Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Incident detection on SIEM events

Petr Poliak

Field of study: Open informatics
Subfield: Informatics and computer science

Supervisor: Štěpán Kopřiva, MSc.

May 2018

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Poliak Petr**	Personal ID number:	**439562**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Branch of study:	**Computer and Information Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Incident Detection on SIEM Events

Bachelor’s thesis title in Czech:

Detekce incidentů nad SIEM událostmi

Guidelines:

1) Study the provided SIEM (Security information and event management) event log dataset and the dataset of labeled incidents (lists of events) of various types. The incidents were generated (labeled) by an existing rule-based system. The ratio of the incidents to valid event sequences is very limited.
2) Design a classifier (or a set of classifiers), which classifies a sequence of incoming events either as an incident or valid sequence.
3) Implement the classifier designed in point 2).
4) Evaluate the classifier on a test dataset, measure performance for each incident type and identify potential incidents from unlabeled data.
5) Optionally deploy the classifier on the distributed computational framework (Apache Spark) to decrease the runtime of classification.

Bibliography / sources:

[1] He, Habib, and Edwardo Garcia. 2009. Learning from Imbalanced Data. IEEE Transactions on Knowledge and Data Engineering 21 (9): 1263-1284.
[2] Wang, Shu, and In Tao. 2012. Multi Class Imbalance Problems: Analysis and Potential Solutions. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics 42 (4): 1119-1130.
[3] N.V. Chawla, N. Japkowicz, A. Kotcz, Editorial: special issue on learning from imbalanced data sets, SIGKDD Explorations 6 (1) (2004) 1-6.

Name and workplace of bachelor’s thesis supervisor:

Štěpán Kopřiva, MSc., Artificial Intelligence Center, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **28.02.2018** Deadline for bachelor thesis submission: **25.05.2018**

Assignment valid until: **30.09.2019**

Štěpán Kopřiva, MSc.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I want to thank everyone from the Facebook chats where I could cry my heart out during the past months' struggles.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 20. May 2018

Abstract

In this thesis, we focus on designing a pipeline with a set of classifiers that will be able to classify a sequence of SIEM logs to flag incidental behavior. Additionally, we tackle a problem of imbalanced learning as the used dataset does not contain many incidents compared to the total size. The pipeline's input will be a sequence of SIEM events, and the result will be an alert type if any. Lastly, it contains the performance of presented designs along with evaluation and an optimal method how to classify input sequences.

Keywords: Imbalanced learning, Sequence classification, SIEM incident classification

Supervisor: Štěpán Kopřiva, MSc.

Abstrakt

Tato práce je zaměřená na návrh pipeline (s několika klasifikátory), která bude schopná zpracovat řadu SIEM logů a klasifikovat pochybné chování. Zároveň řešíme problém učení z nevyváženého datasetu, protože poskytlá data obsahují málo incidentů v porovnání s celkovou velikostí datasetu. Pipeline bude brát za vstup sekvenci SIEM událostí a vracet typ incidentu, pokud se nějaký objeví. Nakonec vyhodnotíme úspěšnost navrhnutých klasifikátorů.

Klíčová slova: Nevyvážené datasety, Klasifikace řad, SIEM detekce incidentů

Překlad názvu: Detekce incidentů nad SIEM událostmi

Contents

Figures

Tables

Chapter 1

Introduction

In a corporate environment, it is a widespread practice to log all entities' actions in a system. Be it the services providing functionality to the users, or the users themselves. The reason behind that is to prefer having some useless data to missing critical information.

The logs then later serve several purposes. We can use them to optimize the network (e.g., when throttling), improve the availability of services when they are overloaded, or monitor for suspicious or undesirable behavior.

After the logs are collected there is usually a group of analysts who either manually search for anomalies or design an automatic rule-based system to flag certain kinds of behavior and then verify whether it is a false alarm.

We are given logs from such a system, and we want to apply machine learning methods to it. Among the benefits we have knowledge distillation [8] to iteratively improve the systems' capabilities. Due to the nature of the data, we have two problems that we tackle in this thesis. Because we are logging behavior over time, we have to be able to accept variable length input data. Secondly, because the systems need to be secure, there aren't many positive samples data. Thus we have a high imbalance of different classes samples.

Our goals for this thesis are to analyze the labeled dataset, design and implement a pipeline to classify incidents from sequential data with high between-class imbalance, and lastly evaluate the design on a suite of experiments.

1.1 Structure of the thesis

The thesis is organized as follows:

Chapter 2 introduces the necessary background needed for this thesis. In three parts it explains *Imbalanced learning*, *Feature selection*, and *Sequence classification*.

Chapter 3 addresses the first assignment of studying the dataset and analyzing the problem.

Chapter 4 contains the proposed models of the classifiers that solve the problems from Chapter 3 for the second assignment. It describes two approaches and then in Section 4.4 it describes the implementation together with the technologies used for the third assignment.

Chapter 5 addresses the fourth assignment and evaluates the performance of proposed models. It describes the pipeline and methodology used for testing and the results of the experiments.

Chapter 2

Technical Background

In this chapter, we will introduce the necessary terms and methods that were needed during solving of this work. In the first section we will describe what is *Imbalanced learning* and what problems it introduces and what are the techniques used to deal with it. Next, we will briefly explain *Feature selection*. What it does and how it is useful. And in the last part, we describe different kinds of *Sequence classification* problems and their standard solutions.

2.1 Imbalanced learning

It is reasonable to assume some level of imbalance in any real-world dataset used for machine learning. Imbalanced learning [7] focuses on datasets that are imbalanced in the individual classes' sample sizes in ranges like 100:1 to 10000:1 where one class dominates the other. There are two cases of imbalanced problems we can encounter. The first considers only a binary classification in which case we refer to them as the majority and minority classes. The second we usually refer to as between-class imbalances where either at least one class dominates the others, or there is at least one under-represented class.

Imbalanced learning also deals with the problem called within-class imbalance. This means that a class has subconcepts that aren't sufficiently represented in the dataset. To demonstrate this imagine a problem of animal classification from an image. Let's focus for now on a subproblem of differentiating between horses and dogs. Let's assume the between-class imbalance is negligible. Now if the dataset showcased a within-class imbalance, it could be that it contains only the pictures of white horses and black dogs. This could introduce errors such as not recognizing the animal in the image or incorrectly classifying black horse as a dog. Demonstrated with stars and circles in Figure 2.1.

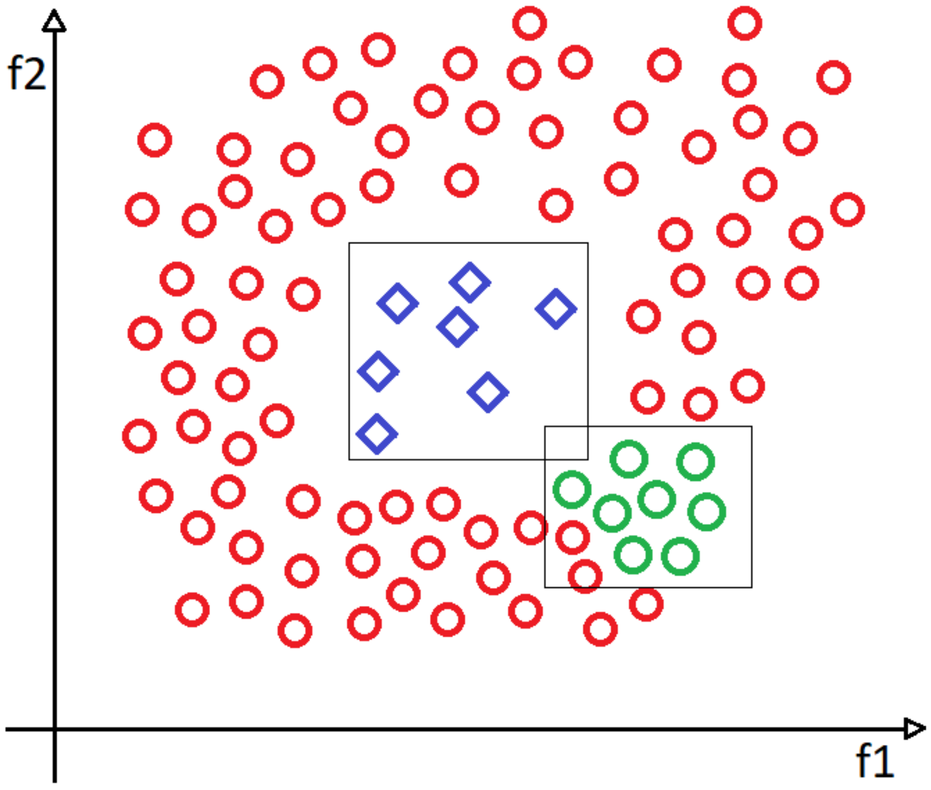

Figure 2.1: We can see both —between-class imbalance (between circles and diamonds) as well as within-class imbalance demonstrated by different colors of circles. The two imbalances are highlighted by rectangles around them.

Since the dataset for this thesis is free from within-class imbalance (each class has enough samples to showcase all properties), the following text focuses on between-class imbalance.

Generally, there are two approaches to dealing with between-class imbalance. The first approach focuses on resampling the dataset by either oversampling —generating new samples with the knowledge from the original samples —or undersampling the majority class thus balancing the ratio of the classes. The second approach is alternating the respective cost of misclassifying either class.

2.1.1 Evaluation techniques for Imbalanced learning

The problem imbalanced datasets introduce is that we cannot use basic evaluation methods (for both learning and model evaluation). For example if we were using simple error rate

$$\epsilon_r = \frac{number\ of\ incorrectly\ classified\ samples}{total\ number\ of\ samples}$$

while having an imbalanced dataset of 1000:1 in a two class classification problem, any classifier with strategy of $\delta(x) = majority\ class$ would have an error $\epsilon_r \approx 0.001$ which could be considered great in most cases but the classification would have no informational value.

Precision and Recall metrics. One approach to measuring the quality of prediction is *Precision* and *Recall* which are on a binary classification problem defined as

$$P = \frac{TP}{TP + FP}$$

$$R = \frac{TP}{TP + FN}$$

A system with high recall but low precision returns many results, but most of its predicted labels are incorrect when compared to the training labels. A system with high precision but low recall is just the opposite, returning very few results, but most of its predicted labels are correct when compared to the training labels. An ideal system with high precision and high recall will return many results, with all results labeled correctly. (Paragraph taken from [1]).

Receiver Operating Characteristics curves. The *ROC* method compares two characteristics —true positive rate and false positive rate where

$$TP\_rate = \frac{TP}{P_C} \quad FP\_rate = \frac{FP}{N_C}$$

An *ROC* graphical plot is obtained [7] by plotting *TP_rate* over *FP_rate*. A point is an assessment of a single hard classifier. An *ROC* curve is obtained for soft classifiers and signifies the tradeoff for benefits vs costs. When plotted with the origin at $(0, 0)$ and *TP_rate* on the y axis the optimal point lies in top left corner $(FP\_rate, TP\_rate) = (0, 1)$. When concerned with *ROC* curves we usually compare the *Area Under Curve* as the metric to compare two classifiers.

2.1.2 Sampling methods for Imbalanced datasets

As mentioned above —sampling methods are divided as over- and undersampling.

Random sampling. The most straightforward approach is random over-/under-sampling. For oversampling we simply replicate examples from minority class until we achieve the desired ratio. Undersampling randomly selects cases from the majority class and excludes them from the current set.

Disadvantages are apparent —naive undersampling can lose some information valuable to the classification. Oversampling is prone to overfitting and also increases the relevance of outliers and malicious samples.

Synthetic minority oversampling technique. The SMOTE algorithm generates new entries by interpolating data points from minority classes. For example, let's have a sex classification problem based on the height of a person. If we have two points $(180, M)$ and $(190, M)$ it is reasonable to assume that there probably exists a point with height in the interval $[180; 190]$ and class M.

Undersampling using Tomek links. Tomek links (or T-links) are used to clean the border between clusters of different classes. If we have the distance between two points $d(x, y)$ defined we call a pair (x, y) a T-link if following conditions hold: (1) x and y are of different classes (2) there is no such point z such that $d(x, z) < d(x, y)$ and $d(y, z) < d(x, y)$. This means that for some point the closest other point is of a different class thus one of those points either being an outlier/noise or both points being on the border of the two clusters.

Removing Tomek links leaves us with a cleaner dataset (less noisy) and better-separated clusters that can improve the classification.

Combining oversampling with data cleaning. Better results can be achieved by combining both informed under- and over-sampling. By first generating new samples using the SMOTE algorithm to better balance the classes and then cleaning the dataset using T-links an improved dataset is obtained as can be seen in Figure 2.2.

2.1.3 Cost-sensitive methods

Where *sampling* methods try to mitigate the dataset imbalance by changing the distribution of samples *cost-sensitive* methods change the penalty for misclassification in favor for the minority class/es. For example, when using the AdaBoost algorithm, we can alter the reweighing of samples to incorporate the imbalance mitigating costs or when using neural networks we can weight the output neurons differently or change the error function to reflect the cost.

