in this section.

To evaluate the effect of changes for the other classifiers, we streamed each of the original UCI data sets in the stream five times in a row, but at each repetition switched the order of the numeric attributes as suggested by e.g.[97, 99, 57]. This permutation of attributes induces the desired concept changes and at the same time keeps the integrity of the data set compared to, say, introducing bias or noise into the data. In the RBF data set we simulated such changes by changing the parameters of the generating function. Figure 4.3 shows the classification accuracy during the lifetime of a RBF-stream with concept changes every 200.000 records.

To compare the accuracy over the total lifetime, we averaged the accuracy over 100 sample points during the stream life time and report the results in Table 4.4. After every concept change, all classifiers fall back in accuracy and recover gradually, but our algorithms recovers at a much faster rate and in this setting of changing streams significantly ($p < 0.1$ in a one-sided t-test for the best-performing setting) outperforms the base classifiers. Figure 4.3 and Figure 4.4 show how accuracy behaves before and after the concept change. For the smaller data set shown in Figure 4.4a the breakdown between change is less pronounced since the classifier has not reached a stable plateau before the concept change as in the longer-lived streams. Our algorithm improves the results of the base classifiers by up to 16% with an average improvement of 3.2%. We improve over HoeffdingOptionTree in three out of five data sets and over HoeffdingAdaptiveTree in five out of five data sets with no significant increase in model size.

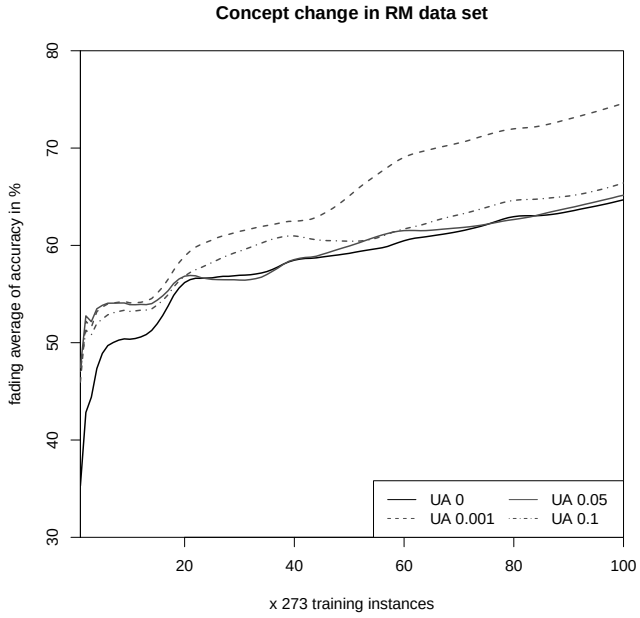


(a) HoeffdingAdaptiveTreePHT

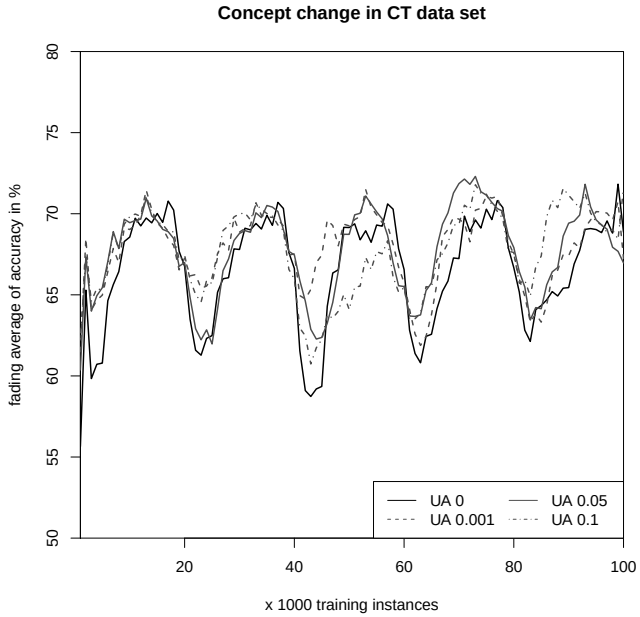


(b) HoeffdingOptionTreePHT

Figure 4.3: Effect of concept changes simulated with RBF stream. UA gives the width-parameter of the PDF replacing the attribute values. Accuracy is the accuracy with a fading factor of 99%.



(a) HoeffdingAdaptiveTreePHT



(b) HoeffdingAdaptiveTreePHT

Figure 4.4: Effect of concept changes simulated with UCI data sets. UA gives the width-parameter of the PDF replacing the attribute values. Accuracy is the accuracy with a fading factor of 99%.

Table 4.4: Accuracy and model size on the data sets with 4 induced concept changes. UA00 is equivalent to the base algorithm, UAmin to a PDF 0.1% width of the attribute range, UA05 to a width of 5% of the attribute range, etc.. Accuracy is the average of the accuracies at 100 sample points during the stream existence. Model size is the number of nodes in the final tree structure.

RM data set $\times 5$								
	Accuracy in %				Tree size in #nodes			
	HOTPHT		HATPHT		HOTPHT		HATPHT	
UA00	**65.43**	± 4.07	58.65	± 0.97	8.30	± 1.06	5.70	± 0.67
UAmin	65.09	± 4.23	**67.16**	± 1.73	8.50	± 0.97	5.40	± 1.65
UA05	60.42	± 0.91	62.23	± 3.78	6.70	± 0.82	8.70	± 2.58
UA10	58.31	± 0.61	63.62	± 3.55	6.70	± 0.67	6.50	± 2.55
PA data set $\times 5$								
	Accuracy in %				Tree size in #nodes			
UA00	**44.66**	± 0.89	43.11	± 0.48	1.00	± 0.00	1.10	± 0.32
UAmin	44.66	± 0.91	**45.22**	± 0.33	1.00	± 0.00	1.40	± 0.52
UA05	44.34	± 0.94	45.05	± 0.32	1.10	± 0.32	1.20	± 0.42
UA10	43.81	± 0.94	44.60	± 0.27	1.10	± 0.32	1.20	± 0.42
NA data set $\times 5$								
	Accuracy in %				Tree size in #nodes			
UA00	97.65	± 0.58	97.61	± 0.21	3.90	± 0.57	2.70	± 0.48
UAmin	97.72	± 0.56	98.31	± 0.51	3.00	± 0.00	1.00	± 0.00
UA05	97.97	± 0.46	98.63	± 0.28	3.60	± 0.52	1.00	± 0.00
UA10	**97.99**	± 0.45	**98.77**	± 0.27	3.40	± 0.52	1.00	± 0.00
CT data set $\times 5$								
	Accuracy in %				Tree size in #nodes			
UA00	63.81	± 1.12	65.81	± 0.50	8.90	± 0.57	2.50	± 0.53
UAmin	63.67	± 1.10	**67.45**	± 0.46	9.00	± 0.67	3.20	± 0.42
UA05	63.59	± 0.95	67.23	± 0.32	8.60	± 1.17	3.00	± 0.00
UA10	**64.41**	± 1.04	67.05	± 0.52	9.10	± 1.20	2.90	± 0.32
RBF data set $\times 5$								
	Accuracy in %				Tree size in #nodes			
UA00	84.70	± 1.16	73.52	± 2.40	12.50	± 0.58	9.40	± 2.70
UAmin	85.62	± 0.69	85.99	± 0.37	13.25	± 1.26	9.80	± 0.84
UA05	87.15	± 0.41	88.61	± 0.70	13.50	± 1.00	10.00	± 0.00
UA10	**87.18**	± 0.98	**89.71**	± 0.70	13.00	± 0.82	10.20	± 0.45

4.4 Related Work

In the last years, substantial research effort has been put into stream analysis and stream processing. Besides the more technical approaches like the streaming engines we discussed in Chapter 2, batch procedures have been adapted to the world of data streams and novel algorithms have been designed to power stream analytics.

The earliest stream classification with iterative tree models has been developed by [38] and built upon by others, for example in [13], [47], [56], [58], [77]. [78] used an uncertainty-aware approach to improve classification models on static data, [70] used a similar approach for online stream-classification.

The area of uncertainty-aware designs overlaps with research into fuzzy algorithm design and covers topics from clustering uncertain data [30, 69, 73] or outlier detection [2] to querying probabilistic databases [68]. Classification with tree models for uncertain data has been done by e. g. [95].

The common idea in these approaches is that the expected distance between two or weights of objects are calculated from probability distributions. These distributions are inherent or derived from the data. Our work in this chapter takes concepts from the research into uncertainty and applies them to classification of streaming data. Uncertainty is not necessarily a quality of the data and acceptable performance in e. g. classification tasks can be reached without addressing it. But even in these cases, we can use it as a tool to extract more information from the data and enhance prediction.

We continue their work and extend the analysis to cases with time-changing data streams. To the best of our knowledge, we have presented the first Uncertainty-aware classifier for data streams with concept change.

4.5 Summary

In this chapter we have shown a generic approach that extends stream classification models to incorporate the concept of uncertain data. We tested this approach on several classifiers with tree models. On five sensor data sets, we achieved significantly improved accuracy with comparable model size and runtime across all the data sets and classifiers we examined.

In the case of data sets with concept change we improve accuracy by up to 16% with 3.2% on average. Our approach reacts swiftly to changing data streams which makes it especially suited to environments where the concept generating the streams changes periodically, as is the case in many industrial or ecological applications. Non-synthetic data where the data quality is quantified, i. e. the actual uncertainty of

measured values is known appears not to be available at the moment. If such data sets become accessible, we expect much more interesting results if we could substitute empirical values for the idealized PDFs. Also, we believe that the approach of uncertainty-aware data can be broadened to other types of algorithms, not limited to classification or tree-like prediction models.

Chapter 5

Dependency Monitoring

In the previous chapter, we discussed the application of machine learning to a labelled data stream, an example of online classification. In this chapter, we will extend stream analysis to settings with a differently defined task. In contrast to the previous supervised learning task, we have no error or loss function to minimize. Instead we want to infer hidden patterns from the data.

We will present several techniques to monitor dependencies between data streams. The reason we look for dependencies is the simple assumption that different data streams that behave similarly over time, i. e. show similar, mutually predictable behaviour, indicate interesting subgroups in the whole system. To evaluate our findings, we use synthetic and real data sets and compare our techniques to other algorithms.

Our contributions in this chapter are published in:

[19] J. Boidol and A. Hapfelmeier. Detecting data stream dependencies on high dimensional data. In *The 1st International Conference on Internet of Things and Big Data, IoTBD 2016*, pages 375–382. INSTICC, 2016

[20] J. Boidol and A. Hapfelmeier. Fast mutual information computation for dependency-monitoring on data streams. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2017

Section 5.1 has been compiled from the Sections "Mutual Information" in [19] and "Data Stream Dependency" in [20] with minor supplements. Section 5.2 has been rewritten with material from "Estimation of PDFs" in [19], Section 5.3 is based on Section "DIMID" in [20] with minor supplements. Section 5.4 combines the Sections "Experimental Evaluation" and "Experiments" from [19] and [20]. Section 5.5 has been extended from "Related Work" in [20]. Section 5.6 has been newly added.

Manufacturing of inexpensive sensors and the desire to monitor systems as diverse as vehicles, industrial plants, smart houses, medical instruments or whole ecosystems produce an abundance of data. WSNs (wireless sensor networks) are used in applications like pollution, traffic or water quality monitoring. The data is created by those sensors, transmitted to a central station, and the streams of data need to be automatically evaluated and analyzed.

Imagine for example one large production plant with thousands of sensors embedded in machinery and infrastructure, delivering information about the system. There are far too many data sources, i.e. sensors, to examine them manually, but they contain important, sometimes unknown and yet undiscovered information in their relationships. Knowledge of the relationships allows previously hidden insights into the underlying system, and changes in the relationships over time might uncover extraordinary or anomalous events.

Faced with the task of monitoring such a large system, we could concentrate on those variables that exhibit strong relationships between them. We could find those if we calculate a measure of dependency for each pair of all variables, tracing for example changes in the relationships.

A specific practical application of this could be for example the validation of sensor readings in the context of multiple cheap sensors. In such environments, measurements are possibly impaired by limited technical precision, processing errors or natural fluctuations. Then, unusual readings might either indicate actual changes in the monitored system or be due to these measuring uncertainties. Finding associated sensors helps differentiate such cases.

For that task, we need a dependency measure that is widely applicable and not limited to modelling assumptions or specific relationships such as a linear functional relationship. While many relationships are well captured in some functional form, others cannot be described by a simple model or a single functional form. Also, limiting the scope of the algorithm to specific relationships automatically limits the possible findings.

An effective way to deal with complex relationships is the concept of mutual information, a measure that can be thought of as the predictability of one variable by another. The data we want to monitor is created in real time in an unending stream of data, e. g. from a system equipped with a large number of sensors to monitor its status. The (infinite) volume of this data and the high dimensionality (from the high number of individual sensors) require techniques that are fast and do not explicitly store past data.

The nature and volume of this type of data make traditional batch learning exceedingly difficult, and fit naturally to algorithms that work in one pass over the data,

i.e. in an online-fashion as discussed in Chapter 2. To achieve the transition from batch to online algorithms, window-based and incremental algorithms are popular, often favouring heuristics over exact results.

Instead of relying e. g. only on single stream statistics to detect anomalies or find patterns in the data, we are concerned with a setting where we find many sensors monitoring in close proximity or closely related phenomena, for example temperature sensors in close spacial proximity or voltage and rotor speed sensors in large turbines. It appears obvious that we should be able to utilize the – in some sense redundant, or rather shared – information between sensor pairs to validate measurements. The task at hand we want to solve first becomes then to reliably and efficiently compute and report dependencies between pairs or groups of data streams.

In the following chapter, we will show how to apply the analysis of information content, measured by their mutual information, to data streams. The two main difficulties are finding an estimator for the entropy of real valued streams (as opposed to the discrete case with event streams) and to find an efficient way to do the computations in an online fashion.

We will start with a simpler technique. First, we implement a straightforward window approach to deal with the streaming data. Second, we develop a suitable estimator for the entropy calculation. The algorithm was published as MID in [19]. This part will be covered in Section 5.2 and proves the validity of our approach.

In a second step we will refine the algorithm and find a truly incremental implementation for dependency monitoring. This will be covered in Section 5.3, the algorithm was published as DIMID in [20]. The evaluation experiments for both follow in Section 5.4

The three-fold advantages of both approaches are that (i) mutual information captures dependencies without limitation to a particular type or narrow model, (ii) is algorithmically suitable to be calculated in an incremental fashion and (iii) can be computed efficiently to deal with high data volume without the need for approximation short-cuts. This leads to a dependency measure that is significantly faster to calculate and more accurate at the same time.

5.1 Mutual Information as Dependency Measure in Data Streams

Shannon entropy and related concepts have been previously discussed in Chapter 3. They focus on the probability distributions underlying the transmitted data as a measure for their information content. For discrete data, the probability distribution is

a simple counting statistic. For continuous variables, knowing the PDF or getting it from a sample is more complex. The source of the random variables we use in this context are the values, i.e. measurements, say temperature measurements as real valued numbers, originating from some system equipped with multiple sensors.

Differential Entropy is the extension of the Shannon Entropy from discrete to continuous random variables. Entropy can also be conditioned on a second random variable Y and extended to pairs of variables X, Y. The continuous mutual information $I(X;Y)$ is a symmetric measure for the mutual dependence of X and Y. With the probability density function f, I is defined either via the PDF

$$I(X;Y) = \int_X \int_Y f(x,y) \log \frac{f(x,y)}{f(x)f(y)} \, dx \, dy, \tag{5.1}$$

or, equivalently, it can be expressed as the difference between the differential entropies $h(\cdot)$ and the joint entropy $h(X,Y)$:

$$I(X;Y) \quad = \quad h(X) - h(X,Y) + h(Y). \tag{5.2}$$

$I(X;Y)$ increases the better X allows us to predict Y and vice versa. It allows a ranking of the dependencies between random variables in a way that is comparable to other measures.

Both the continuous and the discrete mutual information $I(X;Y)$ are bounded by $max(H(X), H(Y))$. For the discrete case, the upper bound can be derived from a uniform distribution as $\log(max(|X|, |Y|))$ with $| \bullet |$ denoting the sample size. The lower bound is equal to 0. With these bounds we can define a normalized $\hat{I}(X;Y)$ which becomes 0 if X and Y are mutually independent and 1 if X can be predicted from Y and vice versa.

$$\hat{I}(X;Y) = 1 - \frac{I(X;Y)}{\log(max(|X|, |Y|))}. \tag{5.3}$$

This makes it easily comparable to the correlation coefficient and other measures with fixed ranges. Furthermore, $\hat{I}(X;Y)$ even forms a proper metric. This property confirms our understanding of $\hat{I}(X;Y)$ as an intuitive distance-like measure. But even without bounds, I serves as a score that allows a ranking of dependencies.

5.1.1 Dependency in Stream Windows

Next, we want to compute I for pairs of streams $s_i \in S$ at times t. $|S|$ is the *dimension* of the overall data stream S in the sense that every s_i represents a series of

measurements of a different type or a different sensor or both. Each stream represents a measurement series $s_i = (\dots, m_t^i, m_{t+1}^i, m_{t+2}^i, \dots)$ without beginning or end. Figure 5.1 shows a stream with three dimensions.

