MASTER'S THESIS | LUND UNIVERSITY 2016

Intrusion Detection System by Statistical Learning

Julian Kroné, Meris Bahtijaragic

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-26

Intrusion Detection System by Statistical Learning

(Using keyword features to classify HTTP requests)

Meris Bahtijaragic
dat11mba@student.lu.se

Julian Kroné
dat11jkr@student.lu.se

June 29, 2016

Master's thesis work carried out at Digifort AB.

Supervisor: Pierre Nugues, Department of Computer Science, Faculty of Engineering, Lund University pierre.nugues@cs.lth.se
Markus Millbourn, Chief Executive Officer, Digifort Sverige AB
markus@digifort.se

Examiner: Jacek Malec, jacek.malec@cs.lth.se

Abstract

A web server intrusion is when a user gains unauthorized access to resources. This is often accomplished using code injection attacks. Intrusion detection systems today often utilize regular expressions to detect code injection attacks. Some attempts have been made to merge the fields of web security and machine learning. However, they often simply distinguish intrusion attempts from regular requests without detailed classification.

In this thesis, we separate benign requests from malign ones by determining the intention of a request. During our process, we found that request intentions are not always easily separable into good or bad. There are certain types of requests that appear to be malicious, but are actually benign. We present a novel approach to multinomially classify requests based on their textual representation.

We explore three data representation methods, as well as four classification algorithms. These algorithms are compared and their applicability is discussed in the context of an intrusion detection system: Triggerfish. Finally, we report results that reach an accuracy of 99.51%.

Keywords: Classification, data mining, intrusion detection, web security

Acknowledgements

We would like to thank Pierre Nugues, our supervisor, for his support and guidance throughout this thesis. We would also like to thank the guys at Digifort, Markus and Henrik, for all the data, expert advice, and coffee. Thanks to Pedregosa et al. (2011) for developing the library, scikit-learn, which has made all of this possible. Last but not least, we would like to thank Ylva Nilsson for reading our report and providing valuable feedback.

Abbreviations

BEN Benign

CV Cross validation

DBSCAN Density Based Spatial Clustering of Applications with Noise

DTC Decision Tree Classifier

FN False Negative

FP False Positive

IDS Intrusion Detection System

KNN K-nearest-neighbors

LOO Leave-one-out

LR Logistic Regression

RI Rand Index

SQL Structured Query Language

SVM Support Vector Machine

TN True Negative

TP True Positive

XBD XSS by design

XSS Cross-site scripting

Contents

1 Introduction **9**
1.1 Background . . . 9
1.2 Related work . . . 10
1.3 Digifort and the Triggerfish platform . . . 11
1.3.1 Triggerfish client . . . 11
1.3.2 Triggerfish backend . . . 12
1.4 Purpose . . . 14
1.5 Limitations . . . 14
1.6 Contributions . . . 14
1.7 Report outline . . . 15

2 Approach **17**
2.1 Cross Industry Standard Process for Data Mining . . . 17
2.1.1 Business understanding . . . 17
2.1.2 Data understanding . . . 18
2.1.3 Data preparation . . . 20
2.1.4 Modeling . . . 22
2.1.5 Evaluation . . . 23
2.1.6 Deployment . . . 23

3 Algorithms **25**
3.1 Clustering algorithms . . . 25
3.1.1 K-means . . . 25
3.1.2 DBSCAN . . . 27
3.2 Classification algorithms . . . 28
3.2.1 Decision tree . . . 29
3.2.2 K-nearest-neighbors . . . 30
3.2.3 Logistic regression . . . 31
3.2.4 Support vector machine . . . 32

4 Evaluation Measures **35**
4.1 Clustering evaluation . . . 35
4.1.1 Purity . . . 35
4.1.2 Normalized mutual information . . . 36
4.1.3 Rand index . . . 36
4.1.4 F1 measure . . . 36
4.2 Classification evaluation . . . 36
4.2.1 Confusion matrix . . . 37
4.2.2 Learning curve . . . 38

5 Results **41**
5.1 Clustering results . . . 41
5.2 Classification results . . . 42

6 Discussion **47**
6.1 Results . . . 47
6.2 Possible applications . . . 48
6.2.1 Clients . . . 48
6.2.2 Backend . . . 49
6.3 Achievements . . . 49

7 Conclusions **51**
7.1 Summary . . . 51
7.2 Future improvements . . . 51
7.2.1 Data set improvement . . . 51
7.2.2 Feature engineering . . . 52
7.2.3 Further hyperparameter optimization . . . 52
7.2.4 Adding more classes . . . 52
7.2.5 Hybrid trees . . . 52

Bibliography **53**

Appendix A Final Decision Tree **57**

Appendix B Results **59**

Chapter 1
Introduction

This chapter describes the context, background and purpose of the thesis. By explaining the objectives, limitations and outlines of the report, it aims to provide an understanding of the work carried out in order to produce the results.

1.1 Background

Web server intrusions are often accomplished by code injections. In this thesis we will focus on two types of code injection techniques: SQL injection and cross-site scripting.

The SQL injection (SQLi) attack is considered to be one of the more severe attacks in web security. If successful, the attacker can potentially manipulate or retrieve any data stored in the database. Typically, it is achieved by sending a part of a SQL query as input to an application. The injection is successful if the input is parsed as a SQL query, instead of plain data.