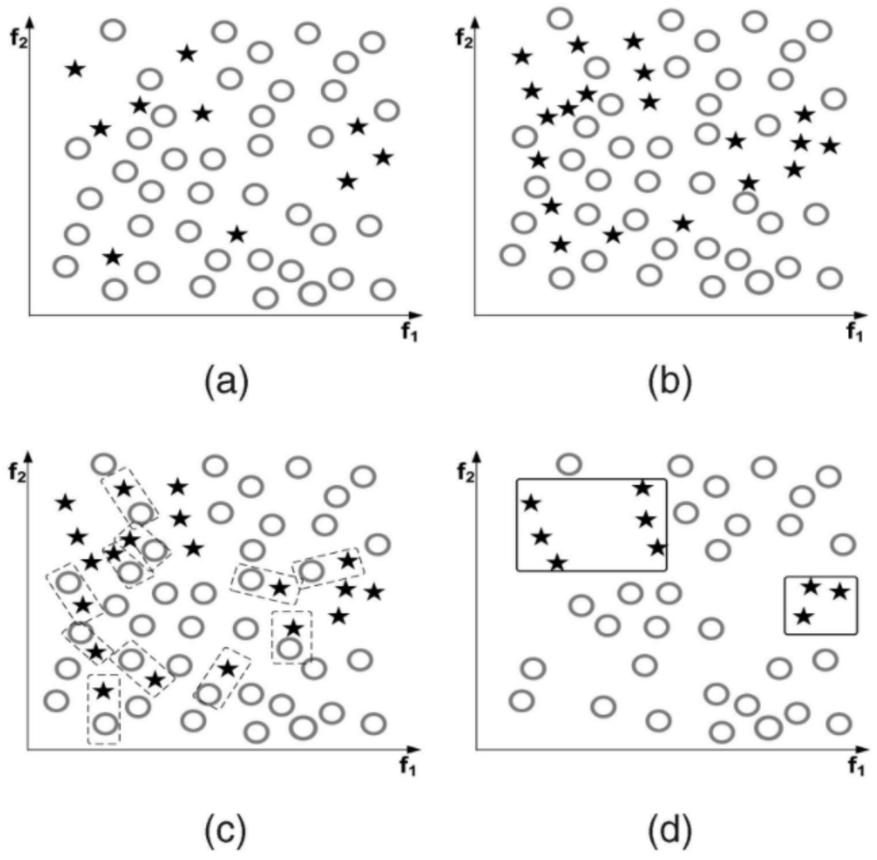

Figure 2.2: Combining SMOTE and Tomek links. In (a) we can see the original imbalanced dataset. (b) Shows the transformation after generating sample using the SMOTE algorithm. (c) Contains the marked T-links. (d) is the final result after removing T-links.
Source: [7].

2.2 Feature selection

Feature selection is a field focusing on selecting a subset of features from the original feature vector while retaining good prediction results. The goal is to remove features that don't provide much information. These can be dependent variables which don't contain any additional knowledge or features which are very noisy and thus reducing the generalization of the model.

The benefits of removing features are improved generalization and the reduction of dimensionality. The difference from simple dimensionality reduction techniques (for example PCA) is that we are reducing the dimensionality by removing axes (the whole individual features) and leaving us with a fewer number of features, not just a lower dimension but of some artificial variables which don't have an interpretation.

Generally, there are three categories of feature selection —filter methods, wrapper methods, and embedded methods which differ in the feature selection criterion. We briefly describe each below.

2.2.1 Filter methods

Filter methods are considered a baseline [6] regarding variable selection. They use methods of statistical analysis to rank variables how relevant they are to the labeling. The variables are then filtered out when not meeting a threshold. Some examples include Pearson correlation coefficient

$$R(i) = \frac{cov(X_i, Y)}{\sqrt{var(X_i)var(Y)}}$$

where X_i represents the values of a feature i and Y represents the labels respective to the feature vectors.

However, [6] showcased that two features that are useless by themselves can have information when combined. Observe Figure 2.3 with the XOR problem. We can see that histogram of neither axis contains information to construct a discriminative classifier. Looking at the original distribution, we can see that the 4 clusters are well separated to classify.

Another disadvantage of Filter methods is that they only consider each variable independently. This means that they will not omit redundant variables which highly correlate to an already selected variable. A simple case would be to duplicate a feature —since the original feature passed the threshold test (and was considered relevant) the duplicate will pass it as well thus the algorithm will not filter out a redundant variable.

Filter methods are still a useful technique though as they are independent of the choice of the predictor and can be used as a pre-processing step.

2.2.2 Wrapper methods

Whereas *Filter* methods rank variables and then filter the worst out, *Wrapper* methods aim to pick the best subset of them. To do so, we evaluate a selected subset of features using a classifier as a black box and the performance of it as the objective function we are trying to maximize. Compared to filter methods this adds the relations between features to the evaluation, thus providing better results.

With N being the number of features there is 2^N number of subsets, so we are limited by the computational ability of our system to try all

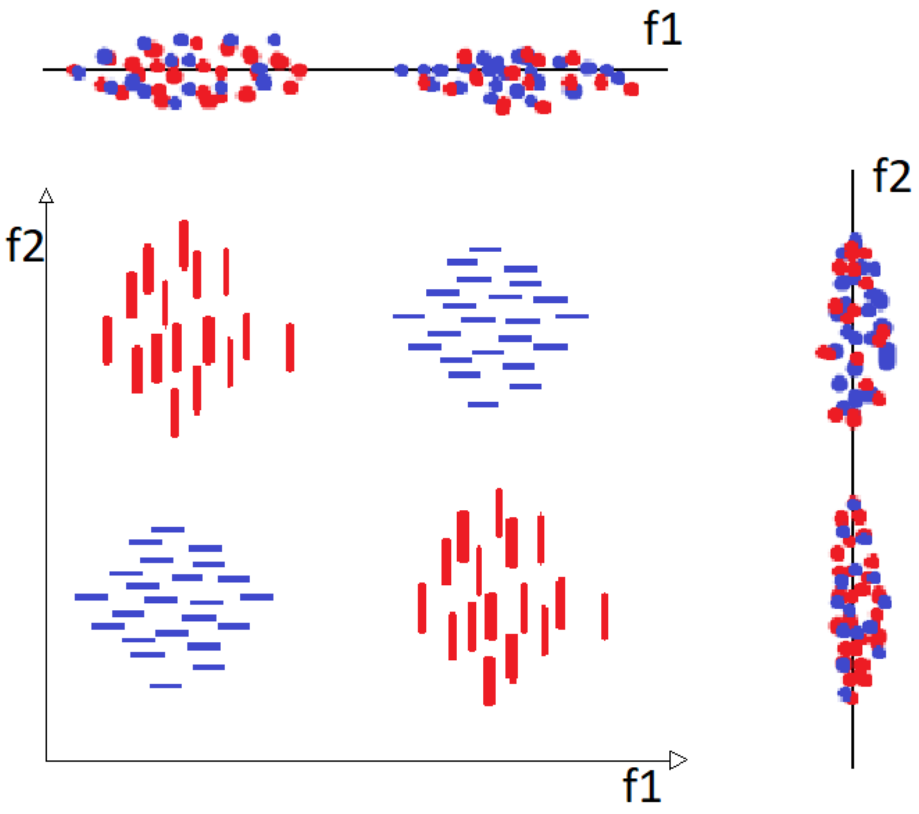

Figure 2.3: When we look at the XOR problem we can see that both axes alone are useless. Together, however, we can see that the problem is composed of well separated clusters of respective classes.

possibilities only for a small N. Thus, we employ heuristical search algorithms to obtain a sufficient but possibly sub-optimal solution.

The methods are divided [4] into two categories first being the *Sequential selection algorithms* and second being the *Heuristic search algorithms*.

Sequential selection algorithms. This is a group of iterative algorithms that construct the final subset by adding one feature at a time. Both forward and backward approaches have been suggested.

Forward approaches construct the set from one element upwards. Each iteration we try to add one additional feature to the current subset and evaluate it. The best *new* subset is then selected for the next iteration.

Backward search uses the opposite approach. We start with the complete set of features and each iteration we remove one feature that has the smallest decrease in the objective function (the performance of the predictor). We repeat this process until the desired number of features has been achieved.

It has been proposed [16] to use *Floating search methods*. This process starts as a forward selection algorithm but incorporates a backtracking step. Each iteration we also try to remove a feature from the current subset if we can improve the performance of the predictor from the previous $n-1$ sized subset. We can imagine this as swapping one of the features in the current subset with the current feature we want to add to improve the result instead of just adding it.

Heuristic search algorithms. A genetic algorithm approach is given [4] as an example of a Heuristic search. We construct N random subsets of the length k we want at the end, and then the genetic algorithm is used to maximize the predictor's performance. A modified (more aggressive) version of GA —CHCGA is recommended.

2.2.3 Embedded methods

Where *Wrapper methods* gain in results, they lose in time complexity and computational demands over *Filter methods*. *Embedded methods* try to take the advantages of Wrapper methods to evaluate whole subsets of features and reduce the time complexity of iterative model re-training and performance evaluation by incorporating the feature selection to the model learning.

Mutual information (MI) is presented:

$$I(Y, X) = H(Y) - H(Y|X)$$

where $H(Y)$ is the Shannon entropy of Y and $H(Y|X)$ is the conditional entropy. A basic objective function for an embedded method is then

$$I(Y, f) - \beta \sum_{s \in S} I(f, s)$$

where "Y is the output, f is the current feature, S is the set of already selected features and β controls the importance of the MI between the current feature and the selected features from S" [4].

2.3 Sequence classification

The literature [19] divides *Sequence classification* into two different cases. The first is called *conventional* sequence classification and it concerns with the whole sequence having one label. The second one instead of classifying the whole sequences tries to classify each element of a sequence with a label is called a *strong* sequence classification.

Each field has many applications [19] lists examples for conventional sequence classification a DNA sequence (coding/noncoding area) or ECG data (healthy/unhealthy patient) and for the strong classification streaming sequences or sequences where we observe a change in state.

The classification methods can be divided into three categories —feature based classification, sequence distance-based classification, and model-based classification [19].

2.3.1 Feature based classification

The first method tries to remove the original restrictions that prohibit us from using regular methods of machine learning (decision trees, neural networks) by representing/compressing a sequence to a feature vector.

k-grams are introduced [19] as a sequence of k consecutive symbols. The sequence can be then transformed into a vector by the presence of each *k-gram* or by its frequency. Instead of *k-grams* an improved way to select n-tuples of symbols is presented. Instead of all *k-grams* we select patterns of subsequences while following certain rules that optimize the distinction between classes. The accuracy is reported to improve by 10-15%.
The problem of representing sequence by smaller subsequences is that we lose some global properties of the sequence by storing only local patterns so the usage of *wavelet decomposition* is noted [19]. Wavelet decomposition allows containing the information of the sequence on multiple resolutions thus providing better context for classification of the whole sequence.

2.3.2 Sequence distance methods

Sequence distance methods make use of distance-based machine learning models. First, a distance function is defined which is subsequently used in some standard algorithm. An example is given of K-nearest neighbors or SVM [19]. Because KNN is a lazy classifier that uses the training data to classify directly, the choice of the distance function is computationally reflected in the performance of the classification.

The first and most intuitive distance metric is *Euclidean distance*. For two sequences a and b of length l we compute it as

$$d(a, b) = \sqrt{\sum_{i=0..l} (a_i - b_i)^2}$$

Another distance metric is called *Dynamic time warping* (DWT) and handles cases when we have two sequences that are very similar but with a differently skewed axis. DWT tries to find the closest distance by aligning the two sequences. See Figure 2.4 for visualization. DWT is calculated using dynamic programming and is an expensive distance function that should be used when Euclidean distance is not providing good results.

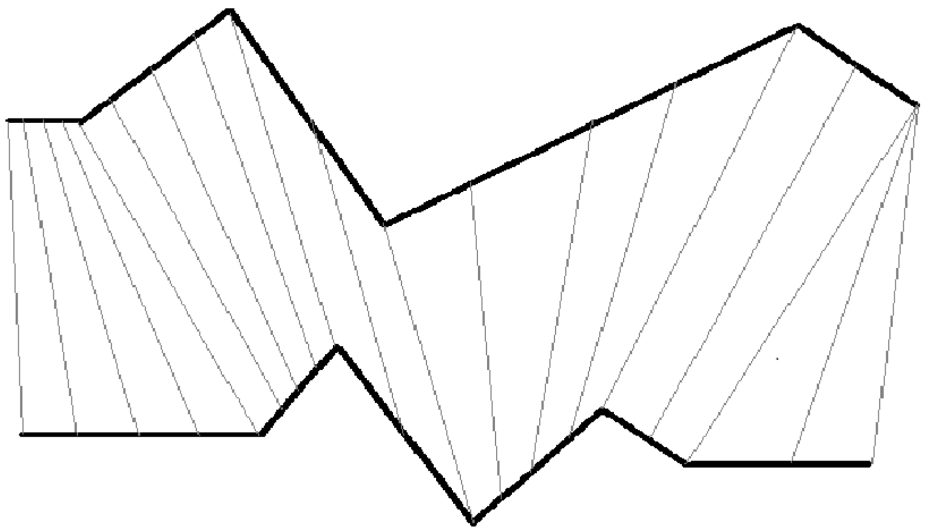

Figure 2.4: Dynamic time warping example where we can see how the two series are linked to minimize the distance.