Since we are only interested in the most recent developments, the most straightforward design is a window-approach. In this approach, we take the most recent values within a fixed period and process those. There are two advantages. First, only those most recent measurements are kept in memory. Second, we can abstract from the stream and work with the current batch of data as we see fit.

Figure 5.1 shows a schematic of the window-wise computation over time for a stream with three sources s_1 to s_3. I is computed for all pairs of those stream variables.

If we restrict ourselves to the most recent values m of the stream, we add indices $s_i^{t,t+w-1}$ to denote measurements from stream s_i from time t to $t+w-1$, i.e. a window of length w. We will drop indices to simplify the notation if they are not relevant or necessary for clarity.

Our goal is then to efficiently calculate the stream dependencies D_t for all time points t in the observation period $t \in [0; \inf)$

$$D_t^w = \{I(s_i^{t,w}, s_j^{t,w}) | s_i, s_j \in S\}. \tag{5.4}$$

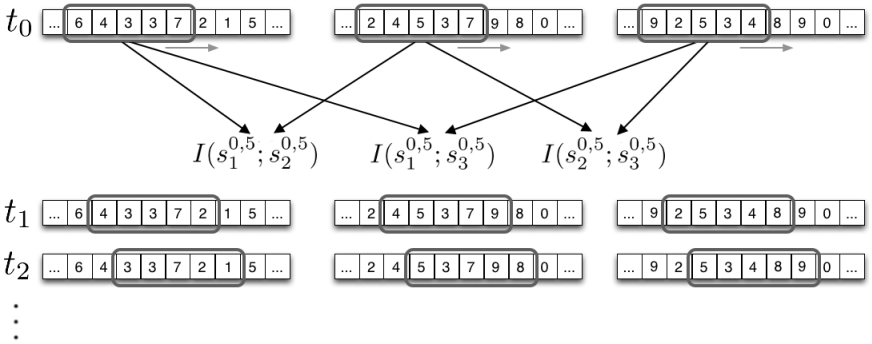


Figure 5.1: Sliding window and pairwise calculation of I for a data stream with window size $w = 5$ and three variables.

In general, we want to compute I for some or all pairs of variables $s_i^{t,w}, s_j^{t,w} \in S$ at all time points t. Effectively, I is calculated with the most recent w data points when new data arrives.

5.2 MID

This section describes our first draft of a dependency monitoring algorithm *MID*, a convenient, global measure to detect dependencies between data streams. We have settled for a window approach to deal with the streaming data so we only need a suitable way to calculate the mutual information in every window.

There are several ways the window can move forwards over time. We assume a fixed window size, i. e. the number of data points in a window stays the same as new data enters at one end at the same rate the oldest data is faded out. This is in contrast to an expanding or shrinking window.

The window can either slide forward if only one data point enters in each step. Alternatively, the window can jump forward, if more than one data point is absorbed at once. The sliding window approach is the most common one as it reduces the chance to miss (jump over) important steps. The sliding window shows also the smoothest behaviour. The parameter that controls the number of steps the window advances is called the step size s.

We adopt a sliding window with step size $s = 1$ for this discussion and for the experiments in later sections.

The window limits the amount of data in memory and also contains only the most recent data, so I is a measure for the current status of relationships in the data streams. In the basic window design, every window is processed separately from previous ones. This also implies that I is calculated for every window from scratch. In Section 5.3 we develop a more efficient design that re-uses the intermediate results and reduces the computation time significantly.

5.2.1 First Estimation of Mutual Information

The key problem to estimating the mutual information is finding a way to determine the joint probability density functions, regardless of whether we want to calculate the mutual information directly or its components separately (see the definitions above in equations 5.1 and 5.2). Two aspects are important: First, data streams often contain both nominal event data and real values. In consequence our model needs to deal with both continuous and discrete data types. Second, the underlying distribution of both single stream values and of the joint probabilities is – usually – unknown and must be estimated from the data.

Choice of Estimation Technique

Finding a mutual information estimate is generally considered a difficult problem. There is a range of entropy and mutual information estimates to find in the literature, coming from different theoretical directions. A good overview is given for example by [10] or [96].

There are two basic approaches to formulate a probability distribution estimate: Parametric methods or kernel-based methods, and binning.

In parametric methods, we would assume that the data is created by some stochastic process and try to pin down the best parameters for this process. Kernel-based methods work similarly but are potentially more flexible. For example in a Gaussian mixture model, we would assume the data is made up of a number of components, each of which can be modelled by a Gaussian distribution. We jointly optimize the distribution parameters to build a model for the data and proceed from the data. However, this not only requires assumptions about the process, the number of components, etc. It also leaves us with a large number of tunable parameters where sensible choices are difficult and maladjustment will lead to biased or erroneous results.[37]

Binning or histogram-based estimators are therefore the safer and more feasible choice for continuous data which have been well studied. [76, 67] They are also obviously the natural way to work with nominal data. They have been used previously in many different applications in data mining and machine learning, e.g. [37, 36, 90, 55].

There are some potential drawbacks: The quantization itself, the finite number of observations or the finite limits of histograms – depending on the specific application – might lead to biased results. However, [37] show that both equidistant and equiprobable binning lead to a consistent estimator of mutual information under mild conditions.

Choice of Histogram Parameters

The choice of the number of bins b is a critical problem for a reliable method. [54] point out that histogram estimators may be used to construct consistent entropy estimators for 1-dimensional samples and describe an empiric method or rule of thumb for histogram construction. The techniques used to derive their rule are related to the Akaike information criterion and the method must be interpreted as a heuristic. Similar to the Akaike criterion, it introduces penalties for different components of the model, in this case for the bias and variance components of the estimation error and balances their influence on the final result.

For our case, the rule can be written as

$$b = n^{(\alpha-1)/(3\alpha-1)}. \tag{5.5}$$

The rule mainly depends on the number of data points n in the sample. α is a regularization parameter with the constraint $\alpha > 3$. With e. g. $\alpha = 4$ this reduces to $b = n^{-0.\overline{27}}$. With values of n in the order of 1.000, this would result in a choice of b in the order of just 10 bins.

Algorithm 4 Window-wise Computation of Dependencies

```
1:  procedure MID(data streams S)
2:      for s^{t-w+1,t} ∈ S do
3:          ŝ ← Discretize(s^{t-w+1,t})
4:          P ← getPDF(ŝ)                          ▷ generate PDFs
5:          H ← entropy(P)
6:          JH ← jointEntropy(P)                   ▷ for all pairs
7:          I ← mutualI(H, JH)                       of streams
8:          output I
9:      end for
10: end procedure
```

Algorithm 5 Incremental Computation of Dependencies

```
1:  procedure IMID(data streams S)
2:      ŝ ← PiDiscretize(s^{1,w-1})
3:      P ← getPDF(ŝ)                              ▷ initialize from
4:      H ← entropy(P)                               window of first
5:      JH ← jointEntropy(P)                         w values
6:      for s^t ∈ S, t > w do
7:          ŝ^t ← PiDiscretize(s^t)
8:          P ← delete(ŝ^{t-w})                    ▷ update PDF with new
9:          P ← add(ŝ^t)                             frequency counts
10:         H ← update(P)                          ▷ update H and JH
11:         JH ← update(P)
12:         I ← mutualI(H, JH)
13:         output I
14:     end for
15: end procedure
```

Choice of Histogram Type

Of the two fundamental ways of discretization - equal-width or equal-frequency - equal-width binning is algorithmically slightly easier to execute, since it is only necessary to keep track of the current minimum and maximum. Equal frequency binning requires more effort, but has been shown to be the better estimator for mutual information.[12, 35] We confirmed this in a separate set of experiments and consequentially use equal frequency binning for our implementation.

For our algorithm, we discretize on a per-window-basis. A window-wise discretization gives us a local view on the data since it depends only on the properties of the data in the window but is also limited to the data currently available in the window. With the discretization in place, the probability distribution of a single stream and the joint distribution of a pair of streams is a simple counting exercise. We call $I(X;Y)$ with per-window discretization **MID** – mutual information dependency. For future reference, we add pseudocode for MID as Algorithm 4. The keyword **output** means a value is returned as output, for example to another program or a text console, but the control flow does not return, i. e. the loop continues to run.[1]

The runtime of MID is constant with respect to every update when new data arrives in the stream. The new incoming values possibly change the histogram boundaries in the window and therefore the underlying empirical probability distribution at each step. We need to discretize after every step, which gives a runtime of $\mathcal{O}(w \cdot n)$ after n steps.

Techniques exist for an incremental discretization, for example the PiD algorithm by [48]. It guarantees a certain maximum error on the ideal, optimal histogram borders to the current ones. This would allow an incremental discretization step in the algorithm and a much more efficient calculation of the PDFs of each stream and pair of streams. The PDFs change only by a $+1$ in one bin and a -1 in another. The pseudocode for such an incremental version of MID could look like algorithm 5.

However, our experiments have shown that such an incremental discretization does not have sufficient fidelity. Results depend heavily on the range of the stream values. If, say, stream values oscillate two stable states either around 20.0 or around 100.0 (for example machine temperatures in standby or production), a discretization that tries to accommodate both states with a range of 80 degrees is far to low grained for either state where differences of a few units have potentially great impact. Our experiments have confirmed this problem on real world data sets. This is not a fault of the particular discretization algorithm (we used the aforementioned PiD) but a general problem of discretization in data streams.

[1] This behaviour mimics the use of **yield** in the Python programming language.

Experiments to show the validity and performance of MID can be found Section 5.4, together with the experiments for the next iteration of our Dependency Monitoring algorithm. This next version will deal with the problems of entropy estimation and incremental calculations.

5.3 DIMID

In this section, we present DIMID, the Distance-based Incremental advancement of MID and a more efficient algorithm to compute a measure for dependency between pairs of real valued data streams. It calculates I, the mutual information of two streams in a sliding window over the most recent data and reports a value for I for every pair s_i, s_j and every window over time. I is, as previously explained, an entropic measure, so the key component in the empirical calculation of entropy or mutual information is a method to estimate the probability distribution from the sample of data in the current window. The main variants of estimates are kernel-, or histogram based. Kernel methods have many useful properties but are not only computationally expensive, they also require a choice of a kernel suitable for the application and in most cases a number of parameters that are hard to optimize, like the number of components used in a standard Gaussian kernel.

Discretizing the data and substituting the discrete case as estimate for the continuous case works well in many situations. Histogram-based estimators have been studied extensively and have been successfully used in data mining applications. [12, 35, 48, 67, 76] We mentioned potential drawbacks previously in Section 5.2.1. In particular, three problems are commonly recognized:

1. If the sample size is not very large and consequently the discretization rather coarse-grained, there is inevitably a loss of precision.

2. Although the determination of histogram borders can be guided by statistical rules to minimize errors, they also introduce bias and loss of precision.

3. When new data is added and old data faded out, the optimal boundaries change and we must either accept another source of imprecision or re-discretize the current data points.

More recently, alternative ways to estimate entropy from sample data have emerged. We will employ a distance-based alternative that computes differential entropy directly for continuous variables. This largely avoids the first two mentioned problems immediately and allows us to deal with the third intelligently. We discuss the steps of DIMID in the next sections and also give the pseudocode for the algorithm.

5.3.1 Incremental Dependency Computation

DIMID consists of an initialization phase and an iterated update procedure. Figure 5.2 visualizes the two phases: At $t = 0$, the window is initialized fully, for all later steps $t > w$, only the newest and the faded-out value are necessary for the re-computation of I.

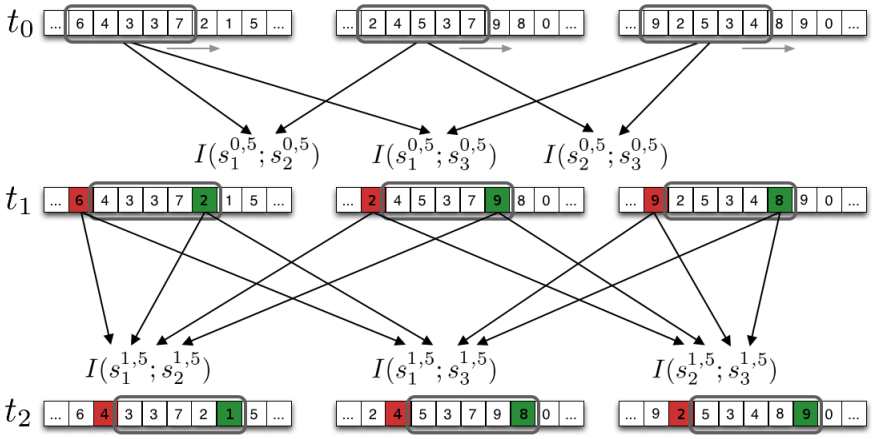


Figure 5.2: Sliding window and pairwise incremental calculation of I.

In the initialization phase, the initial estimates of I and data structures over the initial window are prepared to keep track of the nearest neighbour of the data points.

In the iteration phase, we calculate the changes in the sample caused by moving the window forward. The estimator we use relies on the distance to nearest neighbours of the data points. Adding and removing sample points has an effect on only a limited number of data points kept in the window. We can track the changes and calculate the resulting effect. In the second part of the iterative update, the effect of replacing a sample point is used to amend the previous estimate of I at minimal computational expense. The new estimate is then the output of the current iteration. This process is repeated infinitely, until the data streams are exhausted, or until stopped.

5.3.2 Beirlant Estimates of Mutual Information

We previously stated, how reliably determining the PDFs is the central problem of finding the entropy or mutual information. If we interpret the sample as a number of points in euclidean space, probability density is simply the probability of a sample point appearing in a given volume. High probability regions are determined by

regions with many, densely spaced sample points. Intuitively, in this case we can express the density of data points in a region just as well with the distances between data points. Beirlant *et al.* [10] give an entropy estimator (5.6) that relies on such a formulation of density:

$$h(X) = \frac{1}{n}\sum_{i=1}^{n}(\log(\varrho_i) + \log(n) + \log(2) + C_E) \tag{5.6}$$

$$= \frac{1}{n}\sum_{i=1}^{n}\log(\varrho_i) + \log(n) + \log(2) + C_E \tag{5.7}$$

$$= \frac{1}{n}\sum_{i=1}^{n}\log(\varrho_i) + c(n), \tag{5.8}$$

where C_E is the Euler constant and ϱ_i is the euclidean distance to the nearest neighbour (NN-distance) of the i-th sample. The NN-distance is calculated for each sample and averaged over all n sample points.

Equation 5.8 shows how we can simply split the sum in two parts. The second part, summed-up as $c(n)$, depends only on a predetermined sample size n and is constant with respect to n. The first part depends only on the distance ϱ_i of each sample point and determines the local density of the data points. It is this term only that changes in the underlying sample affect.

If we assume a sorted list of the sample points, finding the nearest neighbour of each becomes trivial. We simply compare the distances to the leftmost and rightmost neighbour which can be done in a single pass. This is of course only possible for one dimensional data points, since higher dimensional data cannot be sorted linearly. In the case of $d = 1$ however, every sample point has only two candidates for a nearest neighbour and in turn can be nearest neighbour to at most two other points.

We can split the mutual information $I(X;Y)$ into the components $h(X), h(Y)$ and $h(X,Y)$, using the equality (5.2). For the one-dimensional components $h(X)$ and $h(Y)$, we can proceed as we just described: Sort the samples, determine the nearest neighbour and calculate $h(X)$, $h(Y)$ via the Beirlant estimator (5.6).

Estimating $h(X,Y)$ in the same way directly is not possible, since the sample points (x_i, y_i) cannot be sorted linearly in a list. We could use a data structure to find the neighbours efficiently, but we want to avoid the necessary considerable overhead. Also, we want to find a scheme that allows quick updates to the NN-distances when we introduce new sample points and remove others. This will happen every time as the window moves forward. Instead, we can reduce the two-dimensional case, if we use the Johnson-Lindenstrauss-Lemma [61], that allows an approximately distance-preserving mapping from higher-dimensional into low-dimensional spaces.

For the mapping, we project the data points s_i, s_j via a mapping f into a random subspace $s_{i \circ j}$ with dimension $d = 1$. For such a low dimensional space, there is a risk of overestimating the influence of one dimension over the other and the procedure

works best if the two dimensions have approximately equal ranges. We accept this risk as a trade-off for the speed-up we gain through the reduction.

The Johnson-Lindenstrauss mapping f for our case can be simplified as

$$f(s_i, s_j) : (s_i^T, s_j^T) \cdot r^T, \tag{5.9}$$

where $r \in \mathbb{R}^2$ is a vector whose entries are independently drawn from a normal distribution $N(0, 1)$ and (s_i^T, s_j^T) is the $2 \times n$ dimensional matrix of the sample points. The projection can be carried out independently per pair of sample points. This means we can map each sample once, as it arrives, in $\mathcal{O}(1)$ per pair. This reduction lets us reuse the same estimation procedure (5.8) we used for $h(X)$ and $h(Y)$.

Now we have an efficient estimator to calculate the NN-distances from a sorted list. All we have to add is a way to handle the changes in the sample, the effects of these changes to our stored data, and the effect of the changing distances to the computation of $I(X;Y)$ incrementally. We want to be as efficient as possible and only calculate the changes, without redoing the unchanged parts in the estimation. This requires careful tracking of the sample points over time.