This attack is generally made possible by applications mixing SQL queries with user supplied data. For example: `select text from posts where id=$id`, is a query where the parameter `$id` is supplied by the user. The logic of the query can be changed by setting `$id` to `1 or 1=1`. This results in the following SQL query being run: `select text from posts where id=1 or 1=1`. The injected query will then return all posts instead of a single post.

The cross-site scripting (XSS) injection attack targets users of a web application, more than the web application itself. Generally, two forms of XSS exist: reflected and stored. Reflected XSS is achieved by crafting a URL with a HTML parameter containing HTML or javascript. Stored XSS is achieved by sending data in the form of HTML or javascript to the web application, where it is stored. The injection attack is successful if the web application presents this input as HTML/javascript, instead of plain text. If successful, the attack can be used by the attacker to steal information shared between the user and the web application. For example: authentication details, private messages, and HTTP

Cookies. Additionally, the attack can be used to deface a website by injecting a script which replaces the body of a web page with a message from the attacker. Detecting XSS is made more complex by the fact that some web applications intentionally let users inject HTML/javascript to the server. The reason behind this is to allow administrators to use the web application to edit HTML/javascript stored on the web application. We consider this a special case and we will handle XSS by design (XBD) as a benign form of injection.

The most usual approach to detect intrusion attempts in web applications is to use a regular expression to identify patterns in HTTP parameters. The purpose of the patterns is to detect code injection. Maintaining a regular expression can be a cumbersome process, which potentially yields false positives and false negatives.

This thesis explores another approach, namely machine learning, which is a way to recognize patterns and make predictions based on input data. There are two approaches to group instances within the area of machine learning, clustering and classification. Clustering is an unsupervised method to group instances with similar attributes without previous knowledge of the data, while classification is a supervised method to predict the class of a new instance based on previously annotated data.

1.2 Related work

There have been several attempts to apply machine learning theory to the field of intrusion detection, many with successful outcomes. This suggests that machine learning theory is an applicable approach to separating normal requests from malicious ones.

- Wressnegger et al. (2013) explore the possibility of using n-grams as features when trying to detect malicious requests. They compare two approaches to solve this: Anomaly detection and Classification, where the first one is accomplished by creating a model of normality to detect abnormal requests. They also apply these approaches to numerous different datasets, where the HTTP dataset is most in line with our thesis. As a result they achieve 100% true positives, when using an acceptance rate of 0.01% false positives. These figures are reached using a 4-gram model to train a binary SVM classifier.

- Cheon et al. (2013) develop a system to detect and prevent SQL injection attacks. They extract the parameters from HTTP requests and translate them to a numerical representation using SQL keywords as patterns. They then classify the parameters using a Bayesian classifier. With this solution, they are able to achieve a high accuracy of detecting SQL injections.

These two studies share the same approach and are both applied to a binary problem: intrusion attempt or not. To the best of our knowledge, no investigations regarding the potential of applying these theories to systems that identify multiple classes have been attempted. This thesis aims to minimize this deficiency by introducing a novel approach to a common problem.

1.3 Digifort and the Triggerfish platform

Digifort is a computer security start up founded in 2009 in Lund, Sweden. The company offers web security services such as penetration tests, code reviews, and threat modelling. Digifort also offers a web application IDS, *Triggerfish*, designed to automatically detect intrusion attempts and provide customers with security related insights into their web application.

The Triggerfish platform is a collection of client side libraries and server side processing tools designed to detect security vulnerabilities, malicious activity and attacks aimed at web servers. Triggerfish consists of two main working parts: the client and the backend. Figure 1.1 shows a diagram of how these parts work together.

1.3.1 Triggerfish client

There are Triggerfish client implementations for a wide variety of web server frameworks – for example Ruby On Rails, PHP, and Java Servlets. The primary responsibility of a Triggerfish client is to implement so called *triggers*. Triggers are API calls which notify the Triggerfish analysis server of an event. These triggers can for example be:

`SuccessfulLogin` Notify the Triggerfish analysis server of a successful login to the web application.

`FailedLogin` Notify the Triggerfish analysis server of a failed login to the web application.

`XSSAttemptInParameter` Notify the analysis server of a cross-site scripting attempt.

`SQLInjectionInParameter` Notify the analysis server of an attempted SQL Injection attempt.

Note that under some circumstances, triggers are called upon automatically – for example if the Triggerfish client detects a cross-site scripting attack. The `XSSAttemptInParameter` is then called without the need for the trigger to be placed within the main application logic. The full set of triggers can be found in the Triggerfish documentation (Digifort Sverige AB, 2015).

After an event has been triggered, its context is saved by collecting the data contained in the web request – parameters containing personal information such as personal identification numbers, passwords, and full names are filtered as configured in the client. When the context has been collected, it is serialized as specified by a common protocol defined both in the clients and the analysis server. The serialized event is then sent to the analysis server for further analysis.

In order to reduce the performance overhead on the web server using the Triggerfish client, the clients are as light as possible in terms of logic. This results in the Triggerfish code injection detection systems being more prone to false positives than systems where more computational time is used. Moreover, if no triggers have been called upon, no event will be sent to the backend – this is both to reduce the load on the web server and the backend.