2.3.3 Model based classification

The last mentioned [19] way of classifying sequences is by using generative models. It assumes that a class of sequences is generated by some underlying probability distribution. We then define the model using domain knowledge or assumptions and train the parameters of such model. When classifying a sequence, we assign it a class with the maximum likelihood.

As in many other classification problems, there is a *Naive Bayes classifier*. It assumes that the events are independent of each other which is in many cases violated. However, it still performs very well with its simplicity taken into account.

Another models cited are *Markov Model* and *Hidden Markov Model*. The latter assumes that the sequence is a Markov process with unobserved states. The use of profile HMM is noted with three types of states of which two are *gap* states —match state (the correct element is present), insertion state (another element is inserted that is missing in sequence model) and deletion state (an element is “deleted” in the sequence model).

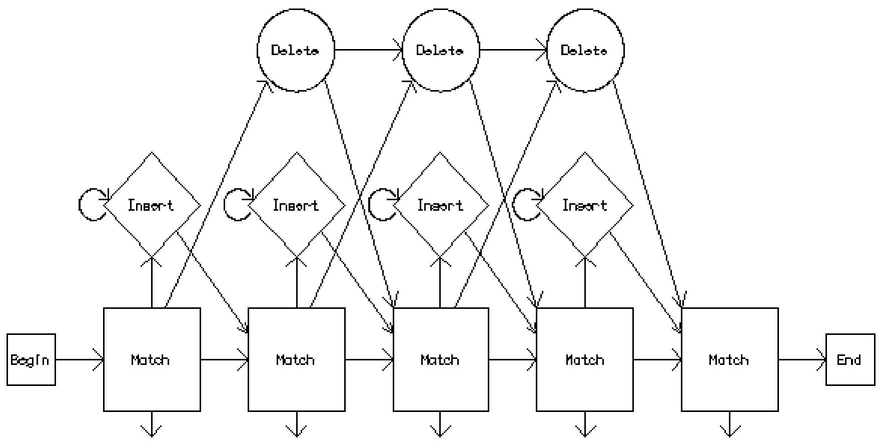

Figure 2.5: An example of profile HMM with three states —match, insertion and deletion. We can see how deletion state “skips” the next match element in a sequence and insertion adds element(s) before the next match.
Source: [10].

LSTM Neural Networks. The rationale behind *Recurrent neural networks* is that when a human tries to understand for example a paragraph of text, he does not read it as a sequence of individual words but read the text as a whole. When the person is reading a word, he has the previous words in memory in a context of the whole sentence.

LSTM or *Long short-term memory* neural networks improve this idea by providing an option to regulate the importance of the *recurring loop*. In Figure 2.6 we can see a detail of an unrolled recurrent neural network. The arrows in and out of the network (left and right) symbolize the loop, the memory or cell state on top and the result h_t on the bottom. Important is the top line where the cell state is modified by the current input which is then passed to the next computation (loop). For a more detailed explanation, we redirect the reader to the original article [13].

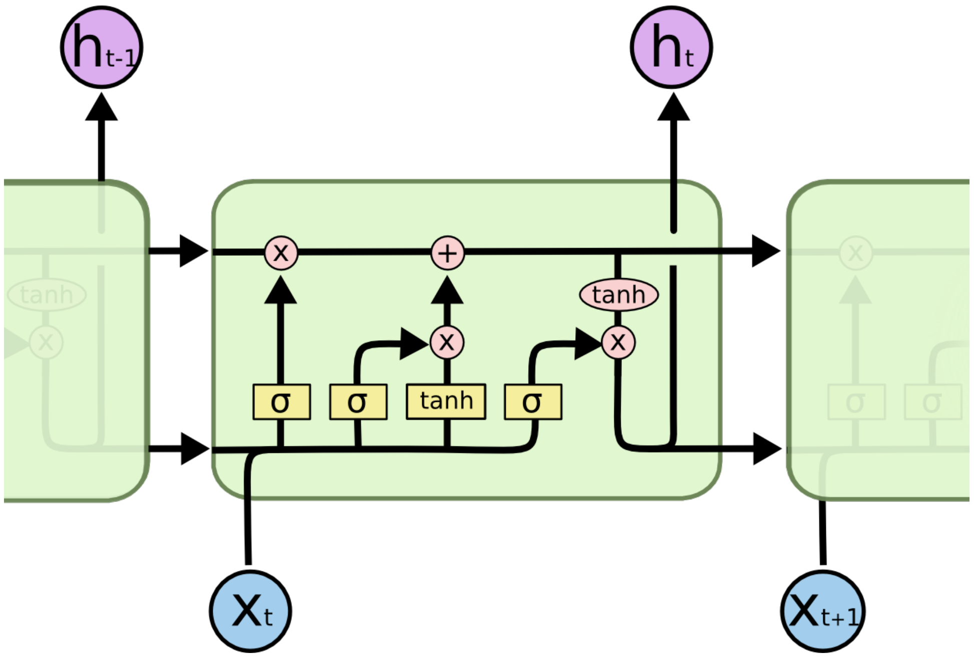

Figure 2.6: Detail of an unrolled lstm neural network. We can see in detail how the information of state and result are passed into the next computation. Source: [13].

Chapter 3

Problem Analysis

The chapter defines the problem we are going to solve in this thesis. Firstly we provide the reader with the information surrounding the issue such as the origin of data and what we are trying to solve. After giving the background information, a formalization of the problem is provided. Then we briefly describe the format of the dataset and lastly outline the solution.

3.1 Problem settings

For corporate enterprise systems, it is a prevalent practice to log vast amounts of data about what is happening in the system and on the network. While logging everything is not much useful very often we can focus on specific subsets of logs determined by our domain knowledge of the problem we are trying to solve.

In our problem, we have the activity logs of users (and internal services). The logs are of resources they are using and how they communicate. What that means is that any agent in the network when accessing a resource (e.g., a file on a shared drive) this action is logged because the resource is only available while being authenticated and authorized.

Generally, each log contains the information of the origin, the event type (signifying what is happening) and the target of the action (where applicable). These logs are created in a service called *Active directory*. (We refer the reader to the internet if a deeper understanding of the system is desired.) The Active directory service handles both authentication and authorization thus when accessing any resource or producing any action we have a place through where everything needs to be routed. The logs are then collected from multiple such instances to provide a single stream of events along with *event time* (also called *log time*) and the *collected time* when was the log received by the collector. Section 3.3 describes the dataset in more detail as we now focus more on the general description of the system.

Incident origin. After these logs are collected multiple queries designed by specialists are run. These queries contain the information about the behavior we are trying to target that could be both the insufficiency and breach of the system. An artificial example of this could be a user overloading the network —firstly a well-behaved user should not be doing that (breach) secondly a single user should not have the permissions to do that (insufficiency of the system).

The task at hand is the following —having sequences of events classified by the given rule-based classifier design a model that will be able to replicate the desired behavior.

Purpose of such system. A reader might now object as to what is the purpose of using machine learning when we already have the original objective function to classify the records? That is a very valid question, and the answer to that is not universal to all such problems. The role of machine learning is that it can draw patterns from data where it is not apparent. This along with the use of an interpretable classifier can be used to enhance the system's design iteratively. The other great property of machine learning is the generalization. Because the analysts had to design the classifier with data analysis and experience they had to make certain decisions such as bounds for when "too much" is actually "too much". The model thus generalizes these rules and makes the bounds softer. This produces some level of false positives but also catches behavior that would go undetected through the system by acting slightly below the hard threshold.

3.2 Problem formalization

Let's now strip away the specificities and focus on the underlying problem abstractly. We have a system that produces events. We then have some *continuous* subsequences classified as incidents. The goal is to create a machine learning model that will learn from the provided labeled sequences and which can classify an incoming series of events as either normal or mark the series as abnormal and why.

We want to create a model g that given a sequence S_i of length n:

$$S_i^n = e_1, e_2, \dots, e_n$$

it assigns the sequence a label such that we minimize the risk R on training set T

$$R(g) = \frac{1}{N} \sum_{i \leq |T|} L(y_i, g(S_i))$$

Where T is the training set of pairs (S_i, y_i) of sequences and corresponding labels and L is the loss function of classifying labeling of sequence S by model g.

3.3 Dataset overview

In listing 3.1 we provide an example log message in JSON format.

We can notice a pattern that there are several metadata fields, usually beginning with an underscore such as `_tz` for timezone, etc. There are several fields in the type of `_type_X` which are then a space-separated list of other fields we can find in the object. The types are `str`, `num` and `ip` and they signify the type of the fields in the object.

There are important fields such as `norm_id` which tells us the origin of the message (the service which generated the log) and `user` or `caller_sid` which identify the originator of the event. Other notable fields are `log_ts` –time of the event used for ordering the events into a single stream, or `event_id` –what is an id of the type of the event being logged.

Additionally, there are several fields without any informational value. These fields are usually a human-readable extension, e.g., a description or a message. We can safely discard them during the feature extraction because they correlate with a coefficient 1 with some other field. For example, description will always be the same for each `event_id`. Even if it changed in the future, it would be the same from then onwards.

```
{
  "description": "This event is generated when a logon
     session is destroyed",
  "event_source": "Microsoft-Windows-Security-Auditing",
  "log_ts": 1518525895,
  "_type_str": "msg device_name collected_at device_ip
     col_type _tz _enrich_policy domain event_source
     event_type caller_sid user object norm_id label host
     event_log logon_id event_category action message
     description _fromV550",
  "device_name": "RUMOSDCW01",
  "logon_id": "0x7583bfa",
  "event_type": "Success Audit",
  "_offset": 32741,
  "host": "RUMOSDCW01.Disagroup.net",
  "action": "logged off",
  "caller_sid": "S
     -1-5-21-2320514144-3261309781-1462386680-119625",
  "col_ts": 1518515096,
  "severity": "6",
  "_tz": "UTC",
  "label": "User,Logoff",
  "event_category": "Logoff",
  "message": "An account was logged off",
  "logon_type": "3",
  "norm_id": "WinServer2008",
  "collected_at": "DKCNTLP001",
  "_identifier": "4",
  "device_ip": "172.24.85.21",
  "event_id": "4634",
  "_fromV550": "t",
  "_enrich_policy": "ueba_enrich_policy",
  "domain": "DISAGROUP",
  "_type_num": "col_ts severity facility log_ts event_id
     logon_type sig_id _offset _identifier",
  "event_log": "Security",
  "_type_ip": "device_ip",
  "sig_id": "6",
  "col_type": "syslog",
  "user": "NORUMOSLT0250F\$",
  "facility": "1",
  "object": "account"
}
```

Listing 3.1: An exmaple log message in JSON format

3.4 Classifier architecture

There are two distinct classes of alerts. The first lies in the behavior pattern of the user which we will try to classify as a sequence because we want to preserve the sequential order of the events. The second classifies the user based on an abnormal quantity of different events.

3.4.1 Sequence classification

The essence of the first alert type lies in the sequential nature of the data. The alert signifies some behavior pattern that an entity presents. The underlying reason for that might be for example a malware breach. There are multiple ways sequence classification is handled. Another problem this solution contains is that we have to handle variable length input because the sequences need not be of the same length.

Anomaly detection. One the literature calls *Anomaly detection* and is more focused towards finding a noticeable change in the behavior. The reason is that the patterns of entity behavior in a system are mostly stable. This is due to the recurrence of tasks.

An example could be a receptionist. The person would probably use collaborative services (such as email and shared calendar) and log on in the morning and leave in the evening. When a breach would occur on such entity, the system would notice that the entity is using unusual services, e.g., accessing shared files, or manifesting an unusual behavior —trying to log in on multiple clients at the same time. Such behavior would exhibit abnormal usage pattern and would then be flagged.

Sequence distance. The other approach —*Sequence distance* (described more in-depth in Section 2.3.2) leverages the order of the events and the patterns that occur because of that. It does not compare the patterns of recent behavior to the current one but instead tries to find how closely a given sequence resembles labeled patterns.

The rationale is that we are trying to flag behavior that resembles the pattern. When a PC is infected with malware, we can monitor its behavior. With the data obtained by monitoring the behavior, we can then classify not just an anomaly in behavior but also what is the cause of it. This approach misses a new kind of attack if the attacker changes the behavior pattern; however, this method is resistant towards random behavior changes that are not caused by an external influence.

3.4.2 Feature vector classification

For the second kind of alerts, we focus more towards the composition of the sequence than the individual patterns. The predictors for these alerts could be called threshold classifiers because usually there will be some limit (or generally a bound space) of what we consider not to be an anomaly. Because of this we do not take the sequence as individual elements but try to represent it in a fixed length input.

An example could be that usually, a person logs on and out a few times a day. A worm would want to spread through the network so it would try to access all the resources (in a simplified view). This would lead to an abnormal count of for example Access denied events.

Because we handled the variable length of the sequence and the input is thus fixed in size, we can use standard machine learning models such as SVM or Decision trees.

Chapter 4

Proposed solution

In this chapter, we focus on the implemented solution to the problem from Chapter 3. Firstly we describe the workflow of the overall solution. Then we focus on the design of the classifiers. In the last section, we describe how we implemented it concerning the technologies used and implementation details.

4.1 Data handling

There are two kinds of information we can draw from data.