5.3.3 Updating Nearest Neighbours

The combination of dimensionality reduction and the Beirlant estimator we introduced, paves the way for an efficient calculation of mutual information. All we need is a way to track changes in the nearest neighbour distance of a changing sample. We can do so with a list of sorted sample points to calculate the neighbour distances and a second list sorted by age of the data points. The second list contains pointers to the first list. The process is shown in some detail as pseudocode in algorithm 8.

Its purpose is to keep the distances between neighbours up-to-date. In the initialization, we sort the initial window by value in a skip list and determine the distances of adjoining points. We also build a list of pointers to the sorted values, sorted by age (i.e. the pointer to the oldest sample point is always at the list head). The updates proceed incremental from here. If the window moves, the update is triggered in the main DIMID routine (line 11 of algorithm 6).

A new sample point arrives, and a new element is added to the list of pointers. We search the the list of pre-sorted, older sample points for the correct insertion position, and determine the nearest of the two neighbours. We also check, if the new data point is nearer to one of its neighbours than this points current nearest neighbour.

To fade out the oldest sample point that is leaving the window, we remove the element at the end of the list of pointers and follow its pointer to the list of sorted

values. we remove the element here and check, if it was a nearest neighbour to one of its two neighbours.

Introducing a new sample point (finding the correct position and adding to both lists) takes only time $\mathcal{O}(\log(w))$ using binary search in w sorted data points. Finding and deleting the oldest point takes likewise $\mathcal{O}(\log(w))$ in a skip list, even though we can find its position in constant time with the list of pointers.

If we remove one point from the sample and add another, we have to check only four points for consistency – two could gain a new nearest neighbour and two more could have lost their nearest neighbour. The changes in the nearest neighbours are then used to update the current entropy estimate.

For the pairs of sample points we proceed in the same manner. First though, we have to perform the dimensionality reduction for every new pair. This is done in the main algorithm 6 in lines 9–11.

5.3.4 Updating Entropy

With the updated NN-distances, the entropy estimates can be updated efficiently as well: If we move the data window one step forward, one value leaves and one value enters the window.

Most data points and most NN-distances remain unaltered and only a few terms in the sum for the entropy estimate (the sum over the log of NN-distances in equation(5.8) change. We can treat the changed terms separately since they are mutually independent. Define $\Delta(h(X^{t_i}), h(X^{t_{i+1}}))$ as the incremental update of the differential entropy.

$$h\left(X^{t_{i+1}}\right) = h\left(X^{t_i}\right) + \Delta\left(h\left(X^{t_i}\right), h\left(X^{t_{i+1}}\right)\right). \tag{5.10}$$

Δ is simply the net difference of the sum over the log-distances, i.e. the contribution of the changes caused by incoming ($x^{t_{i+1}}$) and outgoing ($x^{t_{i+1-w}}$) data points. It is a term we can compute given the updated distances around the faded-out and the new data point.

$$\begin{aligned}
\Delta\left(h\left(X^{t_i}\right), h\left(X^{t_{i+1}}\right)\right) = \quad & \log(\varrho_{x^{t_{i+1}}}) - \log(\varrho_{x^{t_{i+1-w}}}) && (5.11)\\
& - \log(\varrho_{x^a}) - \log(\varrho_{x^b}) - \log(\varrho_{x^c}) - \log(\varrho_{x^d}) && (5.12)\\
& + \log(\varrho'_{x^a}) + \log(\varrho'_{x^b}) + \log(\varrho'_{x^c}) + \log(\varrho'_{x^d}), && (5.13)
\end{aligned}$$

where $x^a, \ldots$ are the neighbours of the newest and oldest data point, $\varrho_{x^a}, \ldots$ are the NN-distances of these four potentially affected neighbours before the update, and $\varrho'_{x^a}, \ldots$ their NN-distances after the update. The total update time is therefore $\mathcal{O}(1)$ for this part.

Algorithm 6 Main Algorithm DIMID

Input: Stream S, window size w
Output: Dependencies $I(s_i^{t,t+w}, s_j^{t,t+w})$

procedure DIMID
  $sv, d, pv, pd \leftarrow$ init() ▷ *cf. Alg. 7*
  for all $i, j \in \langle S \rangle$ **do** ▷ *for all pairs of stream indices*
    $I_{i \circ j} \leftarrow \text{Entropy}(d_i) + \text{Entropy}(d_j) - \text{Entropy}(d_{i \circ j})$
    output $I_{i \circ j}$
  end for
  for $s^t \in S$ **do** ▷ *for every new set of values*
    for all $s_i^t, s_j^t, i < j \in s^t$ **do** ▷ *for all pairs*
      $s_{i \circ j}^t \leftarrow \text{reduce}(s_i^t, s_j^t)$ ▷ *using equ. 5.9*
      update($s_{i \circ j}^t$) ▷ *cf. Alg. 8*
      update(s_i^t)
    end for
    for all $s_i^t, s_j^t \in s^t$ **do**
      $I_{i \circ j} = I_{i \circ j} + \Delta(pd_i) + \Delta(pd_j) - \Delta(pd_{i \circ j})$
      output $I_{i \circ j}$
    end for
  end for
end procedure

Algorithm 7 Initial Phase of Algorithm DIMID

Input: Stream S, window size w
Output: Sorted windows sv, NN-distances d, age-pointer pv and pd

```
1:  procedure INIT
2:      sv, d, pv, pd;                                  ▷ prepare empty lists
3:      v ← s^{t0,w}                                    ▷ get first w values
4:      for all (s_i^{t0,w}, s_j^{t0,w}) do             ▷ for every pair of variables
5:          s_{i∘j}^{t0,w} ← reduce(s_i^{t0,w}, s_j^{t0,w})
6:          v.add(s_{i∘j}^{t0,w})                       ▷ add the joint projection to v
7:      end for
8:      for all v_i ∈ v do
9:          sv_i, pv_i ← sort(v_i)
10:         d_i, pd_i ← distances(sv_i)
11:     end for
12:     return sv, d, pv, pd
13: end procedure
```

Algorithm 8 Update Procedure of DIMID

```
1:  function UPDATE(s_i^t)
2:      delete_v ← pv_i.pop()
3:      delete_d ← pd_i.pop()
4:      insert_v, insert_d ← insert_sorted(sv_i, s_i^t)
5:      pv_i.add(insert_v)
6:      pd_i.add(insert_d)
7:
8:      delete sv_i[delete_v]
9:      delete d_i[delete_d]
10:
11:     dist_l ← sv_i[insert_v] - sv_i[insert_v-1]
12:     dist_r ← sv_i[insert_v+1] - sv_i[insert_v]
13:     dist_del ← sv_i[delete_v+1] - sv_i[delete_v]
14:
15:     insert_at(d_i, insert_d, dist_l)
16:     insert_at(d_i, insert_d + 1, dist_r)
17:     d_i[delete_d+1] ← dist_del
18: end function
```

5.4 Experimental Evaluation

We performed experiments with the two algorithms, MID and DIMID, described above to evaluate their performance regarding effectiveness and efficiency, i.e. correctness of the results and the runtime to produce them. First, we compare MID to several other algorithms for dependency monitoring to prove the validity of the entropy-based approach. Then, we compare MID and DIMID with other algorithms to compare their performance. We do so on different publicly available synthetic and real world data sets.

5.4.1 Data Sets

For experiments with real data streams we choose a large variety of data sets including movement tracking, financial time series and environment sensors with different lengths and a total of 50 streams. Table 5.1 contains an overview of the data sets. They range from 350 to 17 million measurements in length and have dimensions between two and twelve streams or parallel series of measurements. In addition, we use one synthetic data set with linear, non-linear and noisy relationships.

CHF is a series of ECG measurements of patients with congenital heart failure, available from the BIDMC data base.[52] It is widely used in experiments time series studies in the medical field (related to our work for example in [32]). We use 12 of these for our experiments. In total this data sets contains 128 over 17.793.041 time steps. [9]

OFFICE is a data set by the Berkley Research Lab, that collected data about temperature, humidity, light and voltage from sensors placed in a lab office. We use a subset of 12 sensors since there are large gaps in the collection. The subset still contains some gaps that have been filled in with a missing-value indicator. In total this data sets contains 128 measurements over 65.537 time steps. [18]

Sunspot is the famous data set of sunspot activity going back almost 200 years. Sunspots have a periodic peak in activity about every 12 years. We use two subsets of 70 years with daily measurements in our experiments. [88]

PersonalActivity (*PA*) is a data set of motion capture where several sensors have been placed on five persons moving around. The sensors record their three-dimensional position. This data set contains 75 data points each from 5.255 time steps. [62]

NASDAQ contains daily course information for 100 stock market indices from 2014 and 2015, with 600 indicators (including e.g. open and high course or trading volume) over 320 days in total. We choose 12 of those indicators. [94]

LNR was additionally created as a synthetic data set with six time series of 6400

data points each. Each series is either a linear or non-linear function of the elapsed time t, two more are uniform noise (uniform in $[0, 1]$) and Gaussian noise (with $\mu = 0$ and $\sigma = 1$). In combination, the dependency of two functions f_i, f_j on each other behave non-linear, if one of f_i or f_j itself is non-linear, as random, if one of the functions consists of noise, and linear otherwise. The advantage of the synthetic data is a clear knowledge of the dependency in the data which has to be inferred from other data.

Table 5.1: Sets of data streams used in the experiments with length n and number d of streams selected.

Data set description	**d**	**n**
CHF (longtime ECG measurements)	12	17,793,041
Office (office environment data, incl. temperature and brightness)	12	65,537
Sunspot (sunspot activity since 1818)	2	25,900
Personal activity (motion tracking data)	12	5,255
NASDAQ (financial time series 2014-2015)	12	350

Table 5.2: Functions of the time series of the *LNR* data set.

$$
\begin{aligned}
f_1(t) &= t \bmod 400, & f_4(t) &= \sqrt{1 - t^2},\\
f_2(t) &= \sin(t) + \sin(t/3 + 20), & f_5(t) &= 2 \cdot t + 20,\\
f_3(t) &= t + t^2, & f_6(t) &= 100 \cdot t + 20.
\end{aligned}
$$

5.4.2 Experiment Settings

We use similar settings for all algorithms as far as possible to guarantee a fair evaluation. All algorithms calculate a dependency measure in a window over the data stream, so we naturally use the same window size. Other parameters have no direct correspondence.

Window size w determines the scale of correlation we are interested in and ultimately has to be chosen by the user. For the purpose of this evaluation we set it $w = 80$ for the synthetic data and equivalent to 1 second for the turbine data set, 30 seconds for the other sensor data sets, and to 4 weeks for the stock market data set.

The number of bins b for the discretization needs to be small enough to avoid singletons in the histogram but large enough to map the data distribution – we considered criteria for a sensible choice in Section 5.2.1. As a compromise we chose $b = 25$ for the experiments. Two of the three algorithms we compare ourselves against, StatStream and PeakSim, have a similar calibration parameter, that balances runtime and validity. We set their truncating or sampling factor $c = 25$ equal to b.

We calculate dependency of every dimension with every other, e.g. voltage with temperature. So, for a data set $n \times d$ i.e. with n steps and d dimensions we calculate $(n-w) \cdot \binom{d}{2}$ dependency scores. Statistical significance is determined with a standard two-sided t-test.

All experiments have been performed on a single 1.80 GHz i-5 core and 8 GB of RAM.

5.4.3 Evaluation Criteria

With all real data sets, we evaluate the algorithms ability to distinguish dependencies from non-dependencies by classifying pairs of streams over time. For the data sets, we report the area under ROC curve as classification measure. AUC is independent from the number of true positives in the data set and can be understood as a measure determining how well the raw scores differentiate dependent from random interactions:

$$AUC = P(X_1 > X_2), \tag{5.14}$$

where X_1 and $X2$ are the scores for a positive and negative instance respectively.

Also, we report the F1-measure, i.e. the harmonic mean of precision and recall:

$$F1 = \quad 2 \cdot \tfrac{precision \cdot recall}{precision + recall}, \tag{5.15}$$

$$\begin{aligned} precision = &\quad \tfrac{TP}{TP+FP}, \\ recall = &\quad \tfrac{TP}{TP+FN}. \end{aligned}$$

As classification threshold, the score above which we report an instance as positive, we use the one that produces the highest F1-score for each algorithm., i.e. the threshold that best separates positive from negative instances for each algorithm.

We also use synthetic data to check, how well the algorithms discriminates non-linear as well as linear relationships. We report the mean scores for random, linear and non-linear relationships (see 5.4.1 for an explanation of these terms).

5.4.4 Experiments for MID and DIMID

We evaluate DIMID and MID against three other algorithms for stream correlation monitoring, PeakSim, StatStream and MISE, on one synthetic and five real world data sets. We show the raw scores for different interaction types to determine how well each algorithm separates the interaction types. To evaluate the algorithm in a supervised fashion, we first calculated a naive score with the Beirlant estimator for mutual information. We consider windows with a naive score above the median, i.e. the top half, as dependent and therefore true positives. AUC and F1 scores are used to determine the overall classification performance. Finally, we perform an analysis of the runtime for different window sizes.

The score results for the synthetic data is shown in Table 5.3 and the evaluation results for the real data sets are shown in Table 5.4. Tables 5.5 and 5.6 show an overview to compare methods with each other.

Comparison Algorithms

We will compare DIMID and MID to three other algorithms that have been developed for dependency monitoring: StatStream, PeakSim and MISE. They are algorithms with similar goals to ours but using different techniques. MID, StatStream and PeakSim have in common a transformation of the raw data before the main score-computation. This transformation introduces a calibrating factor c, that determines for PeakSim the number of Peaks of the Fourier transform, and for MID the number of discretization bins. MISE introduces a sampling rate for its reservoir. These factors determine in essence the compression rate of the data and influence therefore the speed and quality of the results.

StatStream[101] and PeakSimilarity[85] are algorithms to monitor stream correlation. Both employ variants of a discrete Fourier transformation (DFT) to detect similarities based on the data compression qualities of DFT. More specifically, they exploit that DFT compresses most of a time series' information content in few coefficients and develop a similarity measure on these coefficients.

The similarity measure for PeakSimilarity is defined as

$$peak\,similarity(X,Y) = \frac{1}{c} \cdot \sum_{i=1}^{c} \frac{1 - |\hat{X}_i - \hat{Y}_i|}{2 \cdot max(|\hat{X}_i|, |\hat{Y}_i|)}, \tag{5.16}$$

where X and Y are the time series we want to compare and $\hat{X}_i, \hat{Y}_i$ the c coefficients with the highest magnitude of the respective Fourier transformations.

The similarity measure of StatStream is similarly defined as

$$stat\,stream(X,Y) = \sqrt{\sum_{i=1}^{c}(\bar{X}_i - \bar{Y}_i)^2}, \tag{5.17}$$

on the DFT coefficients of the time series, but here $\bar{X}_i, \bar{Y}_i$ are the largest coefficients of the respective Fourier transformations of the *normalized* X and Y.

StatStream also uses hashing to reduce execution time, but the choice of hash functions is highly application specific. PeakSimilarity relies on a similarity measure specially defined to deal with uncertainties in the measurement, but requires in-depth prior knowledge of a cause-and-effect model to do so.

PeakSim and StatStream use the parameter c, which determines the number of DFT-peaks taken into account and influences runtime and memory in a similar way the number of bins b influences MID. Consequently we set c equal to b, which is in line to the choices of c in [101] and [85].

MISE[63] uses a sampling strategy to query mutual information on arbitrary time window queries applying the Kraskov estimator [67] for mutual information to a reservoir of past data. We fixed the sampling rate such that the last c data points are guaranteed to be in this reservoir. The Kraskov estimator relies on the distance to the k-nearest neighbour, so we have an additional parameter k. We set it to $k = 4$ as in [63].

Evaluation on Synthetic Data

On the synthetic data, we took the eight functions from the LNR set and tested all 28 pairwise interactions. We separate them into non-linear (N), linear (L) and random (R) interactions considering that interactions of a linear and a non-linear function is non-linear itself and that the interaction of noise with any function is noise itself. N and L interactions share dependency via t.

For the evaluation, we report normalized average scores for the algorithms on the different interactions to see how well the score separates dependencies from noise. Table 5.3 shows the mean score per algorithm. PeakSim and StatStream's scores for R and N overlap significantly, while MID and MISE rate N interactions significantly higher, but not with the same confidence as the scores for linear interactions indicate. DIMID's scores for N and L overlap within one standard deviation. DIMID discriminates non-linear and linear relationships almost equally well from noise.

Table 5.3: Mean raw scores ($\pm$ standard deviation) by dependency type on LNR

Interaction	StatStream	PeakSim	MISE	MID	DIMID
R	0.60 ± 0.10	0.43 ± 0.07	0.46 ± 0.11	0.05 ± 0.03	0.05 ± 0.04
N	0.64 ± 0.30	0.57 ± 0.27	0.58 ± 0.30	0.63 ± 0.27	0.62 ± 0.15
L	0.95 ± 0.15	0.98 ± 0.03	0.67 ± 0.15	1.00 ± 0.00	0.77 ± 0.04

Evaluation on Sensor Data Sets

For all data sets, we report the AUC and the F1 score in Table 5.4. To calculate those criteria, we first implemented a naive version of the Beirlant mutual information estimator and run it on the data sets. This gives us a score for every pair of streams and every window. We consider the top half of the scores in each data set as dependent, i. e. a positive instance. We then use the scores calculated by the compared algorithms as if they were classification scores.