1.3.2 Triggerfish backend

The primary function of the Triggerfish backend is to process information sent from the clients and present it in a comprehensible and useful way. This is accomplished by using a modular architecture, where each module has a limited responsibility. The modules which will be discussed in this section are: the analysis engine known as the *analysis server*, and the presentation modules known as the *frontend* and the *reporting service*. This section explains in detail the software architecture of the Triggerfish backend.

Analysis server

The analysis server analyzes incoming events. Events are annotated with more information and persisted in the database, or filtered if the analysis deems the event uninteresting.

The analysis server has many features, for example:

- Geo IP lookups, to resolve which geographical location an IP address originates from.
- Custom alarm aggregation, to group events together by custom criteria. For example: Many failed logins from the same IP during a 5-minute interval should be grouped together as a brute-force login attempt.
- Saving an analyzed event to the database.

Each of the features are implemented as *filtors*, which is a word play on the words *filter* and *mathematical functor*. Each filtor takes an event as input and returns either an event, modified in some way or unchanged, or nothing. The filtors are chained together, and if one filtor returns nothing to the next filtor in the chain, the chain is broken and the event is discarded. However, if the event makes it to the end of the filtor chain, it is saved in the database such that the details of the events can be shown in the frontend.

Frontend and reporting service

The main way for a customer to interact with Triggerfish is through the frontend. It is a web application which displays the events that have been triggered by a customers Triggerfish client. Its primary responsibility is to present the information in the form of charts, lists, or detailed views to the user. The frontend also provides the user with means of filtering or manipulating the information shown in the interface. Figure 1.2 shows an example of the Triggerfish frontend.

In order to present the information, the frontend gathers its data from another module, called the *reporting service*, which provides an API in the form of web resources. This API is called upon with different parameters, which are specified by the user in the frontend interface.

All database manipulation functions are implemented in the reporting service, for example: collection of graph data to be displayed in the frontend, the collection of all data regarding an event such that a customer can view a specific event, archiving of alarms and vulnerabilities, which have in some way been amended by the customer.

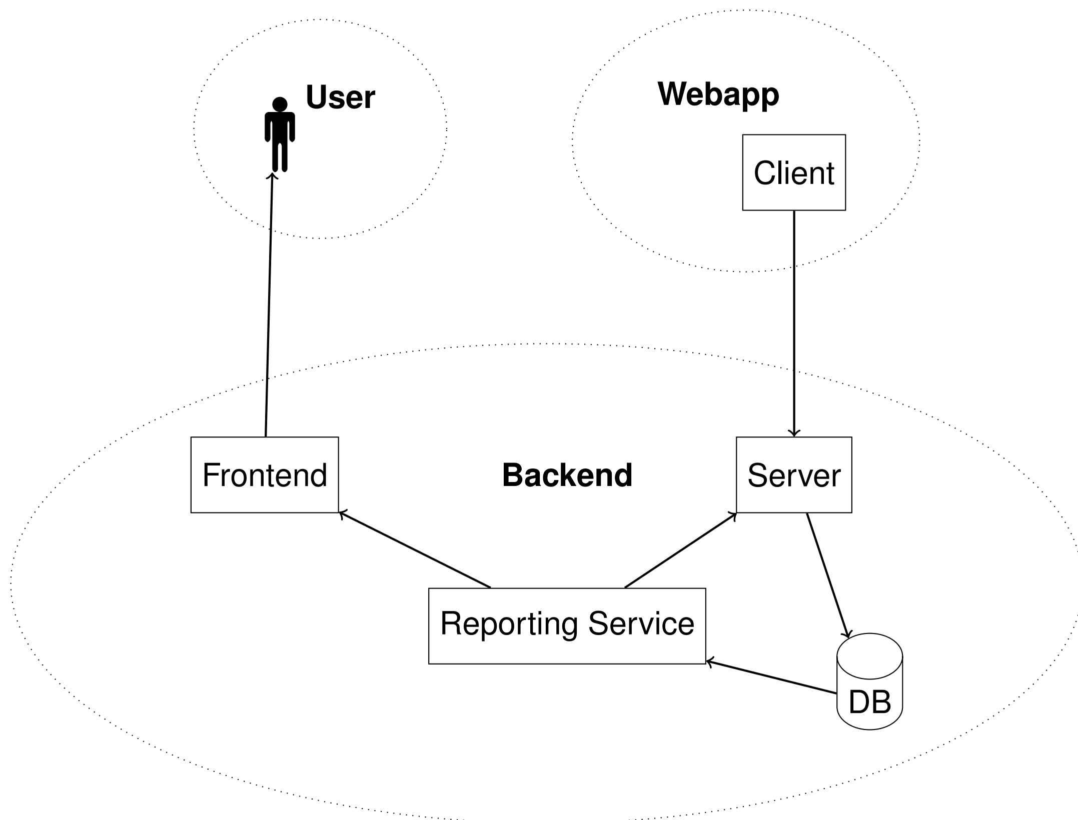


Figure 1.1: A simplified scheme of the Triggerfish architecture, where arrows illustrate regular flow of data.