One that is useful for the data analyst to see basic patterns and extract information based on it. This can be used to tailor the architecture of the model used for the problem or pre-process the data to fit the model only on the complex patterns while also cleaning the input for the model. An example could be that we only try to predict some classes of alerts on logs generated only by a single user while other patterns on services' logs.

The second kind of information in the data is everything else. Whenever we see a field that we do not draw some immediate conclusions as how to use it, it can have some informational value to the classification. The best examples for this are the artificial neural networks. For the famous machine learning algorithm AdaBoost [5] one had to create the small classifiers (or at least specify the classes of such) to use the algorithm. Neural networks do not need any set of possible small classifiers because they emerge inside of the network during the learning process and can be very complicated such that it is hard to draw any information from the model (compared to for example decision trees). This also means that the usefulness of such information depends on the model we choose.

Based on this we can design the general algorithm of finding an incidental behavior. In Section 3.1 we mentioned that the logs are collected from the whole network and merged into a single stream. However, as mentioned above we can lift some responsibility from the classifier by picking out only the relevant logs. We can see from the data that we classify only behavior for a single user. This means we can run the analysis for each user separately because the evaluation is independent of other users' behavior.

Algorithm 1: General algortihm

```
Input: Logs - List of sequences to be classified
Input: $stepSize$ - The size of the step from interval $(0, 1]$
Output: Alerts - Label of the sequence $s$
Data: Windows - List of sliding window sizes to classify with

for $(user, logs) \in$ Logs.$groupByUser$ do
    for $windowSize \in$ Windows $\leq logs.size$ do
        $stepLength \leftarrow windowSize * stepSize$
        for $i \leftarrow 0; i < logs.size, i \leftarrow i + stepLength$ do
            $class \leftarrow$ classifySequence(logs[$i : (i + windowSize)$])
            if isAlert($class$) then
                Alerts.add($class$)
            end
        end
    end
end
return keepSmallestOfOverlappingIntervals(Alerts)
```

In Algorithm 1 we examine each user independently, and this step can be parallelized well on a per-user basis. For each group of logs, we then classify all sliding windows of desired sizes. We do not evaluate all possible windows (which would be of step 1) but only by given $stepSize$ which is relative to the window size (e.g., $\frac{1}{4}$ of the sliding window size). This can significantly improve the speed of the algorithm but introduces some level of fuzziness as we might not select the smallest possible sequence that would trigger an alert but only the smallest window with some possible noisy logs at the beginning or the end. As a last step of the algorithm, we remove overlapping intervals of the same alert type leaving us with the smallest sequence length of the same alert that could have been otherwise triggered by a sequence that is slightly larger thus containing also the incidental behavior.

4.2 Sequence classification

The first classifier takes the task of classifying the whole sequence. It preserves the structure of the sequence as described in Section 2.3 and uses the nearest-neighbor class of algorithms to classify it with a set threshold for the sequence not containing an incident. The requirement for *Nearest neighbor* algorithms is to define a distance metric between two sequences. We go over the proposed solution with increasing detail on the specific functionalities and dependencies. First, we provide the pseudo algorithms for the general classification algorithm then we describe the sequence distance metric used and then finally with the event distance.

4.2.1 General algorithm

Because we use *Nearest neighbor* class of algorithms, we have to calculate the distance between the current element we are trying to classify and the elements of our training set. A slightly modified version of *Nearest neighbor* algorithm in Algortihm 2.

Algorithm 2: Sequence classifier

```
Data: w- Weights for distance between different event types
Data: T- Set of training samples
Data: $T_k$ - Subset of sequences of class k
Input: $s$ - Sequence to classify
Output: $l$ - Label of the sequence $s$
$l \leftarrow$ no alert
$nnCriterion \leftarrow \inf$
for $i \in \{k, T_k \neq \emptyset\}$ do
    distances $\leftarrow [0...0]$
    for $t \in T_i$ do
        distances$[k] \leftarrow$ sequenceDistance($s$,$t$,w)
    end
    if $c \leftarrow$improvesNNCriterion ($nnCriterion$,distances,$i$) then
        $l \leftarrow i$
        $nnCriterion \leftarrow c$
    end
end
```

The algorithm is initialized as the sequence s not being labeled with any alert. We then iterate over all classes of alerts. For each alert class we calculate the distances from the current sequence s, and then we evaluate the class' $nnCriterion$ on all distances together. This is to incorporate two things. First, it allows us to change the *Nearest neighbor* strategies such as *k-NN* or *minimal average distance to a class*. Secondly, it allows us to incorporate thresholding the criterion. We can decide for each class how close we want the sample to be to classify it as belonging to that class. It allows us to not label a sequence as label l just because it is closest even when the distance is not sufficient and thus lowering *false positives*.

4.2.2 Sequence distance

In Algortihm 2 we used a distance metric $sequenceDistance$ to calculate the distance of two sequences. It took three arguments (s, t, w) where s is the sequence we are classifying, t is a sequence from the training set, and w is the matrix of weights for distances between different event types. Algorithm 3 provides pseudocode for such distance metric.

Algorithm 3: Sequence distance

```
Input: $s$ - Sequence to align of length $n$
Input: $t$ - Sequence to align to of length $m$
Input: $w$ - Weights for event types
Output: The minimal sequence distance as defined in 2.3.2
DWT$[1 \ldots \mathrm{n},\ 0] \leftarrow \mathrm{inf}$
DWT$[0,\ 1 \ldots \mathrm{m}] \leftarrow \mathrm{inf}$
DWT$[0,\ 0] \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $m$ do
        $cost \leftarrow$ eventDistance$(s[i],\ t[j])$
        if $t[j]$.eventType $\neq s[i]$.eventType then
            $cost \leftarrow (1 + w[t[j]$.eventType$]) * cost$
        end
        $m \leftarrow$ min(DWT$[i-1,\ j]$, DWT$[i,\ j-1]$, DWT$[i-1,\ j-1]$)
        DWT$[i, j] \leftarrow cost + m$
    end
end
return DWT $[\mathrm{n}, \mathrm{m}]$
```

When computing the sequence distance, we use a modified version of DWT (2.3.2). By using DWT, we are not only trying to align the two sequences but to map each element $(t_i; e_i) \in s$ to an element $(t_k; e_k) \in t$ such that we minimize the sum of distances of events in a mapping

$$|(t_i; e_i) \rightarrow (t_k; e_k)| = d(e_i, e_k)$$

over all mappings while not *crossing* the mappings (formally below) and depicted in Figure 4.1. We can see that the mapping (depicted by the lines between elements) never cross. Further the mapping will be referred with a pair of indexes $\{(t_i; e_i) \rightarrow (t_j; e_j)\} = m(i, j)$ Having two mappings:

$m(i,k)$, $m(j,l)$ Following condition holds true:

$$\forall i,j,k,l : t_j \leq t_i \implies t_l \leq t_k$$

Additionally, we extend the distance metric by a *weight*. The cost depends on the *event_type* of the event $(t_k; e_k) \in t$ we are mapping to. The motivation for this approach is that the sequences are cluttered with unimportant events that aren't valuable to the sequence classification. The events that are informatively valuable to the classification will have a high coefficient thus miss-aligning it with an event of a different type will have a big cost. On the other side events that don't provide much information will have a low coefficient, so we miss-align them to align the important events better. Visually represented in Figure 4.1.

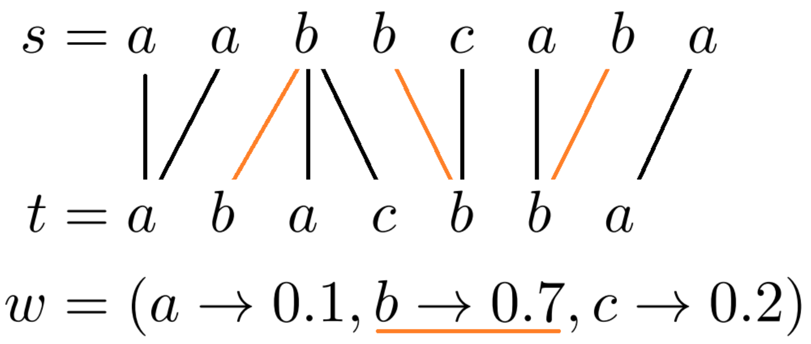
$s = a \; a \; b \; b \; c \; a \; b \; a$
$t = a \; b \; a \; c \; b \; b \; a$
$w = (a \rightarrow 0.1, \underline{b \rightarrow 0.7}, c \rightarrow 0.2)$

Figure 4.1: Alignment of sequence s to sequence t. As event of type b has higher coefficient we prefer aligning b, to miss-aligning two elements. Additionally the picture showcases how we want mappings to only link consecutively and never cross.

4.3 Feature vector classification

Classifying sequences rises the problem of classifying input of various lengths. One of the options to handle that is to create representation and transformation of the input such that the result is of constant lengths (among other approaches described in Section 2.3.1). This is also done in, for example, HashMaps (or dictionaries in some languages) where we hash the key such that we convert it into a constant dimension space. Hashing, however, has the property of having a small change in input results in a significant change in output which is undesirable as it skews the space erasing the essential similarities. In this section, we first describe how we transform sequences to a constant length feature vector then give the description of the architecture of our classifier and finally note how we chose to deal with problems risen in Section 3.

4.3.1 Vector composition

Feature vector elements. The elements of a feature vector are usually constructed using aggregate functions such as min, max, sum, $count$, etc. (and their equivalents for other data types) of the fields that are present in the original events. Other than simply aggregating fields we can also select which fields to preserve. By this, we can remove fields to clean the data (e.g., a description that is the same for each event type). Also, removing fields that are rarely similar can improve the classification (such as timestamps which we use only for the ordering of the events). We construct the feature vector from the sequence using the *event_id* field as we believe this field contains the most information about the sequence concerning the alerts. We experimented with additional features and present the results in Section 5.5.

Feature vector construction. To further deal with the variety in length of the sequences we use the *bag of words* model. We can approach this in two ways. First, because we know that the *event_id* is a four digit number we have the bounds for all the values $v \in [1000; 9999]$. We can then construct appropriately sized vector such that we have a dimension for the count of each possible integer in that range. This approach is more future proof; however, it suffers from the course of dimensionality [2].

The second approach (that we selected) first creates an ordered set E of all possible values of *event_id*s in the dataset. The feature vector is then constructed such that the i-th dimension of the vector v corresponds to the i-th smallest event id in the set. Having a sequence S and an event_id set E

$$S = 1, 5, 5, 2, 2, 1, 1$$

$$E = 1, 2, 5$$

We represent the sequence S as a vector $v = (3, 2, 2)$. This approach solves the problem of how to represent a variable lengths input, but we lose the information that was contained in the ordering of the events in the original sequence as explained in more detail in Section 2.3.1.

4.3.2 Classifier design

Now we want to describe two different approaches to the classifier design. The first is a straightforward application of a conventional classifier to the problem, or in an extended way for multi-class classification. The second approach splits the labeling of a sequence into two steps —firstly decide whether a sequence is incidental or not and if yes decide what kind of alert we should flag it in the second step.

Conventional classifiers. Below we present two different approaches to classification. Both of them depend on the use of conventional classifiers for the actual labeling but handle the process of classification differently. For the underlying classifiers, we have selected SVMs, Decision trees, and Random Forests.

*SVM*s try to find a separating hyperplane to separate the samples of different classes. An extension of soft margins is used when non linearly separable case is encountered. Additionally, several kernel methods are used as well to deal with linear nonseparability of the dataset. Because SVMs do not provide by default a way for multiclass classification, we use a "one-against-one" model to classify different types of alerts. We chose SVM because of its relative accuracy and simplicity, and past widespread use which made it a baseline classifier.

The second class of classifiers is *Decision trees*. Decision trees divide the space into multiple subspaces in an axis-aligned fashion which fits the presumption of thresholding values in feature vectors. Decision trees are naturally multi-class classifiers and thus well suited for this task.

The last kind of classifiers is a Random forest classifier. Random forest train multiple decision trees on random subsets of training dataset and then select the result based on majority vote. This improves the performance because multiple different trees are trained and also increases the generalization ability.

Direct multi-class classifier. As already mentioned —the first classifier is straightforward. We do not select the classifier concerning the between-class imbalances and only focus on the problem of classification. This means we apply the classifiers mentioned above to the problem directly.

Two-layer classification. The second classifier architecture consists of a two-layer classification. The first layer has the task of classifying whether a sequence is incidental or not. The second layer is applied only when the sequence is considered incidental and outputs its alert type. Architecture depicted in Figure 4.2.

We create an additional dataset by transforming the original one into a $y \in \{0, 1\}$ classification problem (we merge all alerts into a single type). This gives us more samples of the minority class (now a single minority class) which improves the between-class imbalance against the majority slightly as well as increases the number of samples we can synthesize data from (using *SMOTE* algorithm etc. more in-depth in section 2.1).