Averaged over all data sets, DIMID achieves an AUC of 0.80, a significant improvement to the next best algorithm, MID reaching only 0.62. The other mutual information based algorithm, MISE, reaches an AUC of 0.54 and performs about on par with the other, DFT-based algorithms.

The same tendency is shown in the F1 score, where MISE reaches 0.70 while DIMID achieves an 17% improvement to 0.82. PeakSim and StatStream perform worse, especially notably in the NASDAQ and Office data set, despite the fact that there are many simple to detect linear relationships in those data sets.

For both measures, DIMID significantly ($p < 0.01$ in a two-sided t-test) outperforms the other algorithms in 6 out of 10 cases and ties in four more.

For a side-by-side comparison, we performed a t-test for each pair of algorithms and present the results in Table 5.5 for the AUC and in Table 5.6 for the F1-score. Here, we see more clearly how StatStream barely outperforms PeakSim and occasionally achieves better results than one of the entropy based methods. MID and MISE perform on the same level, with a slight edge to MID due to better AUC results.

In summary, to detect dependencies, DIMID works significantly better than other entropy based algorithms and much better than DFT-based methods judged by AUC and F1-score in five data sets.

Table 5.4: AUC and F1 score of all data sets. † marks significant improvements ($p < 0.01$ in a two-sided t-test) over the next best algorithm.

	AUC				
Data set	StatStream	PeakSim	MISE	MID	DIMID
CHF	0.63 ± 0.09	0.60 ± 0.07	0.61 ± 0.10	0.55 ± 0.04	0.61 ± 0.13
Office	0.51 ± 0.10	0.52 ± 0.11	0.52 ± 0.13	0.65 ± 0.13	$0.97 \pm 0.03^{\dagger}$
Sunspot	0.50 ± 0.00	0.50 ± 0.00	0.52 ± 0.00	0.68 ± 0.00	0.69 ± 0.00
PA	0.54 ± 0.03	0.51 ± 0.02	0.52 ± 0.09	0.50 ± 0.06	$0.95 \pm 0.02^{\dagger}$
NASDAQ	0.54 ± 0.02	0.58 ± 0.02	0.54 ± 0.04	0.71 ± 0.03	$0.80 \pm 0.05^{\dagger}$
	F1 score				
CHF	0.77 ± 0.08	0.76 ± 0.10	0.76 ± 0.07	0.74 ± 0.08	0.77 ± 0.11
Office	0.65 ± 0.06	0.68 ± 0.06	0.69 ± 0.06	0.69 ± 0.06	$0.95 \pm 0.04^{\dagger}$
Sunspot	0.67 ± 0.00	0.63 ± 0.00	0.67 ± 0.00	0.67 ± 0.00	0.67 ± 0.00
PA	0.67 ± 0.00	0.67 ± 0.03	0.68 ± 0.02	0.67 ± 0.04	$0.88 \pm 0.03^{\dagger}$
NASDAQ	0.67 ± 0.01	0.68 ± 0.02	0.71 ± 0.06	0.73 ± 0.06	$0.84 \pm 0.02^{\dagger}$

Table 5.5: Pairwise comparison of all algorithms: We count significant improvement in AUC (p-value < 0.1 in a two-sided t-test) of algorithm in row vs. algorithm in column in 5 data sets. DIMID scores better in a total of 16 of 20 comparisons.

AUC improvement	vs.				
	DIMID	MID	MISE	PSim	SStr
DIMID	-	4	4	4	4
MID	0	-	3	3	3
MISE	0	2	-	1	2
PeakSim	0	1	1	-	2
StatStream	1	2	2	2	-

Table 5.6: Pairwise comparison of all algorithms: We count significant improvement in F1 value (p-value < 0.1 in a two-sided t-test) of algorithm in row vs. algorithm in column in 5 data sets. DIMID scores 14 wins.

F1 improvement	vs.				
	DIMID	MID	MISE	PSim	SStr
DIMID	-	4	3	4	3
MID	0	-	1	2	2
MISE	0	1	-	3	3
PeakSim	0	0	0	-	2
StatStream	0	1	1	2	-

5.4.5 Run-time Analysis

Considering that the number of pairwise dependencies grows quadratic in the number of monitored dimensions, computation speed is an essential factor to deal with high dimensional data. Besides the theoretical complexity, computation speed is influenced by a number of variables in practice. We performed experiments on the LNR data set to explore these variables.

All the algorithms in our evaluation are window-based and therefore the total run-time is asymptotically $\mathcal{O}(n)$. n refers here to the size of the output to take the pairwise comparisons into account. The other main parameter, the window size w, is a constant and irrelevant for the theoretical complexity. For high-throughput streams however, where new measurements arrive in millisecond intervals, we need to accommodate

more measurements in a given time span and dependency on w matters.

StatStream, PeakSim, MISE and MID also depend on another 'truncating' parameter, which influences the accuracy of their results and run-time. In general, higher fidelity means more computation time and vice versa. We kept the value of 25 for c and b in StatStream, PeakSim, and MID identical to those in the performance evaluation. We also kept the value of k equal to four in MISE. Experiments with more favorable or less favorable values show minimal difference to the results presented here. The influence of the window size w dominates the run-time over all other factors.

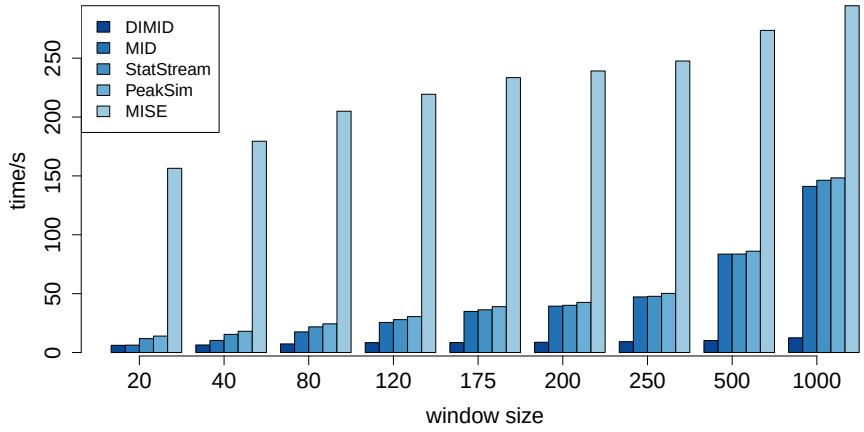


Figure 5.3: Run-time of Dependency Monitoring Algorithms
Run-time of StatStream, PeakSim, MISE, MID and DIMID on the LNR data streams

Figure 5.3 shows the relation between window size and run-time for all five algorithms. The size of the data set, i. e. the stream length was set to $n = 10.000$ and nine increasing window sizes were used. We show the minimum of five repeated runs. This minimizes random interference in the runtime from other processes. The fastest of the five times is the closest to the *ideal* runtime that can be achieved. The measured time also abstracts computation time as much as possible from the streaming process. This simply means that we run the algorithms without delay by the stream itself and new streaming data is queued as fast as it can be processed by each algorithm.

Approximation algorithms like PeakSim and MID scale well but still increase about linearly with window size $\mathcal{O}(n \cdot w)$ in practice. We can see this clearly from the repeated doubling in computation time over the three largest window sizes.

MISE also has a theoretical runtime linear in the size of the window, but it uses a specialized data structure to speed up finding the nearest neighbour. This data structure improves the most expensive step. As a result the algorithm scales better than

PeakSim, StatStream and MID, but has a large constant overhead, making it the slowest algorithm overall.

DIMID scales asymptotically as $\mathcal{O}(n \cdot \log(w))$ and easily processes our data set with the largest window size in the test in just over ten seconds. This is 12 to 24 times faster compared to the other algorithms. The speed advantage results from the incremental approach outlined in Section 5.3.

5.5 Related Work

We gave a more in-depth overview over entropy and streaming algorithms earlier. Here, we shortly summarize recent articles that affect or motivate our own work.

There is recent work that explores the use of entropy as a way to detect dependencies in static data sets. Reshef *et al.* [82] compare several methods to find novel associations in data sets with a large number of variables, including mutual information and develop their own, mutual information-based measure.

Benesty *et al.* [11] use entropy to detect delays within time series and achieve better results than state-of-the-art cross-correlation. Dionisio *et al.* [37] analyzed financial time series and concluded, that mutual information is a superior measure of dependence between random variables. They argue that mutual information is a practical measure of dependence between random variables directly comparable to the linear correlation coefficient, but with the additional advantage of capturing global dependencies, aiming at linear and non-linear relationships without knowledge of underlying theoretical probability distributions or mean-variance models.

There is a growing interest in frameworks and algorithms for stream monitoring. Seliniotaki *et al.* (PeakSim, [85]), Zhu *et al.* (StatStream, [101]) and Keller *et al.* (MISE, [63]) developed frameworks to monitor data streams for dependencies with a similar goal to ours, but rely on transformations or sampling of the data to detect correlation. MISE uses a sampling strategy to query mutual information on arbitrary time window queries applying the Kraskov estimator [67] to a reservoir of past data. PeakSim and StatStream employ a truncated Fourier transform and define a dependency measure on the peaks of the Fourier transform. We compared our own algorithm to those since they have been designed with a similar application in mind.

Clifford *et al.* [29] presented a sketch-based algorithm to estimate entropy over streaming data incrementally, but due to the nature of the sketch for all historic data at once, i.e. without forgetting old data.

5.6 Summary

Analysis of data streams based on their information content and the mutual predictability of their behaviour allows potential insights in the monitored system. Mutual information brings a different perspective to stream analysis that is independent from assumptions on the distribution of or relationship between the data streams.

We used mutual information, a concept from information theory, as a metric that can help to evaluate and monitor sensor readings or other streaming data. We develop two increasingly sophisticated ways to implement a monitor algorithm.

We have shown how such an analysis can be performed effectively with a simpler discretization of data streams. A more sophisticated estimation technique allows an efficient implementation where computations from a previous step can be re-used and the entropy calculations can be done incrementally over time.

We evaluated our algorithms on five real life sensor data sets with up to 17 million records and against three other algorithms to detect dependencies in data streams. They are more accurate for detecting dependencies in the data than those algorithms and take less computation time on all data sets.

With the second resulting algorithm, DIMID, we achieve an increase of 37% in AUC and 17% in F1 in classification accuracy over the competing algorithms. The improvement is due to the ability of an entropy-based distance measure to discriminate not only simple, linear relationships but also more complex interactions from background noise. The computation scales well with window sizes and is 12–24 times faster than similar algorithms.

Considering the growth in available sensor data and the increasingly complex systems they monitor, fast, efficient and universally applicable monitoring algorithms like DIMID remain a promising area of research.

There are a number of possible extensions. While we sidestepped the issues due to discretization or transformation of the data, the window size remains as an important parameter. It would be useful to either find a way to optimize the window size automatically or eliminate a static window size altogether. An example technique how to deal with such adaptive windowing is the ADWIN algorithm for data streams with concept changes. [14] However, window size depends heavily on the required resolution over time and might best be left as a choice for the user. The choice might however influence the sensitivity of a detection algorithm.

Extending the search for dependencies from pairwise to groups of three or more streams increases the computational complexity but brings the potential to broaden the analysis to an entropy-based ad-hoc clustering. This would require a different approach to estimate multivariate entropy than the projection method shown here. The

entropy estimation approach in the next chapter is computationally more expensive, but can be extended to multivariate analysis.

We also did not take the problem of time-delayed relationships into account. Delayed effects after an initial cause for example effectively hide relationships from an observer. We will discuss one way to capture such hidden relationships in the next chapter.

Chapter 6

Delayed Dependency

As we discussed in the previous chapter, the analysis of data streams has become an interesting research area, that provides algorithmic and technological challenges. Data streams appear in diverse environments. Examples can be found in the fields of medicine, industrial or environmental control, finance or social media. Applications of stream analysis include sensor surveillance, motion tracking or the analysis of network traffic. The wide differences in the purpose and nature of the systems where the streams originate make it that much harder to create techniques that can be used for and are actually useful in the analysis of different kinds of streams. We closed in on the complex and diverse nature of data streams with techniques to analyze the dependence in streams originating from the same system. To do so, we use an entropy-based measure that deals successfully with the inherent complexities.

These dependencies, however, can be further complicated if we think about the possibilities of time delays in the data. Imagine, for example, relationships in weather data where temperature changes have delayed effects on humidity or precipitation. The relationships might also only appear for a short time period or be stable for months and years. Another possibility for time delay effects appears if multiple sensors pick up changes caused by the same event, but from varying distances because they are placed in different locations. For these and other reasons, relationships might exhibit a lag that hides the relation from an observer.

The problem of lagged dependency is the analysis of two or more evolving sequences of data for dependence and for the time-delay or lag at which the dependence is the strongest. Since the streams evolve, i.e. dependencies change over time, this also becomes a continuing monitoring task. To solve this problem, an algorithm needs a general measure for dependency in time series, has to work efficiently in constant time and in constant space and provide accurate results over a wide range of time delays.

The material in this chapter has not been published so far. Our contributions discussed below are under review for publication in:

[21] J. Boidol and A. Hapfelmeier. Lagged dependencies in data streams. *In Review to: IEEE Transactions on Knowledge and Data Engineering*, 2017

In this chapter, we introduce the idea of cross-dependency, an analogue to cross-correlation from signal processing transferred to information theory. We discuss possible ways to compress time series while capturing their dynamics and we explore the effect of reconstructing the cross-dependency from limited samples. From these, we develop an algorithm to efficiently calculate the cross-dependency on multiple data streams. We evaluate the algorithm on five data sets, some of them synthetic. Since miscellaneous use cases allow different optimizations of the algorithm, we explore the effects of those optimizations on runtime and accuracy of the algorithm.

We propose an online algorithm called Loglag to calculate the mutual information for lagged data streams. Mutual information as dependency measure captures all possible types of relationships without limiting them to linear or monotonous types. A geometric sampling in conjunction with adaptive compression of old data provides very accurate results for smaller delays and a good approximation for large delays. This algorithm is the first algorithm to calculate lagged mutual information on data streams.

6.1 Time-delayed Streams

Loglag is designed to find the maximum cross-dependency and the corresponding lag on a pair or for all pairs of large numbers of data streams. In this section, we will define cross dependency and introduce the notation for lagged signals.

The notion of a lag between time series stems from signal processing, where it defines the offset between two signals. We can shift one time series relative to the other to align values from different time steps with each other. A common methods involving lagged signals is the cross-correlation. There, one signal is shifted along the other, and for every offset or lag the correlation of the signals is determined. The function with the lag and both signals as input and their correlation at the given lag is called cross-correlation, or auto correlation if a signal is compared to itself.[81]

This method is useful in many ways. For example, we can fit a shorter signal at the appropriate position on a longer signal. Say a short speech signal, a word in a sentence, slides along a longer signal, in this case a whole sentence. The cross correlation would peak around that lag which places the word at the correct position in the sentence.

In a similar manner, cross-correlation can be used to determine the delay between acoustic signals. If two microphones record the same signal at different distances, the lag corresponds to a delay, caused by the transit time of the signal. The cross-correlation aligns the signals for every possible delay and gives for each a measure of similarity of the respective match. The maximum of the cross-correlation function then determines the optimal lag between the signals.

Cross-correlation employs correlation as measure of similarity, a simple and often effective measure. As in the previous chapters, we want to move to a different measure that is more suitable for stream analysis. As a measure of dependency between two time series, we use as again their mutual information I.

As with cross-correlation, the lag l is the relative shift of two time series to each other. The optimal lag now is the position at which the behaviour of one time series is most predictable from the other and vice versa. Given a time series $X = (x_1, \ldots, x_t)$ with the newest element x_t at time t, and a second time series Y, we can calculate a measure of dependency for all lags as a function of the shifted series $D(X, Y, l) = g((x_l, \ldots, x_n); (y_1, \ldots, y_{n-l}))$. If $g(\bullet, \bullet)$ is the normalized covariance, this is identical to cross-correlation. If we take g as the mutual information I, we call it the *cross-dependency*.

We define the cross-dependency $D(X, Y, l)$ between X and Y as

$$D(X, Y, l) = I((x_l, \ldots, x_n); (y_1, \ldots, y_{n-l})). \tag{6.1}$$

The task of cross-dependency monitoring is then to calculate D for all possible lags l and report the optimum lag with the maximum dependency on a pair or group of data streams. Since there can be periodic patterns in the data such as daily or seasonal dependency, it is more adequate and practical to report the earliest local maximum above a specified threshold.

For data streams, there are additional difficulties in lag detection. One in particular in this kind of lag detection is the need to keep large amounts of historic data which can be shifted relative to each other. And as a collateral, the computational cost to calculate the dependency for every shift grows with the length of the data stream.

6.2 Efficient Lagdetection

Two main ideas are behind the design of Loglag. We need to choose again an appropriate estimator for mutual information and have to find ways to calculate the cross-dependency efficiently.

6.2.1 Kraskov Estimator

Estimating mutual information from sample data is a difficult problem, as we established in the previous chapter in Section 5.2.1. However, it functions as model-independent measure and is not limited to certain types of relationships in the data or data drawn from a known distribution. Mutual information has been defined as

$$I(X;Y) = \int_X \int_Y f(x,y) \log \frac{f(x,y)}{f(x)f(y)} \, dx \, dy, \tag{6.2}$$

or as sum of underlying component entropies

$$I(X;Y) \quad = \quad H(X) - H(X,Y) + H(Y), \tag{6.3}$$

where $f(x,y), f(x), f(y)$ are the joint and marginal probability density functions.