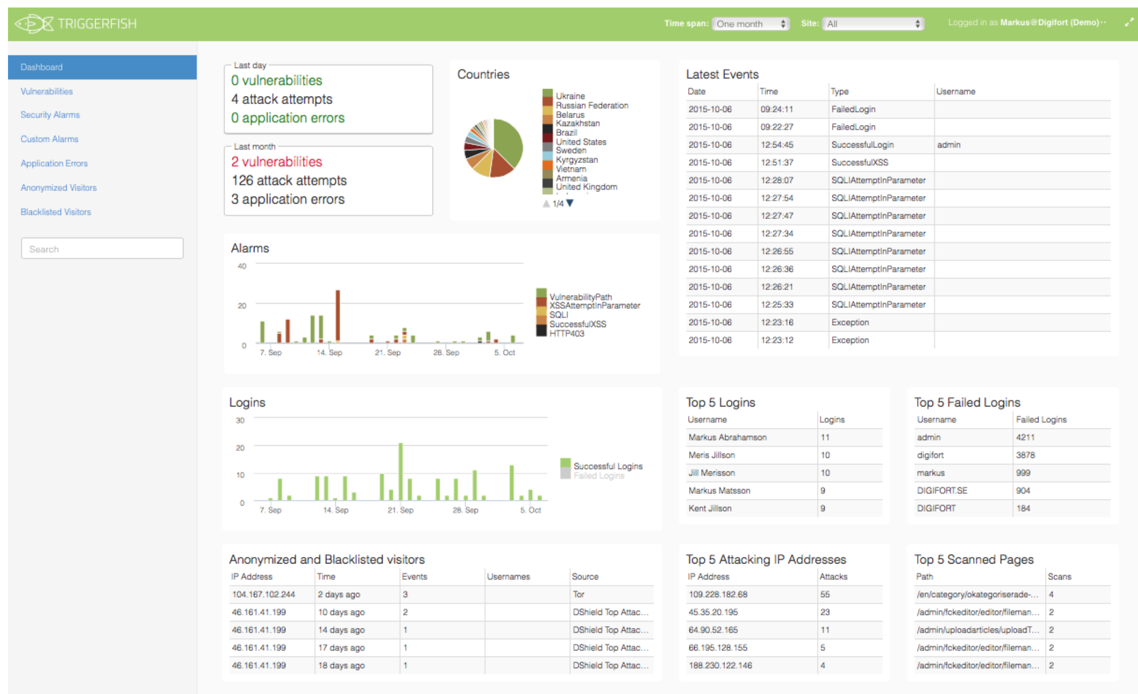


Figure 1.2: Figure showing the dashboard in the Triggerfish frontend.

1.4 Purpose

The purpose of this thesis is to provide Digifort with means of improving the accuracy of their classifications, more specifically, to answer the following question:

> Is machine learning an applicable approach to detect web server exploitation attempts?

In order to answer this question, we have broken down the problem into several goals:

Reach an *acceptable* accuracy
: This goal is somewhat arbitrary, an acceptable accuracy is not defined. False positives, i.e. classifying benign requests as malign, is more acceptable than the opposite. Primarily, a method for measuring and comparing classification models should be suggested. Additionally, the model needs to be able to distinguish between XSS and XBD.

Account for model performance and size requirements
: The models should be described in a way which makes their selection clear given the context, in respect to performance and memory requirements. It should also be clear if there are any trade-offs regarding model accuracy and model performance/size.

Describe how these models can be implemented into Triggerfish
: It should be clear how each model could be used in the Triggerfish platform. Importantly, resource constraints should be taken into consideration for each suggested implementation.

Find a model that provides a second opinion and a degree of confidence
: Compare the resulting models and find at least one that provides a measurement, in addition to a classification, which can be used as a basis for the classification.

1.5 Limitations

This thesis aims to investigate the possibility to apply machine learning methods to the area of web security. The thesis is limited to threat detection, using a distinct set of algorithms described in Chapter 3. These algorithms were chosen for their diversity, while still being common practice. Since we only have access to test data that has previously been classified by Triggerfish, we have opted to limit our dataset to requests classified as either XSS or SQLi by Triggerfish, while hopefully maintaining diversity of the attacks in the dataset. The execution and testing during the thesis has been carried out on our own MacBook Pro's, which has limited performance.

1.6 Contributions

We have both been involved in each step throughout this thesis to some extent, though some parts has been more divided than others. Meris focused more on data collection,

data preparation within the database and clustering exploration, while Julian has been more involved in feature engineering, classification exploration and evaluation setup.

The report has been divided somewhat equally, though Meris has been more involved in Digifort background and information, and contributed more in the approach chapter. Julian has contributed more regarding evaluation measure and algorithm descriptions. The remainder of the report can be seen as equally contributed to by both parties.

1.7 Report outline

Following this chapter, the report is structured as follows:

Chapter 2. Approach describes the workflow used during this thesis, and brings up important choices taken and discoveries found.

Chapter 3. Algorithms describes the algorithms chosen to produce the values used to evaluate the thesis.

Chapter 4. Evaluation measures describes the measures chosen to produce the details of the thesis results.

Chapter 5. Results presents the results gained from the thesis, and provides a short description of them.

Chapter 6. Discussion evaluates the results presented in Chapter 5, and discusses their validity and compares them to our goals. Furthermore, it proposes ways of applying the results to Triggerfish.

Chapter 7. Conclusions summarizes the thesis and proposes future work and improvements.