There are several benefits to this approach. The first is that based on the experimental results we can output two kinds of certainty —how

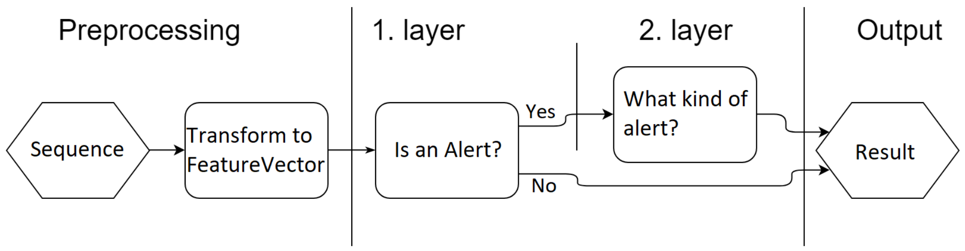

Figure 4.2: The architecture of the two-layer classification. First, we convert the sequence to a feature vector. We then classify the vector as either incidental or normal behavior. Then if the vector is classified as incidental we return the type of alert we believe the vector corresponds to or return that the sequence is clean.

successful the classifier is that a sequence is an alert and how reliable is the prediction of the type of the alert.

Another benefit is that we can separate the problems. Let's now focus on the problems we separated and what it allows us to do.

1. Because the classification of incidental behavior is only a binary problem, we can use some more standard solutions. For example for SVM, this improves the speed of the classifier as multiclass SVM uses the "one-against-all" approach.

2. The imbalances of the alert types are now relevant. We no longer need to balance the alert types relative sample size as the a priori probability of each alert type contains information about the respective occurrences.

4.3.3 Class imbalance

There are some similarities when dealing with the between-class imbalance problem for both of the approaches. As mentioned in Section 2.1 we can use sampling methods that alter the samples of the dataset or cost methods where we alter the loss on a per-class basis.

Cost methods. Because we chose to use *SVM*, *Decision trees* and *Random forests* as our predictors the use of cost method is very straightforward. All methods allow us to incorporate a different penalty for misclassifying a sequence. For the *direct classifier* we can balance the weights such that the sum of weights for alerts equalizes the difference in count against clean training samples. For the *two-layer classifier* we balance only the first $(0/1)$ layer because as mentioned above there isn't any unusual level of imbalance between the different alert types thus it contains some informational value of the a priori probability of each alert type.

Sampling methods. As to using sampling methods on the dataset, we will combine under and oversampling methods (described more in-depth in Section 2.1.2). Because the majority class is in such high imbalance, we randomly sample the majority class to reduce the total size of the dataset as we can safely assume that the general clean patterns will occur multiple times. We can do this for both the direct and the two-layer classifier.

For the sampling of the minority classes, we use the combination of SMOTE to generate new data, and Tomek links to clean the noise and border elements (explained in Section 2.1.2). Tomek links should improve the classification for both decision trees and SVM.

For the two-layer classifier one thing should be noted —the sampling should happen *before* the transformation of the dataset to a $0/1$ problem as using SMOTE between vectors of different alert types could yield incorrect results due to the possibility of a "clean" valley between them.

4.4 Implementation

In this section, we talk about the implementation details of the solution stated above. Firstly we talk about the technologies used together with libraries. Then we provide an overview of the pipeline how the data is handled, what is the format of the training dataset and how the classifier is designed.

4.4.1 Technologies used

As our language of choice, we selected Python. It was a very straightforward choice. It is one of the most widely used platforms for data science/machine learning and a go-to platform for many other fields. Python is a scripting object-oriented language described as

> Python is a simple, yet powerful, interpreted programming language that bridges the gap between C and shell programming, and is thus ideally suited for "throw-away programming" and rapid prototyping. [17]

Hand in hand of the widespread use of Python the surrounding ecosystem has grown as well. Altogether this leads to vast amounts of libraries that are implemented for Python but usually just having a Python wrapper for a C library.

Although described as “suited for ’throw-away programming’ and rapid prototyping”, it has also become widely used production platform nevertheless with some controversy surrounding that choice. Python as an interpreted language is very slow (we refer the reader to “The Computer Language Benchmarks Game” for comparison) compared to a compiled language and its strength lies in the integration of compiled libraries. Another disadvantage is that it is dynamically typed language which introduces some problems when the scale of the project overgrows the original intentions. However, this has been addressed in `Python 3.5` where it has been made possible to use static types.

Development. The development has been done in *Jupyter* [11] using *IPython* notebooks [15]. Together they form a great environment for fast prototyping of Python applications.

Each notebook contains independent cells of either runnable code or a Markdown to document the notebook. The cells can be run independently of the order they are placed in the file and on demand. Each notebook runs in its own kernel having separate state from other notebooks. The state is persisted across cell runs and thus allows to modify parts of the code without re-running the whole program. Additionally, each cell contains an output cell which can be persisted in the saved notebook file.

The only disadvantage of said notebooks is that the files are not saved as regular Python code files but in a JSON format and thus aren’t easily readable or runnable without the IPython technology which would also introduce some additional work when porting the project to a standalone application. This, however, doesn’t pose a problem as the next step in the development would be to largely scale the application to a distributed computational framework *Spark* which is JVM based.

4.4.2 Training set format

The training data was provided in a DataFrame object from the *pandas* framework. DataFrame could be described as a wide-column, tabular data structure. Columns of the DataFrame were the field names of the JSONs (described in Section 3.3) and each log from the original dataset corresponds to a single row of the DataFrame.

Additionally, we have columns for each pattern we have labeled. Each labeled sequence is marked with a unique integer in the appropriate column or a zero if the log does not belong to that sequence. This means that if we have a row with the number 2 in the column *pattern_4* it belongs to the labeled sequence 4 with the label of *pattern_4*. If we want to extract the whole sequence we take all the rows of the DataFrame which have the corresponding number in the respective column. If we want to take a sequence that doesn't belong to any pattern we need to check that we either have at least two unique numbers or only the number 0 in each pattern column.

4.4.3 Pipeline

The pipeline consists of 4 major steps. First, we extract the labeled sequences from the tabular format into groups. Then we extend the set of labeled sequences by non labeled sequences and we assign them the label *pattern_0*. Afterward, the sequences are transformed into the vectorized form. In the last stage, we split the dataset for training and validation and the classifier is trained and evaluated.

1. We iterate over the *pattern_X* columns and create a group for each one of them. For each column, we group by the column's value forming the labeled sequences and discarding the logs under the group 0 as it signifies that the logs do not belong into any sequence of that label.

   After this step, we have an object containing a group for each pattern type each containing a list of sequences of logs. We do not need the original values of pattern columns further as the purpose was only to differentiate the different occurrences of a single pattern.

2. We select the desired count of sequences to fill the group of negative examples (we take the negative group as sequences with no alerts associated and positive as sequences with any alert associated). We select some sequences of arbitrary lengths and for each, we check that each pattern column has either only zeros or at least two different integers.

   After this step, we have the dataset available for training of both positive and negative samples.

3. With the sequences grouped by the label, we then vectorize each of them. The vector is constructed in a way that each dimension represents the count of one specific event_id. To be able to do that we first have to collect all the unique event_ids. We have to collect them from the original dataset from step 1 to be sure that we do not encounter by a small chance a missing event_id that wasn't in the *selected* training sample from step 2.

   After this step, we have a set of vectors each representing one sequence grouped by the label of each sequence.

4. As the last step we first unzip the vectors from grouped representation into a `(X, y)` pair where `X` is the sequence of all the vectors and `y` is the sequence of corresponding labels. Further, it is then split into a training and validation set to evaluate the performance of the classifier on data that were not used for training. We then train the classifier using the convention from the *SciKit* API [3] of training the classifier using the `fit(X, y)` method with the training part of the data.

   After this step, we have a trained classifier, and we can evaluate its performance on the validation set.

4.4.4 Classifier architecture

As noted above we have two kinds of classifiers —one we called *Direct* classifier which classifies either no alert or alert type directly, and the second —*two-layer* classifier where we use one classifier for predicting alert and then a second layer for the alert type.

From this decision we have two base classifier classes —`DirectClassifier` and `TwoLayerClassifier` which define the API for the classifiers. It is inspired by the *SciKit* API [3] where a `clf.fit(X, y)` method is used to train the classifier and `clf.predict(X)` is used to classify samples. Both of them are used as wrappers for the underlying classifiers such as *SVM*.

Both of the classes contain a skeleton implementation that offers subclasses to step in during the training process. The method is called on the input `(X, y)` parameters before the training of the classifier happens. The method is then later overridden by a `BalancedDirectClassifier` and `BalancedClassifier` to provide the balancing of the dataset using the combination of *SMOTE* oversampling and *Tomek links* undersampling. From the two `Balanced*` classifier classes we then have subclasses with different parameters and classifiers used (e.g. `TreeDirectClassifier`).

The main difference between `DirectClassifier` and its subclasses compared to the `TwoLayerClassifier` and its subclasses are that the latter provides two different sets of the “step-in” methods described above. One for the incident classifier and the second for the alert type classifier —as they both need different handling (e.g., for sampling).

4.4.5 Libraries

The main library used was `scikit-learn` [14]. Scikit-learn is an open-sourced machine learning library that contains the implementations of most of the machine learning techniques and models used. In this thesis, it has been mainly used for the implementation of classifiers SVM, Decision trees and Random forests and its utility methods.

As for the implementations of techniques used for handling dataset imbalances we used the `imbalanced-learn` [12] library. It uses the Scikit-learn [3] API and implements numerous sampling methods. In this thesis the implementations of oversampling using *SMOTE* and undersampling using *Tomek links* have been used.

Additionally, we used several data structure libraries such as `Numpy` and `Pandas`. The performance graphs in the Experiments section were generated using `Matplotlib` [9].

Chapter 5

Experiments

In the Experiments chapter, we evaluate the solution which implementation was described in Chapter 4. First, we talk about the Sequence classification —what was the performance, and what we believe are the reasons why the method failed. In the rest of this chapter, we then talk about the Feature vector classification. We define the metrics that were used to compare the performance of the different classifiers. Then, we describe the workflow of the experiments and the implementation details. In Section 5.4 comes the actual evaluation with graphs of classifiers' performance. Lastly we talk about the selected features as referred to in Section 4.3.1

5.1 Sequence-based classification

The sequence classification proposed in Section 4.2 didn't work. We did not finish the implementation of the sequence classifier because the early stages of development were exhibiting inferior performance.

We used the FastDTW [18] implementation along with the regular DTW implementation from the same library. Although the methods differed slightly in the results, the difference was not relevant for the performance.

In Figure 5.1 we can see that the presumption that sequences of the same type would be close to each other fails. The reason the method failed on this dataset is that when inspecting the distances between the sequences we usually see numbers most frequently in the range $[0, 1, 2]$. This means that the method cannot differentiate between sequences of a different type. We believe this to be caused by the insufficient complexity of the sequences.

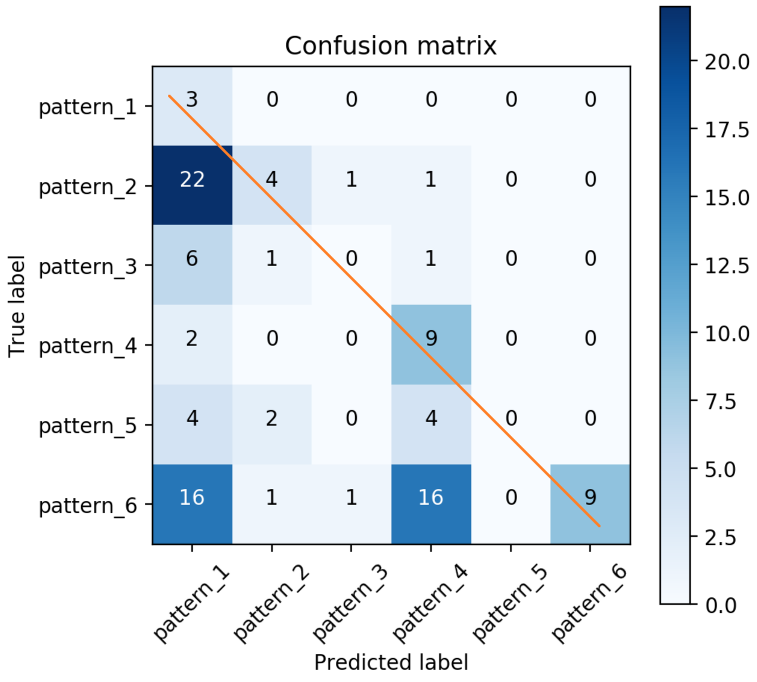

Figure 5.1: Confusion matrix for the sequence distances using the *FastDTW* method. The diagonal is highlighted so we can see that only pattern_1 and pattern_4 are closer to each other than the rest of the patterns.

Further, all sections are concerned with the feature vector classification.

5.2 Experiment metrics

As noted and described more in-depth in Section 2.1.1 we cannot use simple error rate to evaluate the performance of a classifier on an imbalanced dataset. The higher the imbalance for a class the more inclined the classifier would be to blindly overfit to the majority class as it would minimize the error.

We chose to evaluate the performance of the classifiers using ROC (Receiver operating characteristic) with *False positive* and *True positive* rates on x and y axis respectively. Because we did not use any soft classifiers (those that provide the probability of each class instead of only a label), we cannot generate ROC curves and generate only ROC points which will ease the comparison a little.