A number of estimators for I can be found in the literature, for example those by [10], [67], [66], but the kNN-estimator by Kraskov *et al.* [67] has not only been shown to be comparatively little biased and very stable on real world data. It is free from assumptions on the data distribution and therefore insensitive to violations of such a distribution. It is non-parametric and instead based on the density of data points in the neighbourhood around each sample data point. It also has two convenient properties: First, it does not resort to the calculation of the component entropies (see (6.3)). This avoids the errors possibly made in the individual estimates which might not cancel each other and lead to a biased estimation. Second, if we do not need the component entropies and skip the space and time required to calculate those, we can reduce the total space and time complexity. This will be helpful in the final design of the algorithm in which we use the estimator.

Details, especially regarding the derivation can of course be found in the original publication (Kraskov *et al.* [67]). Here, we briefly explain the main idea and the quantities in the final expression.

The Kraskov estimator uses the number of points within a margin of each data point as approximation for the distribution of the sample. The margins for each point are derived from the distance to the k-nearest neighbour. Earlier, we called the estimator parameter-free. This is true despite the choice of a value for k in the estimator, because this value does not have to be adjusted to the sample. Larger values simply provide more accurate estimates but require higher computational cost. It has been found that small values for k (i.e. $k \leq 4$) are already sufficient for the estimate.[64], [96], [63]

Formally, we consider $Z = ((x_1, y_1), \ldots, (x_n, y_n))$ the joint space of X and Y. With the maximum norm as distance measure

$$dist(z, \tilde{z}) = max(|x - \tilde{x}|, |y - \tilde{y}|),$$

we determine d_k as the distance to the k-nearest neighbour z' around a sample point $z = (x, y)$. More precisely,

$$\begin{aligned} d_k &= dist(z, z'), \text{with } z' \text{such that} \\ k + 1 &= |\{\hat{z} \in Z \mid dist(z, \hat{z}) \leq dist(z, z')\}|. \end{aligned}$$

The use of the maximum norm is different from other kNN estimators.[50] It creates a square centered around z_i (see Figure 6.1) and allows a succinct way to express the expectation value of the probability mass p_i within this square. In particular,

$$E[log(p_i)] = \psi(k) - \psi(n).$$

The digamma function ψ arises since we have to account for the $n - k$ ways of drawing the surrounding points.

Interestingly, the two samples X and Y can be from completely different spaces and use different distance norms. In most cases and in our examples however, the euclidean distance is the appropriate distance metric.

In the next step, we count the points $x' \in X \setminus \{x\}$ within a distance of less than d_k of x as n_x and the points $y' \in Y \setminus \{y\}$ respectively as n_y to establish a density estimate. These *marginal counts* can be plugged into the Kraskov estimator for mutual information:

$$I(X; Y) = \frac{1}{n} \sum_{(x,y) \in Z} \psi(k) - \psi(n_x + 1) - \psi(n_y + 1) + \psi(n), \tag{6.4}$$

where n is the size of the sample or the common length of X and Y. The terms $\psi(k) + \psi(N)$ serve as normalization for sample size and the value for k. In practice, this estimator boils down to some simple operations: Finding the k-nearest neighbour and determining how many data points fall within a region determined by this neighbour. With some care, we can optimize these operations for data streams. We will show these steps in Section 6.3.

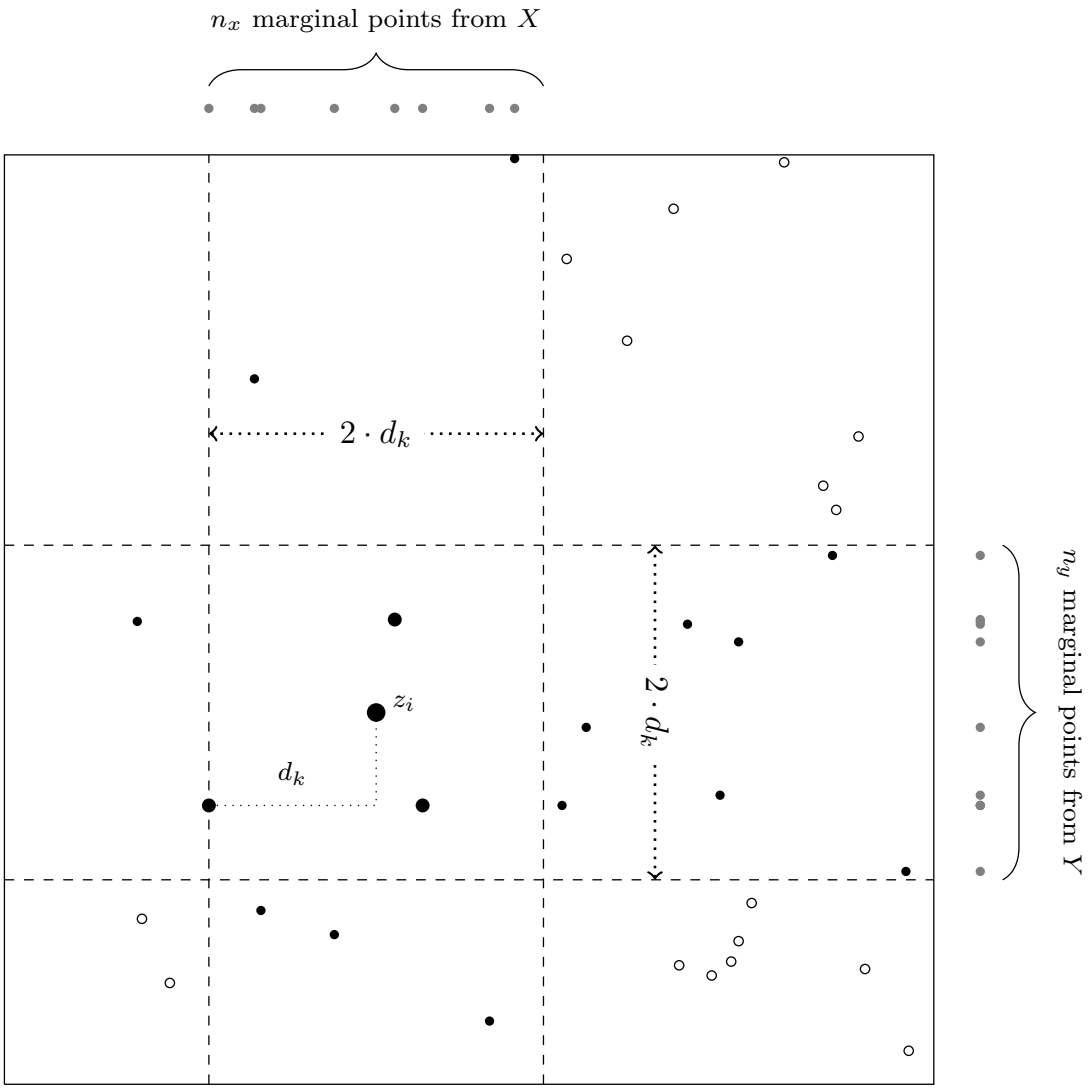


Figure 6.1: Data points in the joint two-dimensional space Z of X and Y. The k-nearest point determines the size $2 \cdot d_k$ of the margins around the selected point z_i. Points from X and Y within the marginal distance are called marginal points. In the example $k = 3$.

6.2.2 Geometric Probing

To find the full cross dependency of two streams X, Y up to a lag l_m, we would have to calculate the dependency at l_m positions. We can save much of the work here, if we recognize two things. First, it is unlikely for the dependency to change abruptly from one lag to the next. Second, we can accept a grade of approximation for larger lags. Together, this means we only have to *sample* the cross dependency function at some lags and reconstruct the complete function from those sample points.

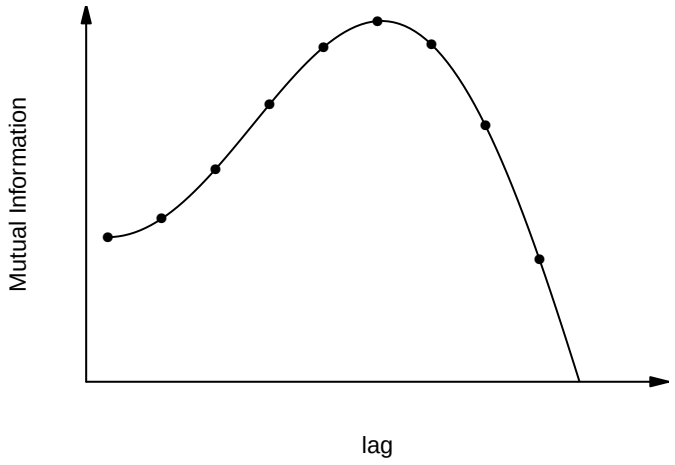
For Loglag, we reconstruct the cross-dependency function from a geometrically spaced probing. Compare figure 6.2 for an example of the idea. Instead of naively calculating the dependency for every possible lag, we take only a subset of all lags, every 2^i-th lag up to a maximum lag l_m. The distance between the actually calculated points increases geometrically with the lag. While we chose to double the spacing, other bases than 2 are of course possible. Other bases would allow to fine-tune the spacing to a greater degree, but they offer no fundamental difference. Later, we will follow two other ways to control the spacing instead.

With the geometric spacing, the error between the filled in and the correct dependency vales will be small at small lags where accuracy matters most. Even for a large lag the *relative* error remains small. Assuming that the interpolation error is proportional to the sampling rate, the relative error will even remain constant since the number of sample points decreases in proportion to the size of the lag. We can justify the sampling further with the following thought: If there are several peaks in the cross-dependency, we want to find the earliest significant one. The earliest peak is also the one with the highest local sampling frequency in this scheme.

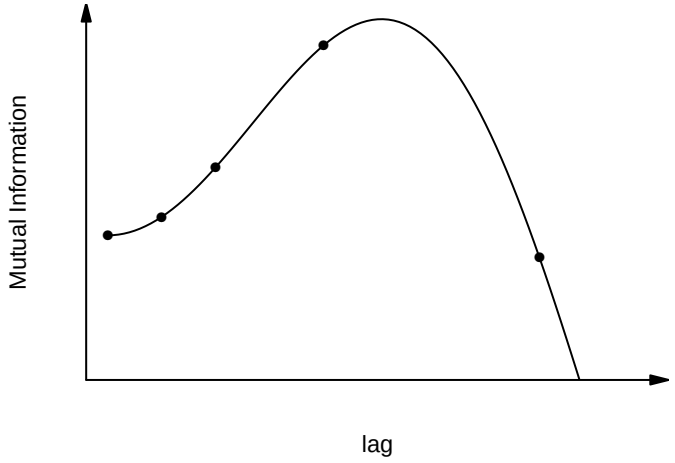
After the sampling, we can fill in the lag values between the probed points with any interpolation method. Our method of choice are cubic splines to get a smooth and efficient interpolation. The probing reduces computation time greatly to $\mathcal{O}(\log(l_m))$ compared to $\mathcal{O}(l_m)$ in the naive solution.

Error Estimation

In addition to the intuitive idea of the probing, we can make some assertions to the accuracy of this scheme: The Nyquist–Shannon sampling theorem states that we can perfectly reconstruct a signal from a uniform sampling, if the sampling is spaced at most $2 \cdot f_s$ Hertz apart and the signal contains no frequency higher then f_s. Even for non-uniform sampling, we can reconstruct the full cross-dependency from the sampling, if the *average* sampling rate is at least $2 \cdot f_s$. [86] For our geometric sampling, this means we can perfectly determine the cross-dependency D up to a lag l, if $l < \frac{2}{f_D}$ and the highest frequency in the signal D is at most f_D. This proves the



(a) Calculate dependency at every possible lag value.



(b) Calculate dependency at geometrically spaced sample points.

Figure 6.2: Naive calculation and interpolation of the cross-dependency function.

initial assumption that sampling will not hurt the overall accuracy to much, provided the cross-dependency function is reasonably smooth. Intuitively, we capture D accurately, if the dependency does not change too fast from one instance to the other.

6.2.3 Smoothing

The geometric probing drastically reduces the computation time necessary to calculate the cross-dependency, but does not change the amount of data necessary to do so. To get the dependency at all sampled points, we still would need to save historic data up to the largest lag value we want to compute. We implement two techniques to reduce the storage requirements.

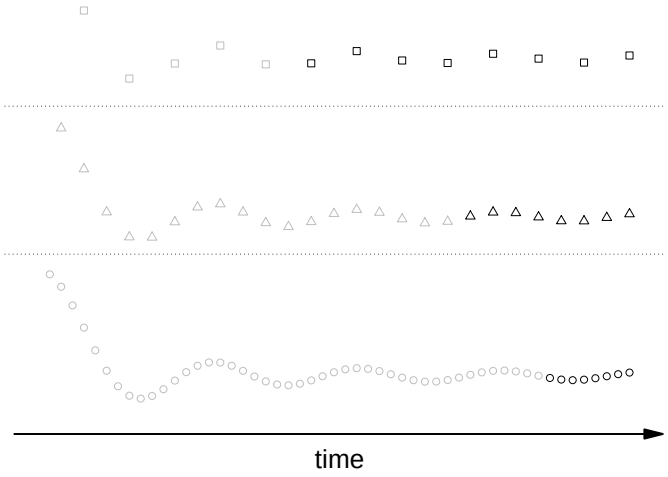
First, we restrict the calculation of the mutual information to a window of the most recent data. There are good reasons to adopt the window-approach beside the cost efficiency it provides. As in the previous chapter, we want to gives more weight to recent data to detect a relationship even if it emerges for example only part of the time or for a shorter time. Also, since we shift two data streams to each other to find their dependency at different lags, the overlapping part of the two streams gets shorter for longer lags. With a fixed window size, the sample size we compute the lag on stays constant and simplifies the implementation.

As a second technique, we compress older data points. Older data is necessary only to compute larger lags. For those, we are more interested in the large scale dynamics of the data stream, since we increasingly approximate and smooth the cross-dependency for larger lags anyway, as we described in the previous section. Figure 6.3 shows the smoothing we want to achieve for a single stream.

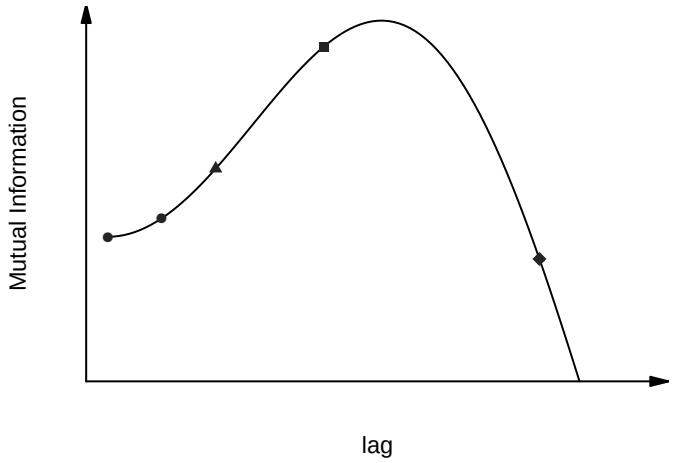
To do so, we create layers of increasingly compressed and smoothed version of the sequence. The first layer stores the most recent $w + 1$ data points, where w is the selected window size. The overhang is necessary to allow a shift with respect to a second data stream and maintain an overlap of w. Every further layer stores the same number of data points but represents a window with the endpoint increasingly further backwards in time. That way the number of points stays constant in each layer but the covered time horizon increases per layer. Compare Table 6.1 as an example with $w = 8$.

We refer to increasingly compressed layers as L_h where L_0 stores the last $w+1$ of the original, uncompressed stream. Every higher layer averages $c = 2^h$ consecutive data points, for example two values from layer L_{h-1}, $x^h_{t_{h-1}}, x^h_{t_{h-1}-1}$ are averaged to $\hat{x}^{h+1}_{t_h}$. t_h denotes an index for each layer h where $t_h = \lfloor \frac{t}{2^h} \rfloor$. t_h effectively counts the number of (averaged) data points that have ever been part of the layer h. An increase by two of t_0 (two data points have arrived in the original data stream and in L_0) translates to an increase of one in t_1 (the two new data points are averaged to one point in L_1.

We can use these layers to enact the geometrically spaced sampling of the cross-dependency. Every layer spans double the part of the data stream than the layer below.



(a) Smoothed data. The solid data points are kept in memory such that each layer has the same number of points but stretches increasingly further back.



(b) Reconstruction of the cross-dependency. Sample points are calculated on the basis of different compressed layers.

Figure 6.3: A smoothed, compressed representation of the data streams is kept in memory. From each layer, we calculate a sample point used in the reconstruction of the cross-dependency.

Table 6.1: Layers of compressed data. The most recent data point is x_{256}, each layer doubles the coverage of the data stream backwards in time. Layer $h = 0$ stores data back to x_{248}, layer $h = 1$ stores data back to x_{239}, etc. The last layer stores data back to $t = 256 - w \cdot 2^h$.

	$w = 8$				
$h = n$	x_{t_n}	x_{t_n-1}	x_{t_n-2}	$\cdots$	x_{t_n-w}
$\vdots$			$\vdots$		
$h = 2$	x_{t_2}	x_{t_2-1}	x_{t_2-2}	$\cdots$	x_{t_2-w}
$h = 1$	x_{t_1}	x_{t_1-1}	x_{t_1-2}	$\cdots$	x_{t_1-w}
$h = 0$	x_{256}	x_{255}	x_{254}	$\cdots$	x_{248}

In the same way, a shift by l in Layer L_h is equivalent to a shift $2l$ in the layerL_{h-1}.