Chapter 2
Approach

This chapter describes the process model standard we used to derive the thesis workflow. It describes the purpose of each step, as well as the process and considerations. It also outlines the tools used and the theory applied to produce the results described in Chapter 5.

2.1 Cross Industry Standard Process for Data Mining

The Cross Industry Standard Process for Data Mining (CRISP-DM) is a process for developing data mining projects introduced by Chapman et al. (2000). CRISP-DM consists of six phases: *Business understanding*, *Data understanding*, *Data preparation*, *Modeling*, *Evaluation*, and *Deployment*, each consisting of several tasks. It is an agile process, where adjacent phases are often iteratively explored and implemented. Figure 2.1 shows a visualization of the CRISP-DM process model.

2.1.1 Business understanding

The first phase is all about understanding the problem from a business point of view, background and objectives are brought up during this phase. This is also when the project plan is produced (Chapman et al., 2000).

During this phase, we discussed the architecture of the Triggerfish platform. We came to the conclusion that XSS and SQLi detection systems would need some improvement. Today, a couple of libraries are used for the detection of SQL and XSS injections – whenever one of these libraries classifies something incorrectly, and produces a false positive, manual intervention has to be taken. This manual intervention is either in the form of correcting the library, or adding a manual filter containing a regular expression allowing a specific form of payload.

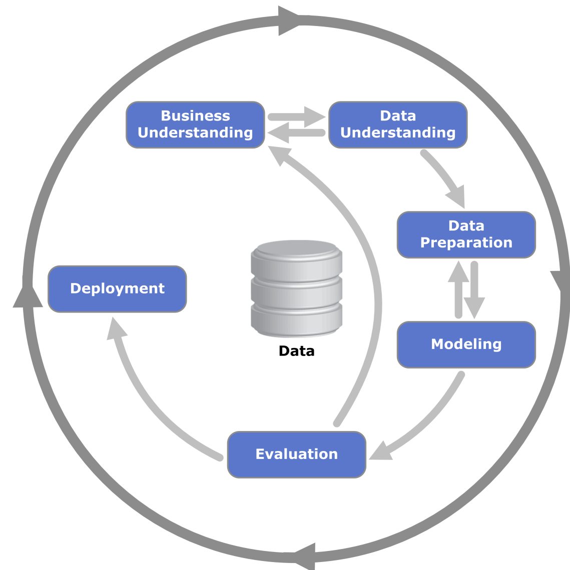


Figure 2.1: A visual representation of the major phases in the CRISP-DM process

False positives can also come in the form of XSS injections by design. For example when editing a WordPress site, one is actually injecting HTML into an editing form. When making a manual intervention against this specific type of payload, one must take into context both the web resource being targeted (e.g. `/wp-admin/edit.php`), and the specific parameter key pointing to the value of the payload (perhaps also that a username is set, e.g. a logged in user is doing the injection).

The first conclusion drawn was that the objective could not be defined as “We need to find code injections”, since some customers might have XSS, or HTML injection pages by design. We need to draw a boundary between malicious code injections, benign code injections, and completely benign payloads (in the sense that they are not a code injection). Another requirement was to provide a confidence, for example in the form of a statistical probability, in addition to a classification.

We limited the scope of the thesis to research and dataset collection, and as such we were free to choose any machine learning software stack available. We decided on using Python in conjunction with the machine learning library `Scikit-learn` created by Pedregosa et al. (2011) due to the wide variety of modelling options it offers.

2.1.2 Data understanding

The data understanding phase starts with an initial data collection, followed by data familiarization activities such as data exploration, detecting interesting subsets, finding correlations and form hypotheses (Chapman et al., 2000).

Initial data collection

In the Triggerfish database, only certain parts of an event are indexed. When data is indexed in a database, it means that this data is searchable. The indexed parts are for example: event

type, event id, classification, and time of the event – the rest is stored as a binary blob in the database. Whenever more details are required, one must deserialize the binary blob and extract the information required.

For this purpose we constructed a tool. This tool first uses the Triggerfish database index to find events of a specific type. In our case, the tool collects the event id for XSS and SQLi events and deserializes them, then extracts the HTTP request and saves it, indexed with all parameters.

Data exploration

When a user wishes to view a certain web resource, she requests this resource (mainly through a browser) by sending a HTTP request to the server containing this resource. Depending on the kind of request, and if the request is valid, the server responds with a HTTP response.

A HTTP request follows a particular specification, as defined by Fielding et al. (1999) and can look like this:

```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: application/x-www-form-urlencoded
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

licenseID=string&content=string&/paramsXML=string
```

A request starts with with a request line, followed by zero or more headers, followed by an optional message body. At a minimum, a HTTP request consists only of a request line, but the host header may be mandatory depending on which HTTP version is used by the server.

The data in the Triggerfish database is structured as an event, where an event contains both a classification, an event type and its corresponding HTTP request. The data in a HTTP request is structured as key-value pairs, consisting of the GET, POST, header and cookie parameters. These key-value pairs will from now on be referred to as parameters.