Furthermore, as the alert classification is not a binary classification problem we collect a per-class error rate to compare the classifiers. However, having two metrics to compare two predicts introduces the problem of which classifier performs better when each has better metrics in one category. We can assume that in production we value more not missing an alert than the lower error rate on the specific alert type. Also, we can compute some other statistics (such as *precision*) and order the classifiers based on that.

5.3 Pipeline and implementation

We implemented the following experimental pipeline. To ease the evaluation we refrained from using the pipeline as outlined in Section 4.4.3 which is oriented more towards regular usage than the batch evaluation on multiple datasets.

1. We create the dataset to run the tests on. We do that by taking the full set of labeled alerts grouped by the alert type and add the *pattern_0* group with c sequences of no pattern. With this, we modify the imbalance ratio of the alert classes to the no-alert class.
2. We convert all sequences to the vector representation.

   After this step, we have the prepared dataset for the evaluation.
3. We create n training/validation split pairs from the current training dataset. We do this step commonly for all the classifiers to evaluate the different classifiers on the same datasets to minimize the number of external variables.

   Also, we split the data in a stratified fashion such that all classes are present with the desired ratio in both training and validation sets.
4. We train and evaluate each classifier on all the training and validation pairs.
5. We calculate the statistics of the performance of the classifier on the validation sets.
6. As the last step, we aggregate the results and generate graphs for each classifier for both aggregated and raw data.

We evaluate the above pipeline on multiple different imbalance ratios with $c \in \{100, 200, 1000, 10000\}$ and the number of tests $n \in \{10, 100\}$.

5.3.1 Classifiers

The following *Direct* classifiers were used:

SVM, C=1 An SVM classifier with the $C = 1$ parameter. The C parameter is used with soft margin classification (in case of non separable data) and marks the penalty multiplier for misclassification. We used the `sklearn.svm.SVC` class from Scikit-learn library.

SVM, C=10 Same as above the class `sklearn.svm.SVC` but with $C = 10$ which means higher penalty for misclassification.

Decision tree We used the `sklearn.tree.DecisionTreeClassifier` class with the *gini* criterion.

Random forest, n = 5 Random forest is an ensemble classifier which uses several Decision trees and then uses the majority rule to select the classification. We used the `sklearn.ensemble.RandomForestClassifier` class with $n = 5$ being the number of underlying decision trees.

Random forest, n = 10 Same as above but with $n = 10$ number of underlying decision trees.

Along with three variants of each classifier class. The first was just the classifier with the parameters described above and the dataset was not resampled. In the second we resampled the dataset using SMOTE and Tomek links. The third version was with the non-resampled dataset but the costs were balanced inversely to the respective classes' frequencies (as described in Section 2.1.3).

> The ”balance” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as $n\_samples/(n\_classes * np.bincount(y))$.
> Taken from the Scikit documentation for respective classifiers.

And the following *Two-layer* classifiers were used:

SVM, C=1 Two `sklearn.svm.SVC` classifiers in series with the $C = 1$ parameter.

SVM, C=10 Same as above except $C = 10$.

Decision tree Two `sklearn.tree.DecisionTreeClassifier` in series with the *gini* criterion.

Random forest, n = 5
Two `sklearn.ensemble.RandomForestClassifier` in series with $n = 5$ underlying decision trees.

Random forest, n = 10 Same as above with $n = 10$ trees.

Additionally, as with the direct classifiers, there were two different instances of each of the two-layer classifiers from the list above. One where the dataset was left intact and the second where we resampled it.

The range of c values. We believe that values of c greater than 10000 would introduce undesired attributes to the dataset. The size of the original dataset is ~ 700000 logs. Because we sample sequences with the length $l \in [5, 100]$, we select on average ~ 525000 logs in sequences. Increasing the c further could lead to higher probability of duplicate sequences and would thus skew the evaluation as two sequences that differ in very few elements could be split such that one ends up in the training and one in the evaluation set.

5.4 Evaluation

In this section, we provide the actual results as well as the evaluation of the performance of the classifiers using the pipeline described above. We will talk about the average performance of each type of the classifier as well as the precision/variation on the performance.

5.4.1 Incidentiality evaluation

For now, we will ignore the idea of multiple alert types and focus only on the binary classification whether a given sequence should be classified as an alert or not. The rationalization is that it is more important to detect an alert than to correctly classify its type because it will be further handled manually by analysts. As noted above we will use the *False positive* and *True positive* rates:

$$TP\_rate = \frac{TP}{P_C}, \quad FP\_rate = \frac{FP}{N_C}$$

with P_C being the total count of positive samples and N_C being the total count of negative samples.

When we plot the results with FP_rate and TP_rate being on x and y axis respectively the classifier that is closer to the $(0, 1)$ point (top left corner) is better. We added concentric circles with the center in the point $(0, 1)$ to the graphs comparing the classifiers to make the performance comparison more clear. Also, it aids in evaluating the classifiers with the highly imbalanced dataset as the axis have a very different scale to such extent that the circles resemble ellipses or straight lines.

Means

We will now provide the averaged results for two different dataset imbalances. The first is balanced with close to a $1:1$ ratio alerts to clean sequences. The second is imbalanced with a $1:100$ ratio of alerts to clean sequences. We would like to remind the reader to pay attention to the scales as the big difference in imbalance ratios makes the axis very skewed.

Dataset with $1:1$ ratio. In the Figure 5.2 we can see the overall performance on the $1:1$ dataset. We can see that there is one larger cluster near the top left corner and then several scattered points.

On separated plots in Figures 5.3 5.4 5.5 we can see that the worst performing is the SVM class of classifiers. Generally, we can see better False positive rate with the forest class of classifier compared to the Tree class classifiers. However, the Tree class of classifiers performs better concerning the True positive rate which we established is more valuable to the solution of this problem along with the points being overall closer to the $(0,1)$ point.

Additionally, we do not see any pattern concerning the balancing of the dataset. This is desirable behavior as the balancing should not be creating data points outside of the hidden patterns and thus shouldn't be lowering the score on non sampled verification dataset.

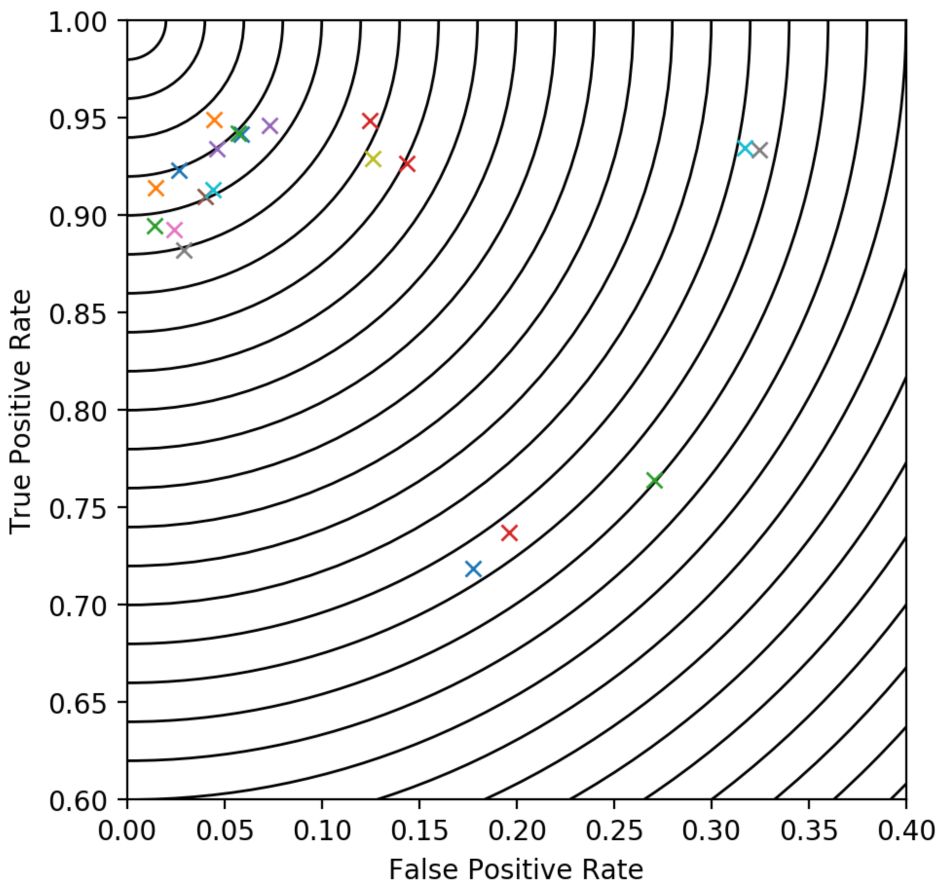

Figure 5.2: The average performance of all the classifiers from Section 5.3.1 for the $1:1$ imbalance ratio dataset. As there aren't enough colors to uniquely plot all the points the image is provided for reference of the overall performance.

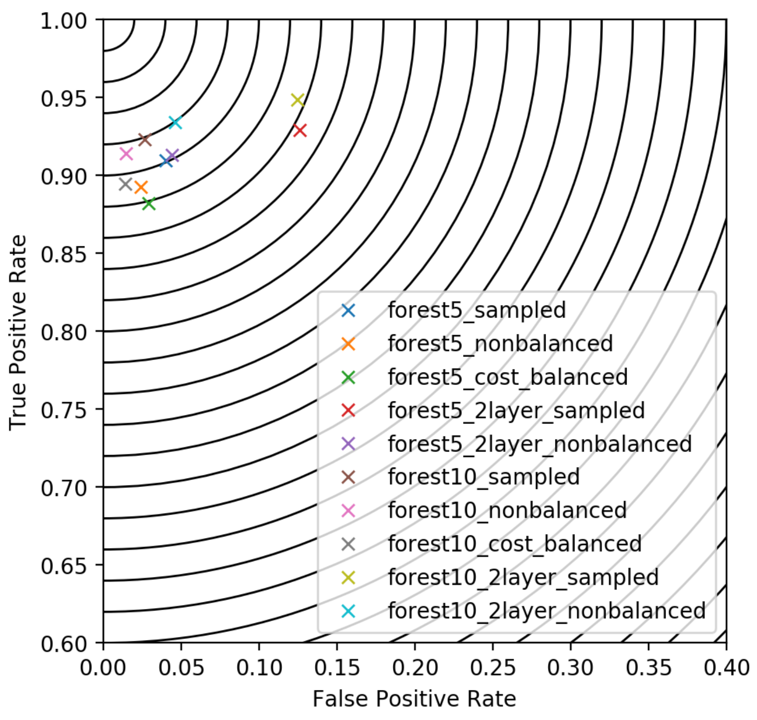

Figure 5.3: An ROC graph for the Forest class of classifiers on the $1:1$ dataset.

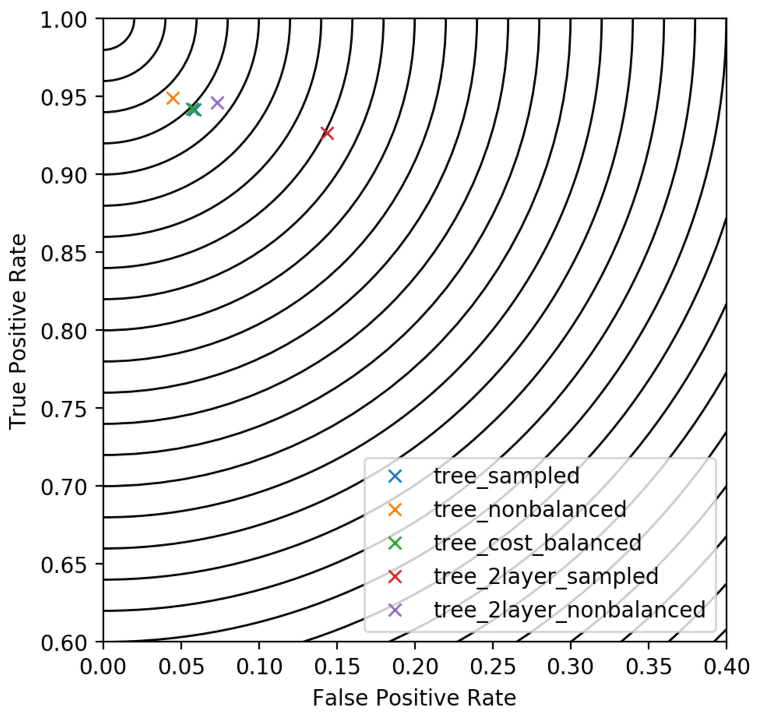

Figure 5.4: An ROC graph for the Tree class of classifiers on the $1:1$ dataset.