To calculate the geometrically spaced lags of 2^h, we use the smoothed stream L_h and calculate the lag $l_h = 1$ corresponding to $l = 2^h$. The maximum lag we can calculate is determined by the number of layers or vice versa. More precisely, for a maximum lag of l_m we need a maximum of $h_m = \lceil \log_2(l_m) \rceil$ layers. The calculation of larger lags in higher layers also has the desirable effect that larger lags are calculated from larger slices of the data and less likely to be the effect of statistical fluctuations. While the effect of a small lag might only be detectable in the most recent data, the effect of a dependency with large delay should also be detectable over a larger time frame.

Error estimation

The smoothing obviously introduces an error in the calculation of dependency. This error depends on the degree to which the smoothing approximates the original data. We can show that for streams with low frequencies the error of the approximation is usually small. We can make this assertion towards its accuracy if we think of the compressed data as discrete Haar wavelet transform (DWT). [83] The Haar DWT repeatedly averages consecutive values and, similarly to a discrete Fourier transformation, the highest coefficients of the transform contain most of the information of the original data. Given the original data x_t and the smoothed $\hat{x}_{t_h}$ in the normalized stream X, we get n Haar wavelet coefficients w_i of X. The quadratic error between smoothed and original data depends on the highest frequencies:

$$\sum_{t=1}^{n}(x_t - \hat{x}_{t_h})^2 = \sum_{i>n/2^h} w_i^2. \tag{6.5}$$

In most real data sets, most of those coefficients are small and only a few contribute significantly to the dynamic of the data, so we can expect the error introduced by the smoothing to be small.[51]

With $\Delta x_t = x_t - \hat{x}_{t_h}$, equation (6.5) tells us that $\Delta x_t \ll x_t$. The probability of a compressed data point crossing the margin defined by the k-nearest neighbour in the Kraskov estimator is Δx_t. The expected error of the marginal count, assuming all points are located on one side of the margin, is then bound by the expected value of a binomial distribution $w \cdot \Delta x$. The error is likely smaller, since data points might cross in either direction and cancel each other. The expected number of false counts is even zero, if we assume a symmetric distribution around the margin.

The effect on the MI estimate I is hard to quantify since the distribution of marginal counts may vary greatly. The mean error $\bar{\epsilon}$ for a given distribution of n is

$$\bar{\epsilon} = 2 \cdot \sum_{i=1}^{\infty} p(n_i)[\psi(n_i) - \psi(n_i - w \cdot \Delta x)],$$

with PDF $p(n_i)$ of the marginal points. The steps of the digamma function $\psi(n)$ are largest for small values of n, so increasing the sample size would help to reduce the error. We also average the w MI estimates from each data point which helps to minimize the error further.

Space and Time complexity

Clearly, every layer can be maintained with the data of the one directly below it as new data arrives, pools in its layer, is compressed and cascades to the next layer. Maintaining layer h requires a small, constant number of operations c every 2^h steps. For all layers, the update operations amount to $\sum_{h=0}^{h_m} \frac{c}{2^h} \leq 2c$ in total. We can update them in amortized time $\mathcal{O}(1)$ per data point over the lifetime of the stream. The space requirement is simply $\mathcal{O}\left(h_m \cdot w\right)$ for the $w + 1$ values in the h_m layers.

Calculating the mutual information for each of the $log(l_m)$ Layers takes time $\mathcal{O}(w^2))$. We have to find the nearest neighbours and count the marginal points in time $\mathcal{O}(w)$ and have to repeat this for each of the w data points in a layer.

From these ideas, sampling and smoothing, we can already build an efficient lag detection algorithm as we will demonstrate in the next section. There are several possible advancements to these fundamental ideas which we will discuss in Section 6.3.1.

6.3 Loglag

As we described previously, Loglag relies on a hierarchy of smoothed layers L_0 to L_{h_m}. We discuss the case of a pair of streams X, Y for simplicity, with obvious generalizations to pairs between three and more streams. The pair of layers of the same compression level L_h from each stream is used to calculate one sample point of the cross-dependency $D(X, Y, 2^h)$. Putting these steps together in an algorithm is fairly straightforward, we state the pseudocode in Algorithm 9 below. The main routine iterates over a maintenance part and an output part.

The maintenance function updates the progressive layers as new data arrives from the stream. The pseudocode for this part, Algorithm 10, describes more in detail, how every new value is added to the base layer and new values are pushed upwards to higher levels when new values have accumulated. Old values that are not needed anymore are discarded at the same rate. We need the $w + 1$ most recent values per layer to calculate the lag at each level for a sample of w values: $(x_{t-w+1}, \ldots, x_t)$ and the lagged version $(x_{t-w}, \ldots, x_{t-1})$.

The output function can be called every time new data arrives, at regular intervals, or at user defined time points. In this case, we defer the evaluation of $D(X; Y)$ until output is actually required. This is advantageous if we expect the streams to move very fast or output is only necessary at larger intervals.

An adaptation for other settings is discussed in Section 6.3.1. We will see that the lazy evaluation provides a formidable speed-up in the right conditions.

For the actual lag detection, we calculate the mutual information at every layer h, with lag l_h as estimate for the lag at $l = 2^h$. This can be seen in Algorithm 11. To interpolate the full function, we fit a cubic spline for each gap we need to fill. To find the maximum, we evaluate the function piecewise with one's favorite method to find local maxima, e.g. Brent's method [24]. The calculation of the mutual information uses the Kraskov-estimator we introduced in Section 6.2.1.

As mentioned in Section 6.1, we can either report the full cross-dependency or, if we suspect periodic dependencies and are given a threshold for the score, we determine the lag l_p with maximum score s in an interval between each neighbouring pair of sample points. If it is not an endpoint of the interval, we report l_p if its score s is above the given threshold. This finds the earliest local maximum of a given lag function.

Algorithm 9 Main Algorithm Loglag

Input: Streams S, window size w, max lag l_m

procedure LogLag
  $L, E, JE \leftarrow$ init()
  for $s_t \in S$ **do** ▷ *for every new set of values*
    for $s_i \in s_t$ **do** ▷ *for every stream*
      $//Maintain\ layers$
      SmoothLayers(s_i)
    end for
    $//If\ output\ is\ desired\ at\ t$
    for $i, j \in \langle S \rangle$ **do**
      $//Calculate\ Best\ Lag$
      LagDetecting(i,j)
    end for
  end for
end procedure

Algorithm 10 Layer Maintenance

Input: s_i

procedure SmoothLayers
  $L_0^i(t) = s_i$
  $del \quad L_0^i(t - w - 1)$
  for $h = 1$ to $\lceil \log(m) \rceil$ **do**
    $t_h = \lceil t/2^h \rceil$
    if $t \mod 2^h = 0$ **then**
      $L_h^i(t_h) = \frac{L_{h-1}^i(2 \cdot t_h) + L_{h-1}^i(2 \cdot t_h - 1)}{2}$
      $del \quad L_h^i(t_h - w - 1)$
    end if
  end for
end procedure

Algorithm 11 Lag Interpolation

Input: i, j

procedure LAGDETECTING
    $lag = \{0, 1, 2^1, \ldots, 2^{\lceil \log(m) \rceil}\}$
    GetMI(i,j)
    $scores = \{MI_l | l \in lag\}$
    fit splines over $(l, scores)$
    maximize splines
    return best lag
end procedure

Algorithm 12 Mutual Information Calculation

Input: i, j

procedure GETMI
    $X, Y = L_0^i(t, t-w+1), L_0^j(t, t-w+1)$
    $n_x, n_y =$ marginal_counts(X, Y)
    $I_0^{i,j}(0) = \psi(k) + \psi(w) - \psi(n_x + 1) - \psi(n_y + 1)$
    for $h = 0$ to $\lceil \log(m) \rceil$ **do**
        $t_h = \lceil t/2^h \rceil$
        $X = L_h^i(t_h, t_h - w + 1)$
        $Y = L_h^j(t_h - 1, t_h - w)$
        $n_x, n_y =$ marginal_counts(X, Y)
        $I_h^{i,j}(1) = \psi(k) + \psi(w) - \psi(n_x + 1) - \psi(n_y + 1)$
    end for
end procedure

6.3.1 Extensions

There are several ways to improve on the basic design of Loglag to optimize memory usage, accuracy or speed.

LoglagA

An obvious extension to the algorithm is to use every layer for more than one sample point for the cross-dependency. The basic space requirement per layer is $w + 1$, but to get another sample point, we need to extend a layer only by a few data points. Let us say we want to perform g sample point calculations or lags per layer. On the base layer, we have to calculate $2g$ shifts since there is no layer below that one. On the higher layers, the first g shifts are already covered by lower layers and we want to calculate the shifts from $g + 1$ to $2g$. Therefore every layer is extended by $2g$ time steps to provide data for g non-overlapping shifts.

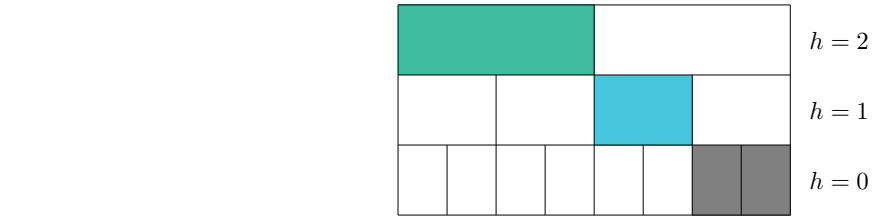
Figure 6.4 shows the layers of an example with $g = 2$. On the base layer, we calculate $2 \cdot g$ dependency scores up to $l = 4$. On every subsequent higher layer, we calculate lags from $(g + 1)2^h$ to $2 \cdot g2^h$, in the example the lags at $3 \cdot 2^h$ to $4 \cdot 2^h$. We need one more data point to skip over the lags covered by the levels below and one more for the second lag we wanted to calculate.

In total, at the expense of $2g \log(m)$ additional memory, we achieve a finer sampling and consequently more accurate results.

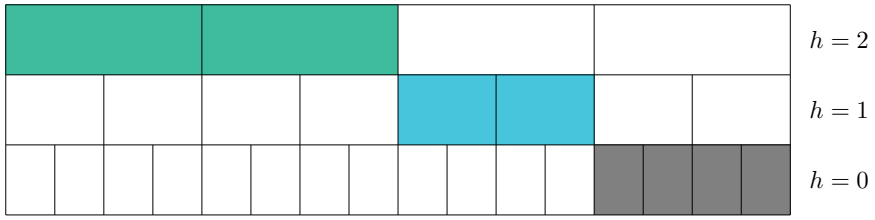
Another improvement is possible, if we realize that not all the layers are actually necessary to calculate all lags. We need to store $(h+1)(w+2g)$ data points to calculate a lag of $2^h \cdot 2g$ on the h-th layer. To calculate the same lag on the first layer, the unsmoothed sequence, we could extend this first layer to $w + 2^h \cdot 2g$. In consequence we can get rid of all the h intermediate layers. Only at L_{h+1} will the smoothing start to save memory. We can solve numerically for the number of layers h to skip, with W as the lower branch of the Lambert W function:

$$h = \left\lfloor -\frac{W\left(-\frac{\log(2)}{w/g+2} 2^{\frac{-2}{w/g+2}}\right)}{\log(2)} - \frac{2}{w/g+2} \right\rfloor .$$

For example, a window of $w = 100$ and $g = 4$ means we have to realize every eighth layer and can do the work of eight increasingly smoothed layers on the first most fine grained layer with no additional memory cost. Window size and other parameters stay the same. We also implement the update process that propagates new values to the layers and the geometrical sampling just as before. That way, the complexity of the space and time requirements stays identical to the base version.



(a) One lag per layer.



(b) Multiple lags per layer

Figure 6.4: Successive layers of Loglag h 0 to 2. In the normal case (a), every layer contributes one lag. The base layer also contributes the lag $l = 0$. In the extended case (b), the layers contribute several (here: two) lags each. Lags and corresponding shifts are larger so as not to overlap with the lags contributed from the layers below.

We call the algorithm with these memory improvements LoglagA and use it as the base version in our experiments. With the parameter that controls the number of lags per level g set to 1, it has the exact behaviour described as in Algorithm 9.

LoglagD

Since Loglag is meant as a monitoring algorithm that produces results continuously, we can refine the search for the optimal lag with previous results. If we have detected a lag l, $(g+1)2^h \leq l \leq 2g \cdot 2^h$, we call h the *responsible* layer. In the next iteration, we can assume that the optimal lag will still be in the area we determined before, somewhere between $2^h(g+1)$ and $2^{h+1}g$. We then can spend some additional memory on the next lower level $h-1$ and grow it by another $2g$ data points to $w+4g$. This allows us to make level $h-1$ responsible for lags up to $4g \cdot 2^{h-1} = 2g \cdot 2^h$, the limit of the layer h where we detected the lag. If the layer $h-1$ has grown to its new size, we substitute the results from the previously responsible layer with those from the finer resolved layer below. We can do so at double the sampling rate.

Once we detect the optimal lag outside the interval $[(g+1)2^h, 2g \cdot 2^h]$, i.e. the layer h would be no longer responsible for the lag, we let the increased layer $h-1$ shrink again to its normal size. The process to achieve this doubling of the resolution starts again at the new responsible layer.

We call this extension of the algorithm LoglagD which has slightly increased memory and runtime cost compared to LoglagA.

LoglagI

LoglagA (and LoglagD) spends most effort in the main routine on maintaining the smooth layers and defers the calculations related to the evaluation of the optimal lag as long as possible. The maintenance or insert complexity as shown in Section 6.2.3 is $\mathcal{O}(1)$ per data point. The evaluation of the mutual information $I(X;Y)$ on the other hand has complexity $\mathcal{O}(w^2)$. Depending on the insert-to-eval ratio this is not desirable.

The evaluation of $I(X;Y)$ requires for every point first the distance to the k-nearest neighbour and second the number of marginal points within this distance, provided this marginal point still falls within the current window. We can use a small data structure to track the nearest neighbours over a time w (the window length): Every point z_i in the window keeps a set N^+ of the k-nearest neighbours added *after* the point itself and an array of sets N^- for the possible nearest neighbour sets *before* it. The different handling of newer data points and older data points has the following reason: If an evaluation is triggered at time t, the window stretches from the newest data point z_t to some point z_{t-w} behind z_i but the time difference between z_i and z_{t-w} is not known at the time the data structure for z_i is created.

The set of future neighbours is simply updated every time, we move the window forwards since those future neighbours are guaranteed to fall within the window w. The array of sets for previous neighbours is constructed whenever a new point is added, say at time t'. We iterate over the data points in the window before it and spawn a new set for the time $t'-s$ if the data point $z_{t'-s}$ is closer to $z_{t'}$ than the k-nearest point from $z_{t'-1}$ to $z_{t'-s+1}$.

The insert of a new point takes amortized time $\mathcal{O}(w)$. The time to construct the array of possible sets N^- is $\mathcal{O}(w)$ and the time to keep the set N^+ up-to-date is $\mathcal{O}(w)$ over the w time steps a point is kept in the current window.

To find the actual k-nearest neighbours of z during the evaluation, whatever the position of z within the window, we form the union of N^+ with the correct set N^- of the step that contains no points older than the current window. The list of the N^- sets is ordered by time by construction, so even in the worst case of w such sets we

can find the correct one in time $\mathcal{O}(\log(w))$. The union of the sets and determination of the k-nearest neighbours takes time $\mathcal{O}(k)$, i.e. a very small constant. This brings the total time to determine the nearest neighbours at evaluation to $\mathcal{O}(w \cdot \log(w))$ for the whole window.

This pre-computation comes with additional memory cost, since we not only augment every point with this data structure. We also need a hierarchy of compression layers for every pair of streams instead of one per stream as in LoglagA. We can no longer store the values from each dimension of a data stream in a separate hierarchy each, since we need the two-dimensional points from the joint space Z in every layer. We also need one hierarchy per lag that is computed per layer, the parameter g, since the layers cannot be shifted as required.

Depending on the insert-to-eval ratio, however, it allows a considerable speed-up by more than two magnitudes as shown in our experiments. We call this version with incremental pre-computation of the nearest neighbour LoglagI.

Table 6.2 sums up the time and space complexities of the three variants we discussed here. An experimental evaluation of both time and accuracy follows in the next section.

Table 6.2: Asymptotic Time and Memory Complexity of the Loglag variants. w is the window size, d the dimension of the data stream, l_m the maximum detectable lag, g is the number of lags calculated per layer.

	Time		**Memory**
	Insertion	kNN Evaluation	
LoglagA	$\mathcal{O}(d)$	$\mathcal{O}(d^2 \cdot g \cdot w^2 \cdot \log(l_m))$	$\mathcal{O}(d \cdot (w+g) \cdot \log(l_m))$
LoglagD	$\mathcal{O}(d)$	$\mathcal{O}(d^2 \cdot g \cdot w^2 \cdot \log(l_m))$	$\mathcal{O}(d \cdot (w+g) \cdot \log(l_m))$
LoglagI	$\mathcal{O}(d \cdot w)$	$\mathcal{O}(d^2 \cdot g \cdot w \cdot \log(w) \log(l_m))$	$\mathcal{O}(d^2 \cdot g \cdot w^2 \cdot \log(l_m))$

6.4 Experimental Evaluation

To evaluate the performance of Loglag, we run the algorithm with different settings on five data sets. LoglagI by design gives identical results to LoglagA, but LoglagD should show improved accuracy as it increases the sampling rate at the region around

a suspected lag. LoglagA, LoglagD and LoglagI each have different actual time complexities despite the identical asymptotic runtime of variants A and D (cf. Table 6.2).