To better understand what makes a request malign, we consulted an expert at Digifort. We found some examples of code injection events and tried to identify what parts of an event needed to be inspected to draw the conclusion that it was malign. We analyzed the contents of these events and reduced the amount of context needed to determine the intent of the payload. We concluded that we should look at the HTTP request at the parameter level, partly because one parameter could be a XSS attempt, and another parameter could be a SQL injection attempt. It is only necessary for a single parameter in a HTTP request to be malign for the whole request to be considered malign. Thus, our data should be viewed as a collection of parameters instead of a whole HTTP request.

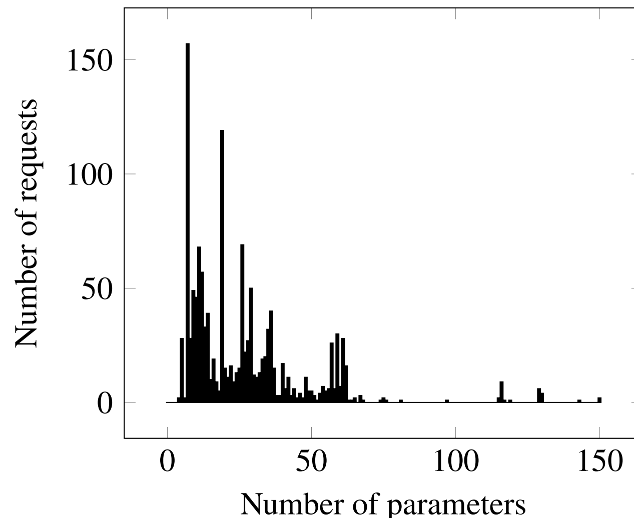


Figure 2.2: A histogram illustrating the distribution of parameters per request.

Data description and quality

The initial dataset consisted of 1,350 HTTP requests, where at least one HTTP parameter had either been classified as an XSS or SQLi by Triggerfish. The total number of parameters was 38,801 with 4,191 unique values. Figure 2.2 shows the distribution of parameters per request. Since there often are a large amount of parameters in a request, each request would most likely provide at least one malicious parameter, and several benign parameters. By extracting only malicious request, we would actually acquire a dataset consisting of parameters of both malign and benign intentions.

2.1.3 Data preparation

During the data preparation phase, all the necessary steps to convert the initial dataset into the final dataset needed by the model are covered. This is done through iterations together with the modeling phase, since new important information about the data might surface. Tasks such as limiting the dataset, final attribute selection, data transformation, and data cleaning are performed during this phase (Chapman et al., 2000).

Data selection and cleaning

In order to normalize the text based data, we first had to eliminate possible HTML entities and URL encoding, that can either be a way for an attacker to mask an attack, or done automatically by a web browser. Since our data exploration concluded that only the parameter value could decide if a request was malign, we reduced the dataset from 38,801 key-value pairs to 4,191 by eliminating duplicates. Initially, we wanted to reduce the imbalance of the dataset classes, even though duplicate values indicate that the values are recurring. Later we concluded that this had another positive effect: the importance of a specific instance is not affected by the frequency of the instance. To balance the classes in

our dataset, we continually added more instances during the process. In the end we had 2,055 SQL, 204 XSS, 721 XBD, and 2,980 BEN.

Data construction

To facilitate and automate the evaluation process, we created a tool for annotating data. Each instance would manually be assigned one of four classifications based on its textual representation: XSS, SQL, XSS by design (XBD), benign (BEN). Where the latter two are non malicious, or negative, classifications. During the annotation, we assumed that the data consisted of a partition of four classes. In order to train a model, we had to convert the text based value to a feature vector representing the meaning of the value. Several approaches to this were attempted, at first a basic approach of representation called *n*-grams was explored.

An *n*-gram is a sequence of *n* tokens, where a token is a predetermined number of characters in sequence. In our case, a token is simply one character, given that the value of a parameter is not necessarily derived from any specific natural languages. To convert a dataset to a set of features, first a set of all the *n*-grams from the whole training set is collected – this set corresponds to the vector space from which each instance in the dataset is modeled after. Then each instance x, is converted to a feature vector v, where each element in the feature vector, v_i, describes the frequency, or count of the feature in the instance. For example, given the vector space, `[a, b, c]`, the instance `aabbbca` would have the feature vector `[3,3,1]` – assuming that each dimension counts the occurrences of the characters in the instance. Figure 2.3 illustrates this.

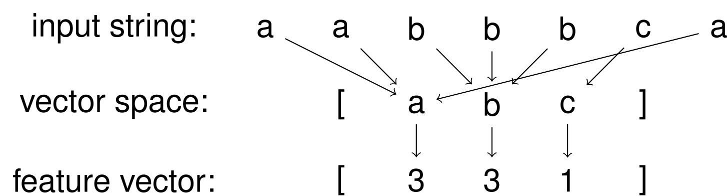


Figure 2.3: An illustration of how a 1-gram representation can be derived from text.

Initially we used models with 1- or 2-grams. This allowed n-grams which occur in SQL injection attacks to be represented. For example, in the case of a 1-grams the string literal (`'`), and in the case of 2-grams the start of a comment (`--`). We concluded that a combination of both 1- and 2-grams would have to be used to capture the most discriminating features. This results in a quite large vector space with features that have a large variation of discrimination. Some features, such as single alphanumeric characters, can occur in all four classes without necessarily indicating an attack – which makes short alphanumeric sequences a nondiscriminating feature. Using 1-grams resulted in a model which has 90 features. A combination of 1- and 2-grams resulted in a model with 3,533 features, which increased the time required to run the tests.