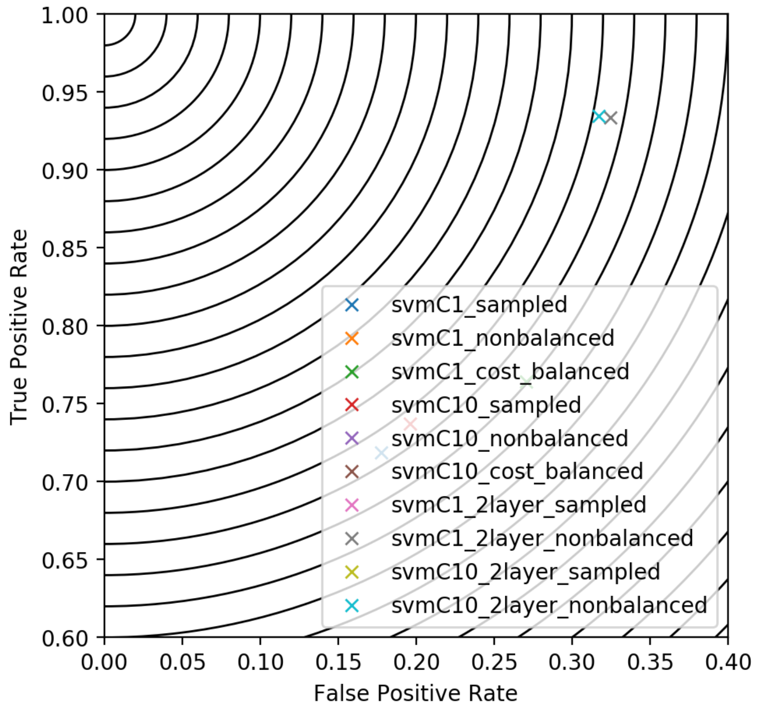

Figure 5.5: An ROC graph for the SVM class of classifiers on the $1:1$ dataset.

Dataset with $1:100$ ratio. In the Figure 5.6 we can see the overall performance on the $1:100$ dataset. The reader should keep in mind the different scale of the two axes. The lines are still concentric circles with the center in the $(0,1)$ point.

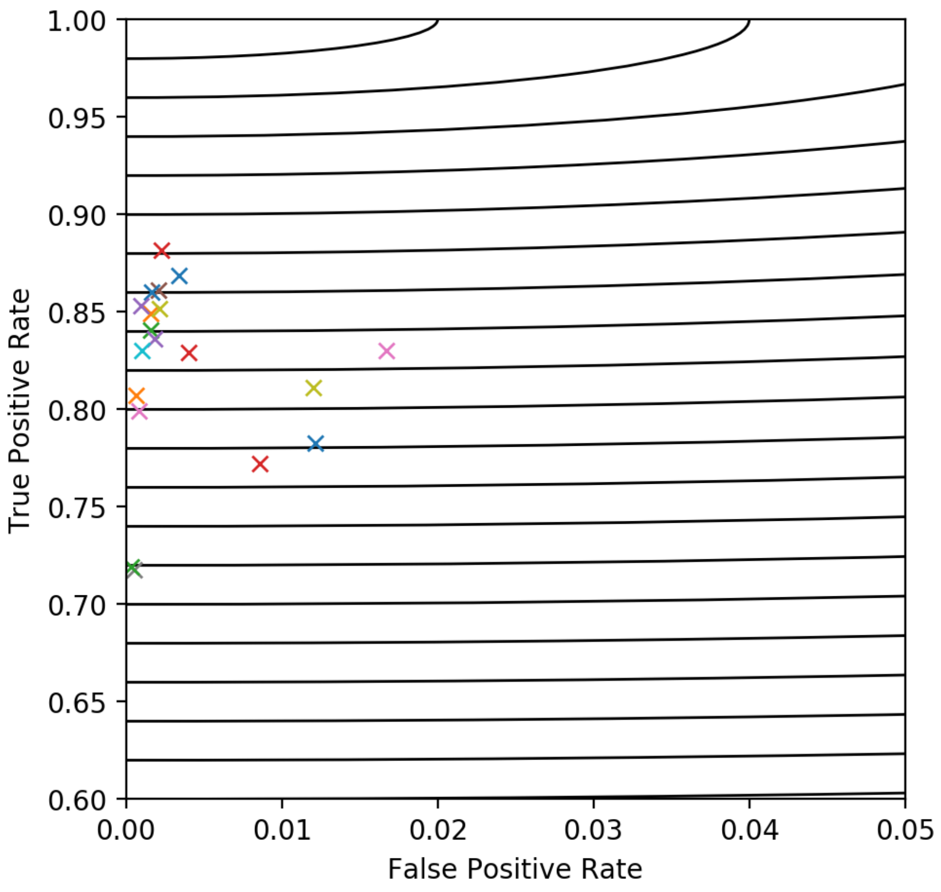

Figure 5.6: The average performance of all the classifiers from Section 5.3.1 for the $1:100$ imbalance ratio dataset. As there aren't enough colors to uniquely plot all the points the image is provided for reference of the overall performance.

At first sight, we notice that the increase in the ratio towards the negative class shifted all the classifiers towards smaller False positive rate values but also smaller True positive rate. This is expectable as the a priori probability of the negative class increases compared to the positive class. The shift to lower True positive rate values is expectable as well as the classifier is more inclined towards negative class.

As with the smaller dataset, we can see the SVM class of classifiers generally performing the worst with several being outside of the plotted area which is, however, considerably smaller this time. Interesting observation that can be seen in Figure 5.9 is that only the *sampled* versions of the

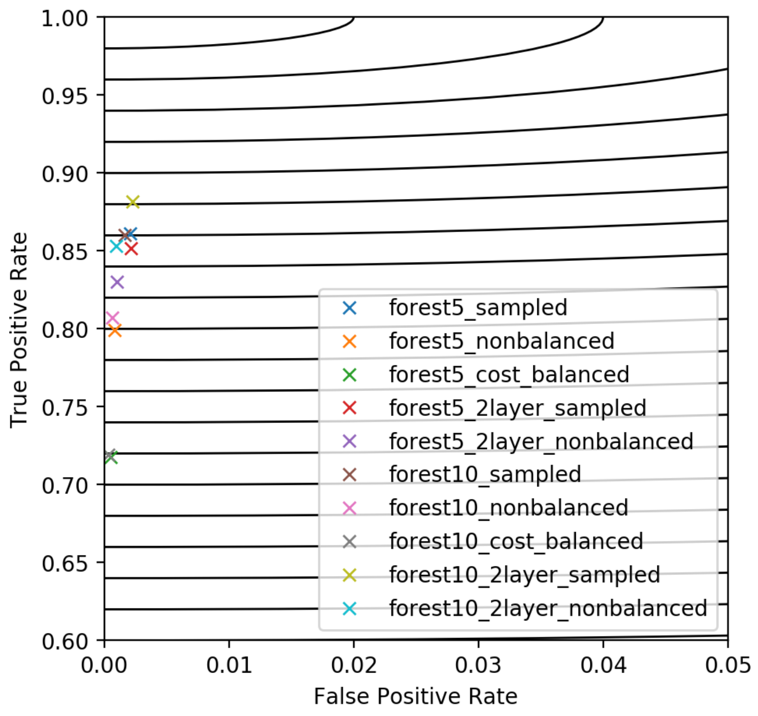

Figure 5.7: An ROC graph for the Forest class of classifiers on the $1 : 100$ dataset.

classifiers are in the frame. Neither original datasets nor cost balanced classifiers performed well enough.

With the forest class of classifiers in Figure 5.7 we can see that the cost balancing method is the worst performing one by a large margin. The best performing classifier overall is the *two layer* classifier with the resampled dataset and generally the resampled dataset was increasing the performance of the classifier architecture.

Tree class classifiers (Figure 5.8) performed most reliably with a small size of the cluster compared to the other two underlying classifier classes.

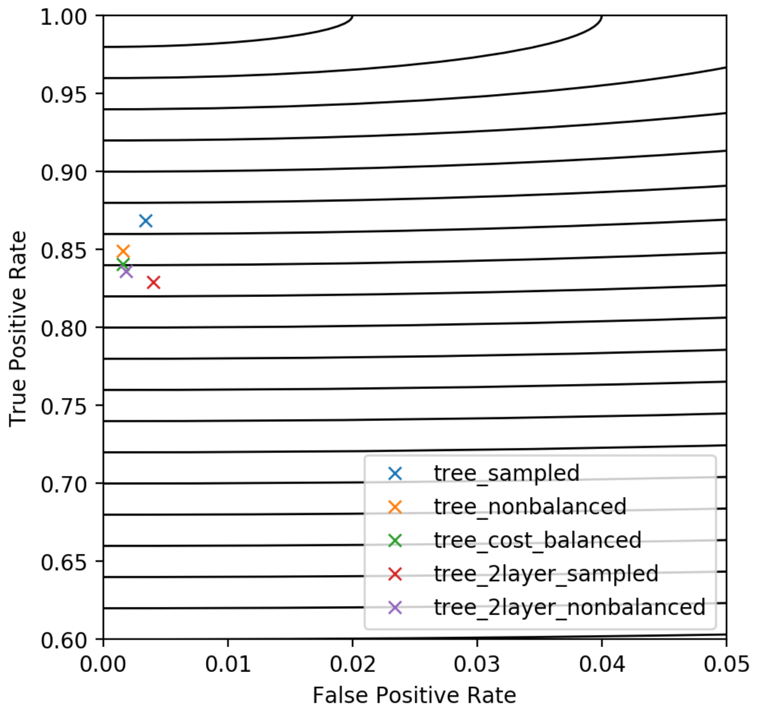

Figure 5.8: An ROC graph for the Tree class of classifiers on the $1:100$ dataset.

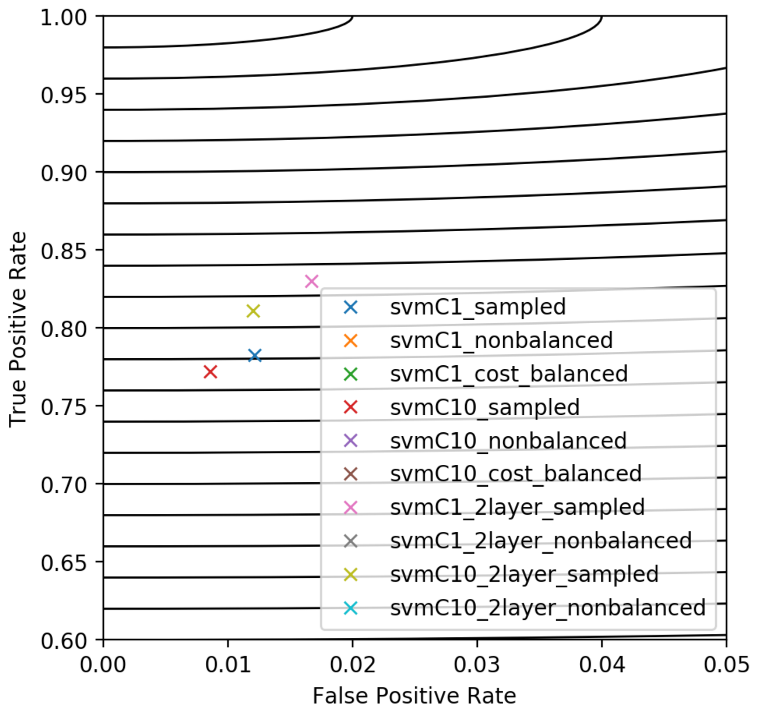

Figure 5.9: An ROC graph for the SVM class of classifiers on the $1:100$ dataset.

Variability

An interesting pattern can be seen regarding the variance of the performance of the classifier with regard to the balancing method used. We can see in Figure 5.11 a Direct decision tree classifier without any balancing technique used. When we compare it to the cluster in Figure 5.12 we can see that sampling the dataset increases the variability of False positive rate but doesn't affect the variability in the y axis. On the other hand using a cost balancing method in Figure 5.10 we can see that it increases the variability in the True positive rate axis.

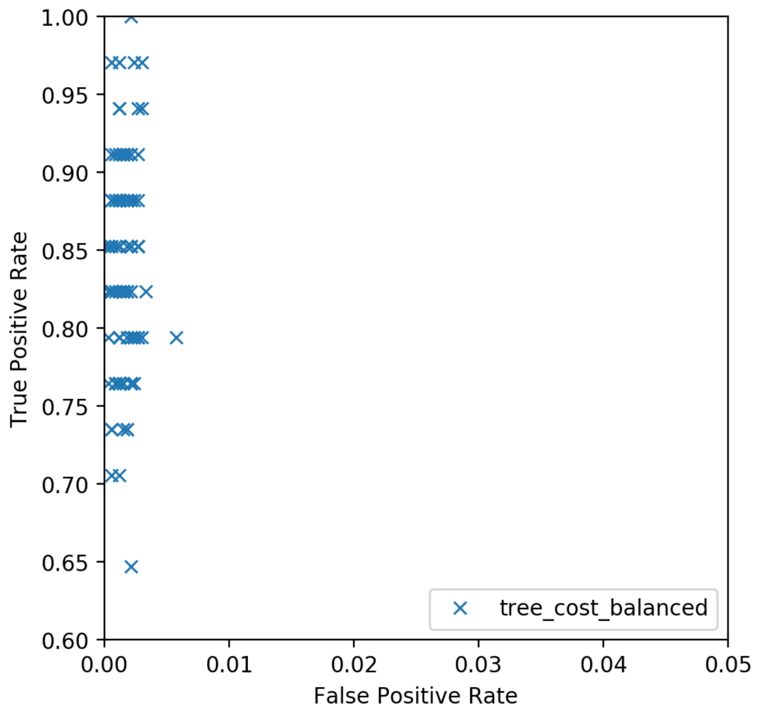

Figure 5.10: An ROC graph for the direct Tree classifier on all the $1 : 100$ datasets with cost balancing.