Therefore we compare the accuracy of LoglagA and LoglagD to a naive calculation of the mutual information. Likewise, we compare the speed-up over a naive implementation of LoglagA, LoglagD and LoglagI at different insert-to-eval ratios.

6.4.1 Data Sets

We evaluate the ability of Loglag to accurately determine lags on a variety of different both synthetic and real world data sets. In each we expect lagged dependencies due to the nature or construction of the data set which makes them adequate for the present evaluation.

Temperature and **Light** are data sets collected by the Berkley Research Lab. They contain measurements temperature and light intensity from 54 sensors distributed in a large office space. We only use a subset of 12 sensors each and about three weeks uninterrupted measurements.[18]

The **Sunspot** data set contains the measured sunspot activity over almost 200 years. Sunspot activity has a known periodicity of ca. 10 years between maxima. We use two subsequences of about 70 years each.[88]

The **Spike** data set is a synthetic data set with periodic bursts of activity and intermittent noise, a pattern that occurs regularly in real world situations, e.g. daily traffic spikes or scheduled maintenance with associated abnormal activity. We use two such streams with the same period but shifted spikes.

Sine is a second synthetic data set consisting of two streams, each a mixture of sine functions. Such a mixture of functions with different periods mimics the overlay of e.g. hourly, daily, yearly, etc. periodic activity.

Table 6.3 contains an overview of the data sets with the length and number of streams in each data set.

6.4.2 Experiment Settings

For all experiments, we assume a constant speed of the streams, i.e. no delays through the streams themselves. We report the relative deviation between the optimal lag of the full sequence detected by a naive computation and our algorithm variants. For groups of streams, we report the average of all pairs. We chose a small value of $k = 4$ (cf. Section 6.1) and a value of $g = 4$ for all experiments. For LoglagD, we used the results of the LoglagA run as hints for the optimal lag position and the responsible layer to extend (cf. Section 6.3.1 for an explanation of *responsible layers* in LoglagD).

Table 6.3: Sets of data streams used in the experiments with length n and number d of streams selected.

Data set description	d	n
Temperature (office environment data) [18]	12	65,537
Light (office environment data) [18]	12	65,537
Sunspot (sunspot activity since 1818) [88]	2	25,900
Sine (mixture of sine waves)	2	6,500
Spike (pulse train with fixed period)	2	6,500

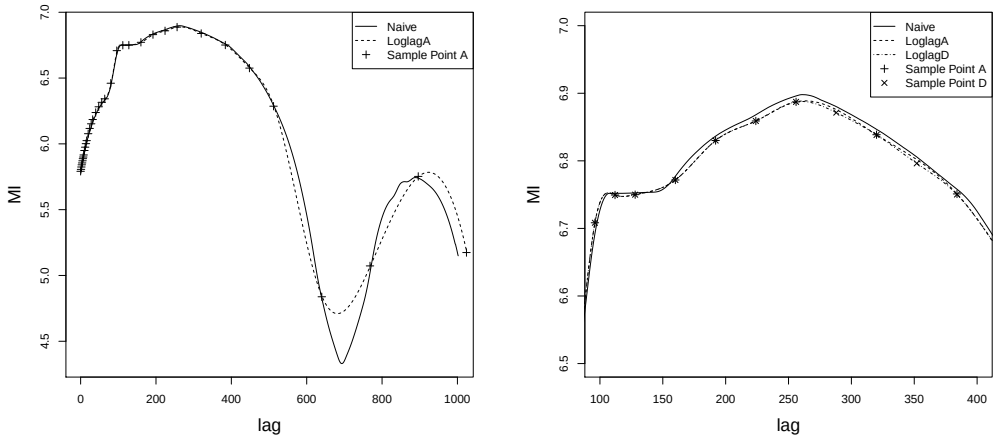
All experiments have been performed on consumer grade hardware with an Intel Xeon 1.80 GHz CPU. There was no hard limit set on the available memory, but physical memory never exceeded 100MB.

6.4.3 Evaluation Results

Table 6.4 shows the lags detected as optimal by naive calculation and our approximation. Overall, the estimates keep within 5.5% of the correct lag, with as little as 0.76% error by LoglagA and 0.38% by LoglagD. The increased sampling rate of LoglagD does improve the detection accuracy slightly, but not significantly (by 0.22% with $p = 0.89$), in two cases it increases even. This happens as the approximation of Loglag finds general trends and peaks but is not fine-grained enough to determine the mode of a peak exactly. The deviations leave room for small errors in both directions. If the lag is fairly small, as in **sine**, the differences between LoglagA and LoglagD are small (cf. Fig. 6.5b). For larger lags, which are calculated with a coarser approximation, LoglagD can help to provide not only more accurate results (cf. Fig. 6.7b), but also more precise absolute values for the MI score, as can be seen in Fig. 6.6b.

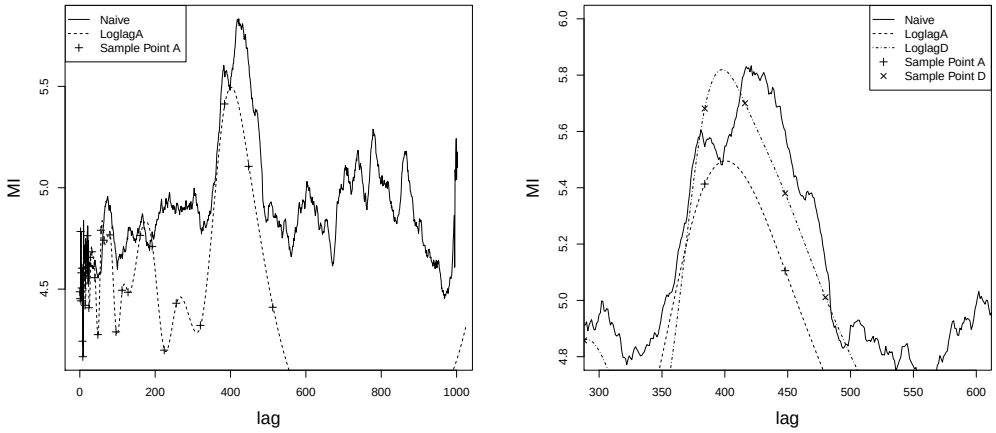
Table 6.4: Detected Lags and Errors of all Data Sets.

	Lag position			Error	
Data set	naive	LoglagA(I)	LoglagD	error A(I)	error D
spike	422	402	399	4.74%	5.45%
sine	262	264	261	0.76%	0.38%
sunspot	757	715	770	5.54%	1.72%
light (avg)	-	-	-	2.23%	6.70%
temperature (avg)	-	-	-	4.87%	2.78%
average				3.63%	3.41%



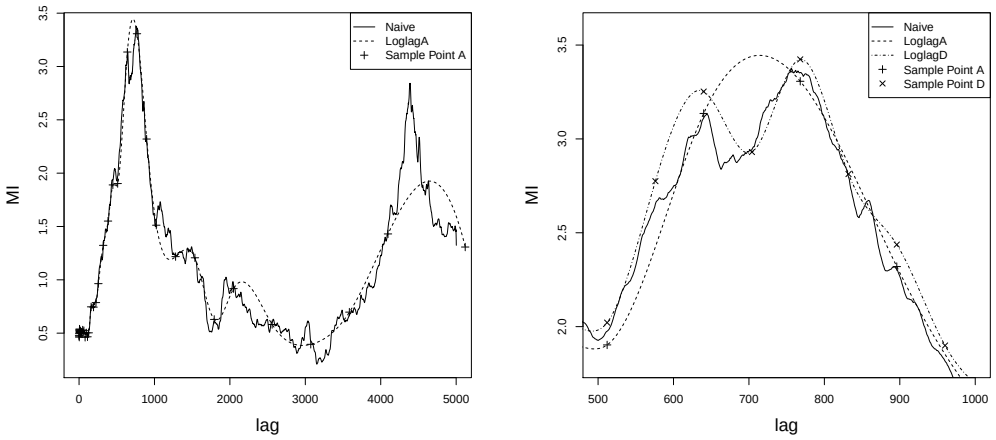
(a) Lag function (b) Lag function, close-up of best lag.

Figure 6.5: Entropic lag function from the **sine** data set.



(a) Lag function (b) Lag function, close-up of best lag

Figure 6.6: Entropic lag function from the **spike** data set.



(a) Lag function (b) Lag function, close-up of best lag

Figure 6.7: Entropic lag function from the **sunspots** data set.

6.4.4 Run-time Analysis

The main difference between LoglagA and LoglagD to LoglagI is the lazy evaluation of the dependency in the former two compared to the pre-computation in the latter. We therefore compare the runtimes for each flavour at different frequencies of evaluation, specifically at ratios r of 1 evaluation after every 1, 10 and 100 new data point arrives. We also compare different values for the maximum lag we want to be able to detect ranging from 100 to 10000. To evaluate the speed, we use one of our synthetic data sets, spike, with $n = 20000$ and run every algorithm for every setting. For every setting, we report the minimum of five runs, to achieve a result that is as free from interference as possible.

The most unfavorable settings for LoglagA and LoglagD are the ones with frequent evaluation, i.e. small r. Conversely, the most unfavorable settings for LoglagI have high r values. For both, we expect a higher gain – compared to the naive solution – with a larger maximum lags.

The evaluation results show, that even in the least favorable setting, significant speed-ups are realized. Figure 6.8 gives a complete overview.

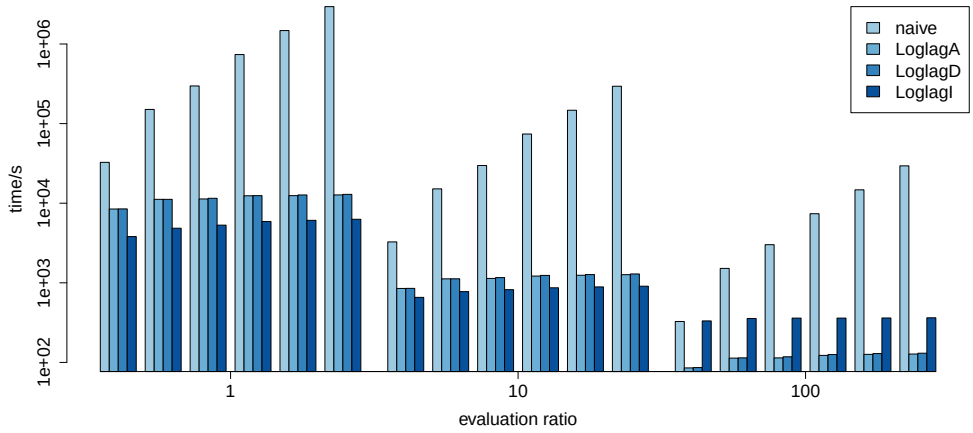


Figure 6.8: Naive, LoglagA, LoglagD, LoglagI on synthetic data set **spike** with $n = 20000$. Runtime in seconds is given for eval-ratios of 1, 10, 100 and max. lags of 100, 500, 1000, 2500, 5000, 10000. The time axis is logarithmic.

We achieve a speed-up for LoglagA of at least $3.8\times$, and $232.8\times$ at best, over the naive implementation. On average, including all settings, we improve by a factor of $75.6\times$. For LoglagD, the results are very similar, as expected: In the worst case, we are $3.8\times$ faster, in the best and average case the speed-ups are $228.1\times$ and $74.1\times$.

LoglagI by design works best for frequent evaluations. For those settings, it is up to $469.0\times$ faster, with an average of $95.8\times$. For infrequent evaluations ($r = 100$), lazy calculation not surprisingly performs better, but even there the LoglagI implementation is at worst on par with the naive implementation and on average still $26\times$ faster.

6.5 Related Work

Entropy-based measures are not limited to specific kinds of relationships and have attracted research effort for some time. Fraser and Swinney [43] used MI to detect concept changes in static data already in 1986. More recently Reshef *et al.* [82] and da Costa *et al.* [34] evaluate several algorithms to find associations or detect concept drift in data sets and report good results for MI approaches. There is even an stury by Benesty *et al.* [11] that tests a minimum entropy approach for time delay estimation in acoustic signals with superior results compared to standard techniques.

There is also a growing interest and corresponding work on the analysis of data stream dependency. A recent example is Keller *et al.* [63] who developed a framework to estimate mutual information on data streams to monitor dependencies, based on the Kraskov estimator [67] for MI but for arbitrary window sizes and window offsets. However, they do not consider the effect of delays on dependency. Earlier work by Sakurai *et al.* [83] on delayed correlation developed an approximation to the cross-correlation function, based on Pearson's correlation coefficient. This allows lag detection on streaming data, provided the relationships are simple enough to be captured by their linear model.

6.6 Summary

Finding dependencies in data streams promises insights into the hidden connections and relationships in a monitored system. This task becomes vastly more complex if time delays are taken into account. We developed an efficient and extremely fast algorithm, Loglag, to calculate dependencies with an optimal delay via the mutual information between data streams. The use of a nearest neighbour-based mutual information estimator covers a wide, distribution-independent spectrum of dependencies. An approximation via geometric probing allows efficient calculations over a wide time horizon. Loglag has been further optimized for high eval-to-insert ratios.

In experiments with five different synthetic and real world data sets, we achieve high accuracy and a much higher throughput compared to a standard mutual infor-

mation calculation. The average error in lag detection is 3.63% ranging between 0.76% and 5.54%. The speed-up averages $74 - 96\times$ including unfavorable settings.

We employ a fixed window size and uniform compression to limit memory consumption while still allowing the detection of large delays. In principle, sampling strategies like they have been utilized in [63] could be adapted to allow arbitrary window sizes in the evaluation of dependency and allow more flexibility in the monitoring. To reconcile this with the demands on low memory consumption and necessary speed for a monitoring algorithm remains an objective for further work.

These types of online analysis increasingly gain interest as both the amount of streaming data produced by smart and complex systems and the demands to extract their value grows. Algorithms like Loglag help to deal with the challenges presented in this environment.

Chapter 7

Conclusion

7.1 Summary

In this work, we dealt with two problems of stream mining, stream classification and stream dependency. Most practical applications that learn from, monitor or interpret data streams pose challenging tasks since streaming data is in many ways different from static batches of data. The most significant differences are the dynamic nature of data streams. It requires the ability of algorithms to forget older data and relearn when new concepts appear. They must also deal with the transience of the data, which is never complete and in practice impossible to store exhaustively. We deal with both supervised and unsupervised problems, which are both highly relevant scenarios in sensor surveillance and in the monitoring on large scales.

In the first part of this thesis, we dealt with supervised learning. We presented Probabilistic Hoeffding Trees, a novel approach towards stream classification. The probabilistic model treats examples to learn from not as absolute set of values, but as samples of a probability distribution. We showed how probabilistic learning greatly improves the flexibility of online decision trees and their ability to adapt to changes in data streams. The general technique is not only applicable to a variety of classification models, but fast to compute without significantly greater memory cost than base line models. We tested the Probabilistic Hoeffding Tree on several classifiers with tree models. On five sensor data sets, we achieved significantly improved accuracy with comparable model size and runtime across all the data sets and classifiers we examined.

We improved the classification accuracy by up to 16% with 3.2% on average in dynamic streaming data sets over other Hoeffding Tree models. Our approach reacts swiftly to changing data streams which makes it especially suited to environments where the concept behind the streams changes frequently, as is the case in many industrial or ecological applications.

As a second problem, we turned towards unsupervised monitoring of data streams. Knowledge of the relationships between streams allows hidden insights into a monitored system. Changes in these relationships over time might uncover extraordinary or anomalous events. To detect relationships in high dimensional data, we exploited their mutual information, which serves as a measure of mutual predictability without prejudice for particular forms of relationships. We presented MID and DIMID, algorithms that deal with the technical and theoretical complexities of calculating mutual information on data streams. MID performed a simpler discretization of data streams with good results. Beyond simply discretizing data, we also tested a more sophisticated distance-based estimation technique. This allowed an efficient implementation of DIMID where computations from a previous step can be re-used and the entropy calculations can be done incrementally over time. The incremental procedure reduced the theoretical complexity from $\mathcal{O}(w)$ to $\mathcal{O}(\log(w))$ where w is a window over the data stream.

We evaluated these two algorithms on five real life sensor data sets with up to 17 million records and against three other algorithms to detect dependencies in data streams. They outperformed those algorithms both in accuracy and computation time. DIMID achieved an increase of 37% in AUC and 17% in F1 in accuracy over the next best algorithms. The computation scales well with window sizes and is 12–24 times faster than the competing algorithms.