In order to decrease the number of nondiscriminating features, we studied the syntax of injection attacks. We found that for each class we could typically expect one set of languages to be used. Some kind of SQL syntax would always have to be used to accomplish SQL injection attacks. Typically a combination of javascript and HTML would be used in

XSS payloads, but would also occur in our negative class: XBD. Each of these languages is precisely and finitely defined by its syntax and grammar. We took advantage of this and collected the reserved keywords from SQL, javascript, and HTML. Then we used these keywords as discriminating features. By extracting these features from an instance, the class could be represented precisely. This was our final feature extraction algorithm, with which we achieved highest accuracy and performance. Using this method, we found 166 features. Figure 2.4 shows an illustration of how this representation can be derived from a parameter in the case of a SQLi.

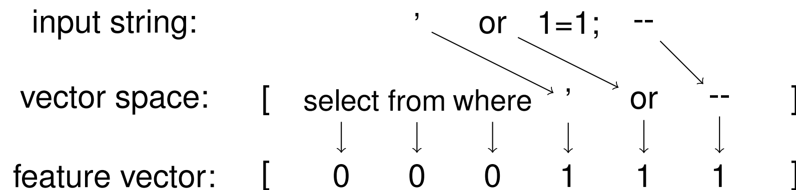


Figure 2.4: An illustration of how a keyword representation of a SQLi can be derived from text.

2.1.4 Modeling

During this phase, various modeling techniques are selected and parameters are calibrated in order to find the optimal setting. A test design is formed to aid in assessing each model (Chapman et al., 2000).

Modeling techniques

There is a wide variety of methods for modeling both classification and clustering problems. Since the project is constrained in time we had to find ways to pick a set of techniques and thoroughly investigate them.

For clustering, we initially chose `k-means` and `DBSCAN` since they are both described as general purpose algorithms, and having different approaches to the same problem. There exists a huge variety of classification models, though most of them are derived from the same subset of core models. From these core models, we chose four different algorithms to investigate:

Logistic regression provides a confidence measurement in the form of probability estimates.

Decision tree visualizes the solution in a way that is easy to understand, furthermore it has a low time complexity regarding classification.

K-nearest-neighbors is easy to understand.

Support vector machine is a commonly used algorithm for classification.

Some of these algorithms have hyperparameters, that can be tuned to better suit a specific dataset. The hyperparameters for each available model were optimized iteratively with the evaluation phase, to determine if a result was acceptable, or had the opportunity to be improved. A more detailed description of these algorithms can be found in Section 3.2.

Test design

Before creating these models, we had to determine how we could compare them to each other. For the clustering models, we chose four common external evaluation criteria: *purity*, *normalized mutual information*, *Rand index*, and *F1 measure*. For the classification models, we chose to both utilize *learning curves* to see the models learning capabilities, and *confusion matrices* to determine exact measures such as *accuracy*, both average and per-class. Chapter 4 describes the details of these evaluation measures.

2.1.5 Evaluation

As mentioned, this phase is often reached iteratively throughout the entire development process, each time with more understanding of the problem and the data. The purpose of this phase is to evaluate the results acquired when utilizing the test design, mentioned in the previous section, and to decide upon how to proceed (Chapman et al., 2000).

For clustering we started out evaluating our groups manually, by simply examining the contents of the clusters. Eventually, we streamlined this process by using the test design we decided upon. For classification, we initially used cross-validation from scikit-learn to evaluate the models, and decide how to tune the hyperparameters. In the end, we used the described test design to evaluate and rank our classification models.

2.1.6 Deployment

This final phase includes the production of the final report and the final presentation. It also includes a complete project review that mentions the development process, the evaluated project outcome, and future project improvements (Chapman et al., 2000).

Chapter 3
Algorithms

This chapter describes the theory behind the clustering and classification algorithms we have used. We explain the k-means and DBSCAN algorithms, as well as the decision tree, k-nearest-neighbors, logistic regression, and support vector machine algorithms.

3.1 Clustering algorithms

Clustering analysis is a subset within unsupervised learning, used to detect patterns within data. The objective of a clustering algorithm is to divide a set of instances, $x_n, n = 1, \ldots, N$, into groups, $c_k, k = 1, \ldots, K$, where the grouped instances have a higher similarity towards each other, and a lower similarity towards instances in other groups. For our cluster analysis we chose the two commonly used algorithms *k-means* and *DBSCAN*.

3.1.1 K-means

K-means, proposed by Lloyd (1982), is a clustering algorithm with the purpose of dividing N instances of data, represented by vectors in I dimensions, into K clusters. The means, i.e. the centers of each cluster, is denoted m_k. The distance between the points can be calculated using an arbitrary distance metric, $d(x, y)$, for example Euclidean distance assuming that $x_n \in \mathbb{R}$ and $m_k \in \mathbb{R}$. The goal is to assign a cluster to each point and minimize the intra-cluster distance to the cluster mean. This algorithm requires the desired number of clusters as input from the user, as it has no way of resolving this on its own. This can be seen as both an advantage and as a disadvantage, depending on what the user knows about the dataset.