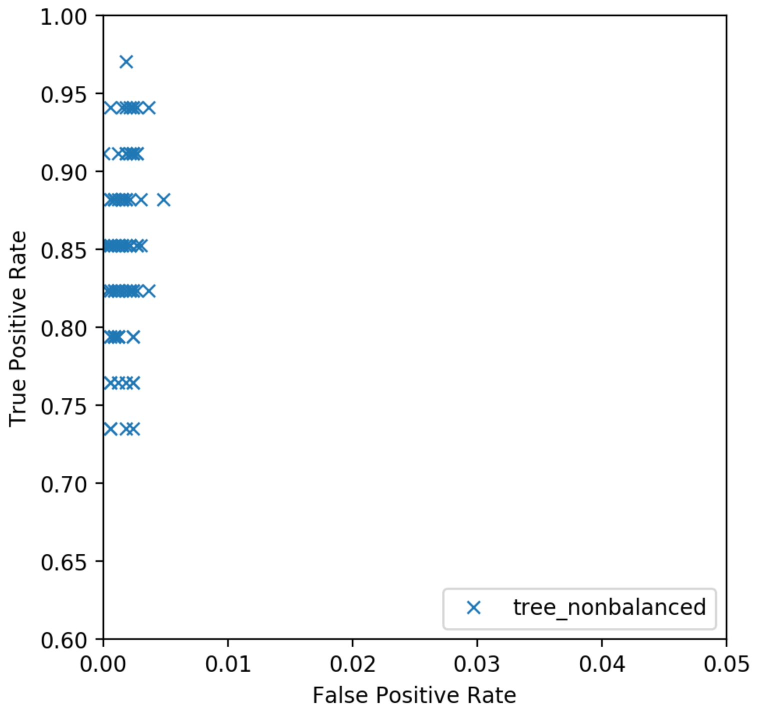

Figure 5.11: An ROC graph for the direct Tree classifier on all the $1:100$ datasets without any balancing.

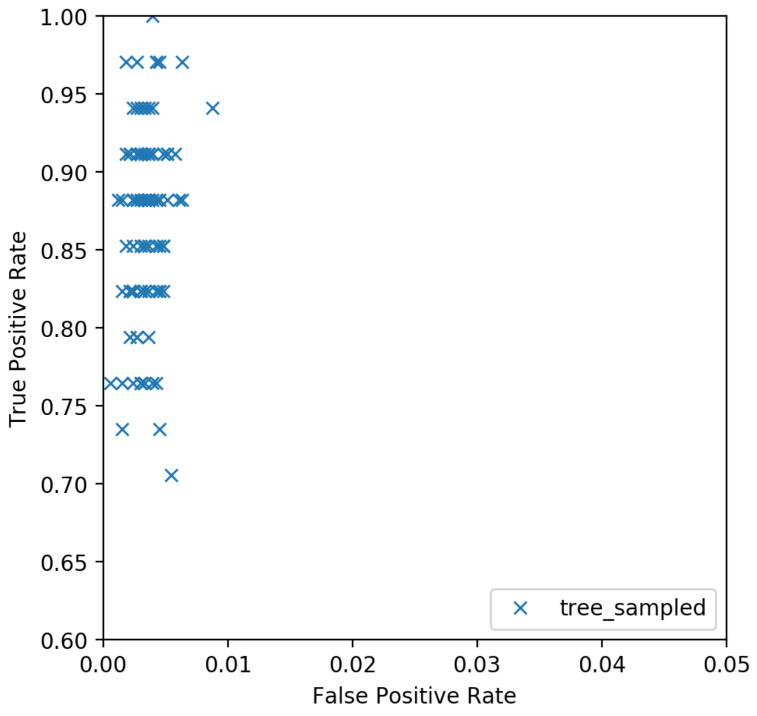

Figure 5.12: An ROC graph for the direct Tree classifier on all the $1:100$ datasets with SMOTE and Tomek links balancing.

5.4.2 Alert type evaluation

Because TP_rate and FP_rate only reflect the ability of the classifier to spot an alert we should also evaluate the error rate on the alert types of the different classifiers. In Figure 5.13 we can see an average weighted error for each classifier with the weights being the respective frequencies.

We can see that resampling the dataset improves the average weighted error in all cases except for the Two-layer decision tree classifier. Additionally, we can see that cost balancing is not very effective. The best performing classifier is the Two-layer forest of size 10 that used the resampling method to deal with the imbalances which also performed the best in the incidentiality detection.

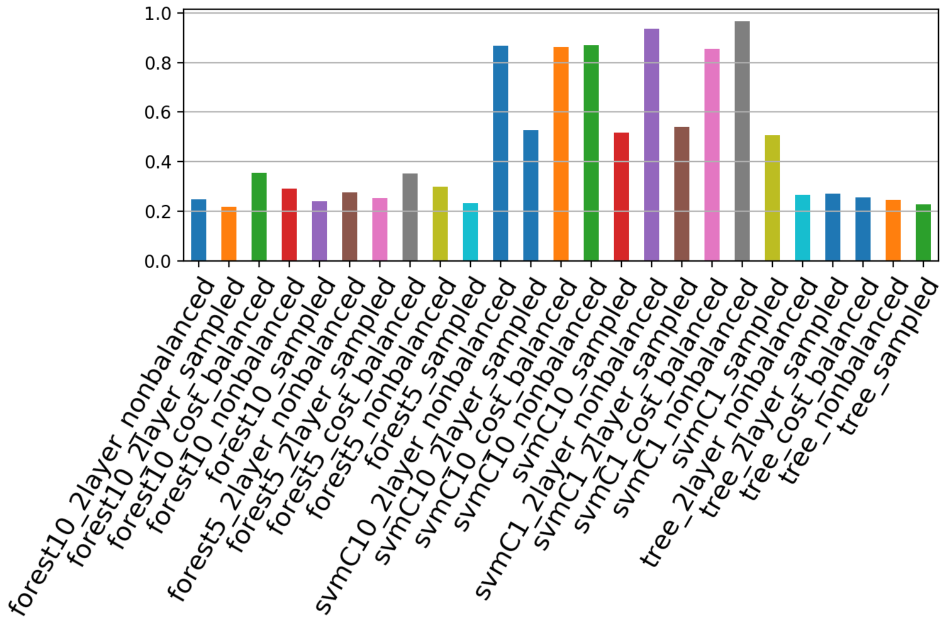

Figure 5.13: A average weighted error of alert type classification on the $1 : 100$ dataset.

5.5 Features selected

In Section 4.3.1 we talked about using the `event_id` field to construct the feature vector. In Figure 5.14 we can see the results of using some additional features. Noticeable is the improvement of both TP_rate and FP_rate. However, the average Alert type errors didn't improve that much.

We believe that introducing these additional features exhibits a case of *Data leakage*. Data leakage is a case when some outside information, that shouldn't be known to the model is (often in an accident) incorporated to the data. The data leakage is introduced by the artificial creation of negative samples. Positive samples naturally emerge, but we generate the negative samples artificially using a randomized algorithm. This leads to the predictors focusing much more on the artificial features because the negative samples are easier to separate by them. If we look at the diagram of a Decision tree (included with this thesis as a *tree.dot* file) learned with these feature vectors, we can see that it depends on the new features in many nodes that separate the negative class.

This easier separation of negative samples leads to improvements in FP_rate and thus also in alert type classification, as we decrease *false negatives*.

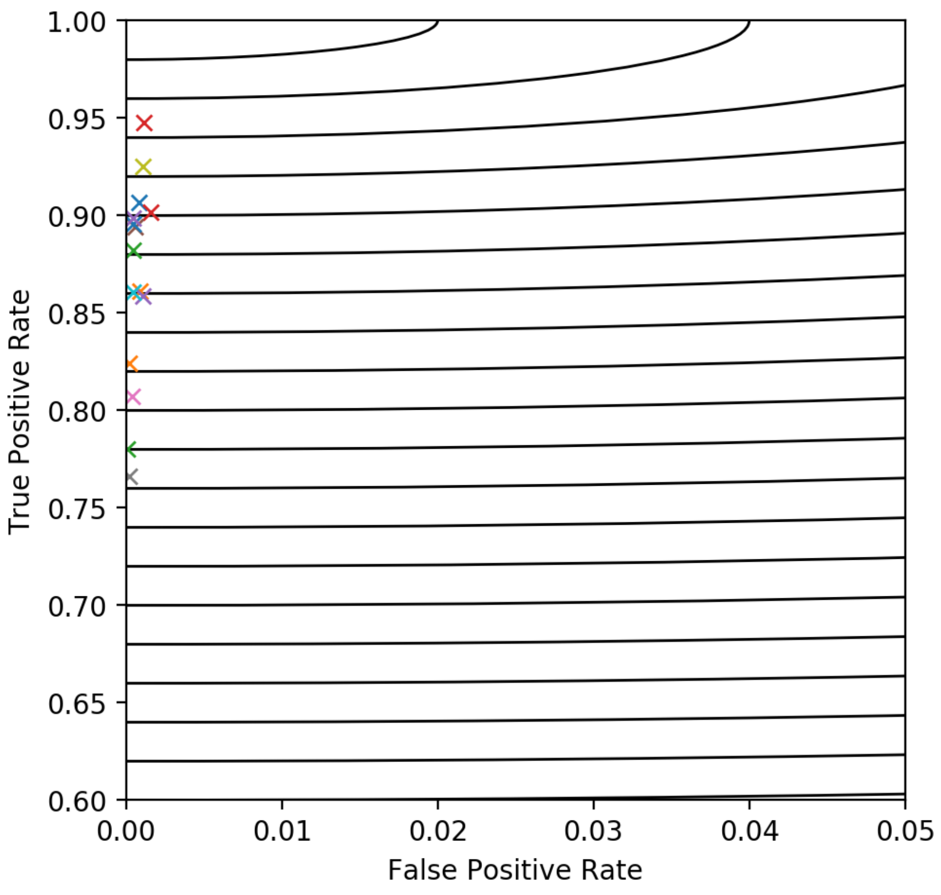

Figure 5.14: The average performance of classifiers using additional features. We can see the drastic decrease of FP_rate when compared to classification without the additional features in Figure 5.6.

Chapter 6

Conclusion

We presented two approaches towards classifying incidental behavior for a SIEM event log dataset. The first approach, based on sequence alignment, didn't work, which we believe is caused by the nature of the dataset.

For the second —feature vector —approach, we designed several different configurable classifiers and then evaluated their performance on many different subsets. Additionally, we incorporated methods to deal with the imbalanced nature of the dataset.

We observed that for some configurations of classifiers, balancing the dataset (most notably with cost-based methods) worsened the performance. Generally, however, the performance improved, increasingly with the dataset imbalance. Interestingly the balancing of the dataset did not have the same effect for Decision trees and Random forests even though Random forests use Decision trees.

From the experiments in Chapter 5 we found that the best performing classifier was the Two-layer classifier consisting of Random forests with 10 Decision trees that were learned using the dataset balanced with sampling using SMOTE for oversampling and Tomek links for undersampling.

6.1 Further development

Future work on this topic would include wrapping the model/pipeline in an application that would handle integration with the analyst platform for handling alerts. It would work as described in Algorithm 1. It could be further expanded by some Active learning methods to further improve the accuracy of the model with feedback from the analysts.

Concerning the model, further research would be needed to improve the method described in Section 4.2. Additional data would be beneficial to experiment with the method.

Appendix A

Bibliography

[1] Scikit-learn precision recall tutorial.

[2] RICHARD BELLMAN. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.

[3] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[4] Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16 – 28, 2014. 40th-year commemorative issue.

[5] Yoav Freund and Robert E. Schapire. A short introduction to boosting. 1999.

[6] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, March 2003.

[7] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21:1263–1284, Sept 2009.

[8] G. Hinton, O. Vinyals, and J. Dean. Distilling the Knowledge in a Neural Network. *ArXiv e-prints*, March 2015.

[9] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[10] Accelrys Inc. Profile hidden markov model.

[11] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[12] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5, 2017.

[13] Christopher Olah. Understanding lstm networks.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[15] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[16] P. Pudil, J. Novovičová, and J. Kittler. Floating search methods in feature selection. *Pattern Recognition Letters*, 15(11):1119 – 1125, 1994.

[17] Guido Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[18] Stan Salvador and Philip Chan. Toward accurate dynamic time warping in linear time and space. *Intell. Data Anal.*, 11(5):561–580, October 2007.

[19] Zhengzheng Xing, Jian Pei, and Eamonn Keogh. A brief survey on sequence classification. *SIGKDD Explor. Newsl.*, 12(1):40–48, November 2010.

Appendix B

CD content

```
CD
├── notebooks ..............code sources in IPython notebooks format
├── results ..........................................................
│   ├── res_c_n ......................the graphs generated from results
│   ├── res_c_n.p .......................................the results files
│   ├── confusion_dtw.png ..........the confusion matrix using DTW
│   └── confusion_fastdtw.png .the confusion matrix using fastDTW
├── thesis .......................the source files of this thesis in LaTeX
│   └── zav_prace.pf ...............................thesis assignment
├── thesis.pdf ............................................this thesis
└── tree.dot ....the graph of a tree classifier with extended features
```

The data we used in this thesis' experiments were not included on the CD as they are bound by an NDA with the data originator. For further clarification contact the thesis supervisor.