In addition to the time-aligned analysis, we introduced delayed relationships as a further complication in the dependency analysis. In reality, the phenomena monitored by e.g. some type of sensor might depend on another, but measurable effects can be delayed due to technical reasons, i.e. different stream processing speeds, or because the effects actually appear delayed over time. We presented Loglag, the first algorithm that monitors dependency with respect to an optimal delay. It utilizes two approximation techniques to achieve competitive resource requirements. We demonstrated its scalability and accuracy on real world data, and also gave theoretical guarantees to its accuracy. The first key idea was a geometric probing for lags to reduce the computation time. This simple but effective measure lowered the complexity from $\mathcal{O}(l)$ to $\mathcal{O}(\log(l))$ where l is the maximally detectable delay. The detection accuracy de-

creases mostly for larger lags where the same absolute error is more acceptable and can be refined in later steps anyway. Since the computation of dependency with large lags is also costly in terms of memory storage, we coupled this with a progressive compression which reduced the space complexity likewise from $\mathcal{O}(l)$ to $\mathcal{O}(\log(l))$ and larger lags are calculated with lower grained approximations. Again, the error is in proportion to the size of the lag and larger lags can be calculated with sufficient precision to detect the stream dynamics and refine the detection over time. In experiments with five different synthetic and real world data sets, we achieved high accuracy and a much higher throughput compared to a straightforward calculation. The average error in lag detection is 3.63% ranging between 0.76% and 5.54%. The resulting speed-up is excellent with $74 - 96\times$ shorter computation time, even with respect to unfavourable settings.

7.2 Outlook

Considering the growth in available sensor data and the increasingly complex systems they monitor, fast, efficient and universally applicable algorithms like those developed over the last chapters remain a promising area of research. We will end this thesis with our thoughts on areas that might merit further work in the future.

Using an entropy-based dependency measure and proper estimators for this overcomes the issues due to discretization or transformation of the data. The choice of window size is still left to the user depending on the desired application. Ideally, we could find an optimal window size that balances performance, the dynamic of the stream and frequency automatically. Better yet we could adapt it continuously to the current stream dynamic. Combining the presented techniques with methods for concept change has the potential to give less user dependent optimal results.

While we dealt with pairwise dependency, there is no theoretical obstacle to extend the monitoring from stream pairs to groups of streams. Multivariate mutual information would allow a broader base for the dependency analysis and could be extended to the construction of dependency groups or clusters. The lagged version of the problem appears similar to the construction of multiple sequence alignments, an interesting sub-problem in bioinformatics with a host of methods to solve this optimization problem.

Besides these algorithmic extensions, there is a recent trend in the stream mining community to incorporate geo-spatial information into an analysis. This could also be used to construct and compare local groups, optimize placement of sensors to minimize either failure or redundancy, or add spatial constraints to dependency detection.

Considering the resource efficiency of algorithms is often guided by the desire to perform computations not in large data centers, but on small cheap chips integrated in for example other machinery. Protocols for distributed computation in WSNs, wireless sensor networks, are another current research topic which can be combined with the type of analysis we performed here. Moving computation closer to the source allows for example lower latency and greater cost efficiency in large scale systems.

These types of online analysis increasingly gain interest as both the amount of streaming data produced by smart and complex systems and the need for smart analysis grows. Algorithms like those discussed here hopefully help to deal with the challenges presented in this environment.

Bibliography

[1] C. C. Aggarwal. *Data streams: models and algorithms*, volume 31. Springer Science & Business Media, 2007.

[2] C. C. Aggarwal and S. Y. Philip. Outlier detection with uncertain data. In *SDM*, pages 483–493. SIAM, 2008.

[3] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

[4] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[5] Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org/`, 2017. note = Accessed: 2017-01-30.

[6] Apache Software Foundation. Apache Kafka. `http://kafka.apache.org/`, 2017. note = Accessed: 2017-01-30.

[7] Apache Software Foundation. Apache Spark. `http://spark.apache.org/`, 2017. note = Accessed: 2017-01-30.

[8] Apache Software Foundation. Apache Storm. `http://storm.apache.org/`, 2017. note = Accessed: 2017-01-30.

[9] D. S. Baim, W. S. Colucci, E. S. Monrad, H. S. Smith, R. F. Wright, A. Lanoue, D. F. Gauthier, B. J. Ransil, W. Grossman, and E. Braunwald. Survival of patients with severe congestive heart failure treated with oral milrinone. *Journal of the American College of Cardiology*, 7(3):661–670, 1986.

[10] J. Beirlant, E. J. Dudewicz, L. Györfi, and E. C. Van der Meulen. Nonparametric entropy estimation: An overview. *International Journal of Mathematical and Statistical Sciences*, 6(1):17–39, 1997.

[11] J. Benesty, Y. Huang, and J. Chen. Time delay estimation via minimum entropy. *Signal Processing Letters, IEEE*, 14(3):157–160, 2007.

[12] H.-P. Bernhard, G. Darbellay, et al. Performance analysis of the mutual information function for nonlinear and linear signal processing. In *Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on*, volume 3, pages 1297–1300. IEEE, 1999.

[13] A. Bifet, E. Frank, G. Holmes, B. Pfahringer, M. Sugiyama, and Q. Yang. Accurate ensembles for data streams: Combining restricted Hoeffding trees using stacking. In *ACML*, pages 225–240, 2010.

[14] A. Bifet and R. Gavalda. Learning from time-changing data with adaptive windowing. In *Proceedings of the 2007 SIAM International Conference on Data Mining*, pages 443–448. SIAM, 2007.

[15] A. Bifet and R. Gavaldà. Adaptive learning from evolving data streams. In *Advances in Intelligent Data Analysis VIII*, pages 249–260. Springer, 2009.

[16] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. Moa: Massive online analysis. *The Journal of Machine Learning Research*, 11:1601–1604, 2010.

[17] J. A. Blackard and D. J. Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in agriculture*, 24(3):131–151, 1999.

[18] P. Bodik, W. Hong, C. Guestrin, S. Madden, M. Paskin, and R. Thibaux. Intel lab data. `http://db.csail.mit.edu/labdata/labdata.html`, 2004. note = Accessed: 2015-08-10.

[19] J. Boidol and A. Hapfelmeier. Detecting data stream dependencies on high dimensional data. In *The 1st International Conference on Internet of Things and Big Data, IoTBD 2016*, pages 375–382. INSTICC, 2016.

[20] J. Boidol and A. Hapfelmeier. Fast mutual information computation for dependency-monitoring on data streams. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2017.

[21] J. Boidol and A. Hapfelmeier. Lagged dependencies in data streams. *In Review to: IEEE Transactions on Knowledge and Data Engineering*, 2017.

[22] J. Boidol, A. Hapfelmeier, and V. Tresp. Probabilistic Hoeffding trees. In *Industrial Conference on Data Mining, Best Paper Award*, pages 94–108. Springer, 2015.

[23] L. Bottou. Large-scale machine learning with stochastic gradient descent. *Proceedings of COMPSTAT'2010*, pages 177–186, 2010.

[24] R. P. Brent. *Algorithms for minimization without derivatives*. Courier Corporation, 2013.

[25] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan. FlumeJava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010.

[26] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[27] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 551–562. ACM, 2003.

[28] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1789–1792. IEEE, 2016.

[29] P. Clifford and I. Cosma. A simple sketching algorithm for entropy estimation over streaming data. In *AISTATS*, pages 196–206, 2013.

[30] G. Cormode and A. McGregor. Approximation algorithms for clustering uncertain data. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 191–200. ACM, 2008.

[31] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[32] M. Costa, A. L. Goldberger, and C.-K. Peng. Multiscale entropy analysis of complex physiologic time series. *Physical review letters*, 89(6):068102, 2002.

[33] T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[34] F. da Costa, R. Rios, and R. de Mello. Using dynamical systems tools to detect concept drift in data streams. *Expert Systems with Applications*, 60:39–50, 2016.

[35] G. A. Darbellay. An estimator of the mutual information based on a criterion for conditional independence. *Computational Statistics & Data Analysis*, 32(1):1–17, 1999.

[36] C. O. Daub, R. Steuer, J. Selbig, and S. Kloska. Estimating mutual information using B-spline functions–an improved similarity measure for analysing gene expression data. *BMC bioinformatics*, 5(1):118, 2004.

[37] A. Dionisio, R. Menezes, and D. A. Mendes. Mutual information: a measure of dependency for nonlinear time series. *Physica A: Statistical Mechanics and its Applications*, 344(1):326–329, 2004.

[38] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80. ACM, 2000.

[39] P. Domingos and G. Hulten. A general method for scaling up machine learning algorithms and its application to clustering. In *ICML*, volume 1, pages 106–113, 2001.

[40] P. M. Domingos and G. Hulten. Catching up with the data: Research issues in mining data streams. In *DMKD*, 2001.

[41] F. Farnstrom, J. Lewis, and C. Elkan. Scalability for clustering algorithms revisited. *ACM SIGKDD Explorations Newsletter*, 2(1):51–57, 2000.

[42] D. H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine learning*, 2(2):139–172, 1987.

[43] A. M. Fraser and H. L. Swinney. Independent coordinates for strange attractors from mutual information. *Physical review A*, 33(2):1134, 1986.

[44] A. L. Freire, G. A. Barreto, M. Veloso, and A. T. Varela. Short-term memory mechanisms in neural network learning of robot navigation tasks: A case study. In *Robotics Symposium (LARS), 2009 6th Latin American*, pages 1–6. IEEE, 2009.

[45] J. Gama. *Knowledge discovery from data streams*. CRC Press, 2010.

[46] J. Gama and M. Gaber. *Learning from Data Streams: Processing Techniques in Sensor Networks*. Springer, 2007.

[47] J. Gama, P. Medas, and P. Rodrigues. Learning decision trees from dynamic data streams. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 573–577. ACM, 2005.

[48] J. Gama and C. Pinto. Discretization from data streams: applications to histograms and data mining. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 662–667. ACM, 2006.

[49] J. Gama, R. Sebastião, and P. P. Rodrigues. On evaluating stream learning algorithms. *Machine Learning*, 90(3):317–346, 2013.

[50] S. Gao, G. Ver Steeg, and A. Galstyan. Efficient estimation of mutual information for strongly dependent variables. In *AISTATS*, 2015.

[51] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *VLDB*, volume 1, pages 79–88, 2001.

[52] A. L. Goldberger, L. A. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. Physiobank, physiotoolkit, and physionet components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000.

[53] Google Cloud Platform. Cloud dataflow. `https://cloud.google.com/dataflow/`, 2017. note = Accessed: 2017-01-30.

[54] P. Hall and S. C. Morton. On the estimation of entropy. *Annals of the Institute of Statistical Mathematics*, 45(1):69–88, 1993.

[55] M. Han, W. Ren, and X. Liu. Joint mutual information-based input variable selection for multivariate time series modeling. *Engineering Applications of Artificial Intelligence*, 37:250–257, 2015.

[56] Y. Hang and S. Fong. Stream mining dynamic data by using iOVFDT. *Journal of Emerging Technologies in Web Intelligence*, 5(1):78–86, 2013.

[57] S. Hashemi, Y. Yang, Z. Mirzamomen, and M. Kangavari. Adapted one-versus-all decision trees for data stream classification. *Knowledge and Data Engineering, IEEE Transactions on*, 21(5):624–637, 2009.

[58] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 97–106. ACM, 2001.

[59] E. Ikonomovska and J. Gama. Learning model trees from data streams. In *Discovery Science*, pages 52–63. Springer, 2008.

[60] E. Ikonomovska, J. Gama, and S. Džeroski. Learning model trees from evolving data streams. *Data mining and knowledge discovery*, 23(1):128–168, 2011.

[61] W. B. Johnson and J. Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary mathematics*, 26(189-206):1, 1984.

[62] B. Kaluža, V. Mirchevska, E. Dovgan, M. Luštrek, and M. Gams. An agent-based approach to care in independent living. In *Ambient Intelligence*, pages 177–186. Springer, 2010.

[63] F. Keller, E. Müller, and K. Böhm. Estimating mutual information on data streams. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, page 3. ACM, 2015.

[64] S. Khan, S. Bandyopadhyay, A. R. Ganguly, S. Saigal, D. J. Erickson III, V. Protopopescu, and G. Ostrouchov. Relative performance of mutual information estimation methods for quantifying the dependence among short and noisy data. *Physical Review E*, 76(2):026209, 2007.

[65] D. E. Knuth. *The Art of Computer Programming, volume 2: Seminumerical Algorithms, 3rd edn., p. 232.* Boston: Addison-Wesley., 1998.

[66] L. Kozachenko and N. N. Leonenko. Sample estimate of the entropy of a random vector. *Problemy Peredachi Informatsii*, 23(2):9–16, 1987.

[67] A. Kraskov, H. Stögbauer, and P. Grassberger. Estimating mutual information. *Physical review E*, 69(6):066138, 2008.

[68] H.-P. Kriegel, T. Bernecker, M. Renz, and A. Züfle. *Probabilistic Join Queries in Uncertain Databases (A Survey of Join Methods for uncertain data)*, volume 35. Springer, 2010.

[69] H.-P. Kriegel and M. Pfeifle. Density-based clustering of uncertain data. In *Proceedings of the eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 672–677. ACM, 2005.

[70] C. Liang, Y. Zhang, and Q. Song. Decision tree for dynamic and uncertain data streams. In *ACML*, pages 209–224, 2010.

[71] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[72] S. Muthukrishnan et al. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005.

[73] W. K. Ngai, B. Kao, C. K. Chui, R. Cheng, M. Chau, and K. Y. Yip. Efficient clustering of uncertain data. In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, pages 436–445. IEEE, 2006.

[74] C. Olah. Visual information theory. `http://colah.github.io/posts/2015-09-Visual-Information/`, 2015. note = Accessed: 2017-01-30.

[75] A. Oulasvirta, T. Roos, A. Modig, and L. Leppänen. Information capacity of full-body movements. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1289–1298. ACM, 2013.

[76] L. Paninski. Estimation of entropy and mutual information. *Neural computation*, 15(6):1191–1253, 2003.

[77] B. Pfahringer, G. Holmes, and R. Kirkby. New options for Hoeffding trees. In *AI 2007: Advances in Artificial Intelligence*, pages 90–99. Springer, 2007.

[78] B. Qin, Y. Xia, and F. Li. DTU: a decision tree for uncertain data. In *Advances in Knowledge Discovery and Data Mining*, pages 4–15. Springer, 2009.

[79] J. Quinlan. C4.5: Programs for machine learning., 1993.

[80] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[81] L. R. Rabiner and B. Gold. Theory and application of digital signal processing. *Englewood Cliffs, NJ, Prentice-Hall, Inc., 1975. 777 p.*, 1, 1975.

[82] D. N. Reshef, Y. A. Reshef, H. K. Finucane, S. R. Grossman, G. McVean, P. J. Turnbaugh, E. S. Lander, M. Mitzenmacher, and P. C. Sabeti. Detecting novel associations in large data sets. *Science*, 334(6062):1518–1524, 2011.

[83] Y. Sakurai, S. Papadimitriou, and C. Faloutsos. Braid: Stream mining through group lag correlations. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 599–610. ACM, 2005.

[84] M. Sax. Storm compatibility in Apache Flink: How to run existing Storm topologies on Flink. `https://flink.apache.org/news/2015/12/11/storm-compatibility.html`, 2015. note = Accessed: 2017-01-30.

[85] A. Seliniotaki, G. Tzagkarakis, V. Christofides, and P. Tsakalides. Stream correlation monitoring for uncertainty-aware data processing systems. In *Information, Intelligence, Systems and Applications, IISA 2014, The 5th International Conference on*, pages 342–347. IEEE, 2014.

[86] C. E. Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[87] C. E. Shannon. Prediction and entropy of printed English. *Bell Labs Technical Journal*, 30(1):50–64, 1951.

[88] SILSO World Data Center. The international sunspot number. *International Sunspot Number Monthly Bulletin and online catalogue*, 2016.

[89] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. Hambrusch. Indexing uncertain categorical data. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 616–625. IEEE, 2007.

[90] A. Sorjamaa, J. Hao, and A. Lendasse. Mutual information and k-nearest neighbors approximator for time series prediction. *Artificial Neural Networks: Formal Models and Their Applications–ICANN 2005*, pages 752–752, 2005.

[91] H. Späth. *Cluster analysis algorithms for data reduction and classification of objects*. Horwood, 1980.

[92] S. J. Stolfo, W. Fan, W. Lee, A. Prodromidis, and P. K. Chan. Cost-based modeling for fraud and intrusion detection: Results from the JAM project. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 130–144. IEEE, 2000.

[93] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[94] The NASDAQ Stock Market. NASDAQ daily quotes. `http://www.nasdaq.com/quotes/nasdaq`, 2015. note = Accessed: 2015-03-01.

[95] S. Tsang, B. Kao, K. Y. Yip, W.-S. Ho, and S. D. Lee. Decision trees for uncertain data. *Knowledge and Data Engineering, IEEE Transactions on*, 23(1):64–78, 2011.

[96] J. Walters-Williams and Y. Li. Estimation of mutual information: A survey. In *International Conference on Rough Sets and Knowledge Technology*, pages 389–396. Springer, 2009.

[97] P. Wang, H. Wang, X. Wu, W. Wang, and B. Shi. On reducing classifier granularity in mining concept-drifting data streams. In *Data Mining, Fifth IEEE International Conference on*, pages 8–pp. IEEE, 2005.

[98] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[99] Y. Yang, X. Wu, and X. Zhu. Combining proactive and reactive predictions for data streams. In *Proceedings of the eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 710–715. ACM, 2005.

[100] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the 4th USENIX Workshop on Cloud Computing*, 2012.

[101] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 358–369. VLDB Endowment, 2002.