The algorithm consists of four different phases:

1. Initialize each mean m_k by selecting K values from the data domain of the points in x_n. There are a number of ways of initializing the means (Hamerly and Elkan, 2002), for example:

Forgy Set m to be K random observations from the data set.

Random Partition Also known as Lloyd's algorithm. Randomly pick K values within the domains of the data set.

In this thesis, a slightly more advanced initialization algorithm `k-means++` will be used for this step, the algorithm will be described further below.

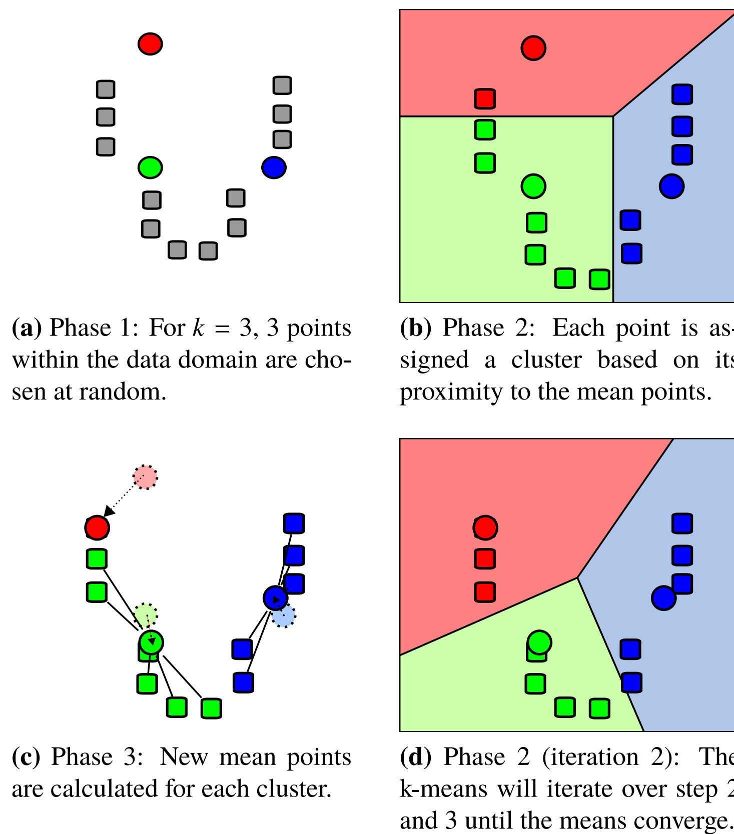
2. Each point x_n is assigned to a cluster based on the proximity defined by the distance metric as

$$c_k : \{n | d(x_n, m_k) \leq d(x_n, m_l), l \neq k\}.$$

3. For each cluster, a new mean is calculated:

$$m_k = \frac{\sum_{n \in c_k} x_n}{|c_k|}, \forall k. \tag{3.1}$$

4. Repeat step 2 and 3 until the means have converged.



(a) Phase 1: For $k = 3$, 3 points within the data domain are chosen at random.

(b) Phase 2: Each point is assigned a cluster based on its proximity to the mean points.

(c) Phase 3: New mean points are calculated for each cluster.

(d) Phase 2 (iteration 2): The k-means will iterate over step 2 and 3 until the means converge.

Figure 3.1: Illustration of the four first steps of the k-means algorithm with the parameters using the *Random Partition* initialization method. The squares indicate the points of the given dataset, and the circles indicate the current means (Weston.pace, 2007).

The `k-means++` algorithm

The `k-means++` initialization algorithm was proposed by Arthur and Vassilvitskii (2007), where it is shown that the `k-means++` algorithm improves both the running speed and accuracy of k-means. Let X be the input data set and let $D(x)$ denote the shortest distance from a data point, x, to the closest center we have already chosen.

1. Assign the first cluster center, c_1, randomly from X.
2. Assign a new center, c_i, choosing $x \in X$ with probability

   $$\frac{D(x)^2}{\sum\limits_{x \in X} D(x)^2}.$$

   In this step, data points which are further away from a cluster have a higher probability of being assigned.
3. Repeat step 2 until K cluster centers have been assigned.

3.1.2 DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN), is a clustering algorithm for spatial data, i.e. data related to space – defined within a coordinate system. DBSCAN was proposed by Ester et al. (1996) and evaluates the density of points within a given space. It extracts high-density areas, distinguished by a low-density boundary, which are to be clustered together as a group. Apart from the data set, DBSCAN requires only two parameters:

minPts is the minimum number of neighbors that a data point needs to be classified as high-density.

eps (ϵ) is the maximum distance between two data points for them to be considered neighbors.

The points within the data set are divided into three classifications; core, non core, and outliers/noise. A point is a core point if at least minPts are within ϵ distance of it. A point is a non-core point if it has at least one core point within ϵ distance of it, but less than minPts within ϵ distance. Clusters are made up of core- and non-core points, where the non-core points make up its *edges*. Lone data points with too few neighbors, are marked as outliers and do not belong to any cluster. As opposed to k-means, DBSCAN does not require to specify the number of clusters beforehand, and the algorithm has the possibility to find arbitrarily shaped clusters.

The algorithm consists of three different phases:

Finding
Find a point, x_n, that does not already belong to a cluster.

Counting
Calculate the density d_n of x_n where density is the number of points that are within

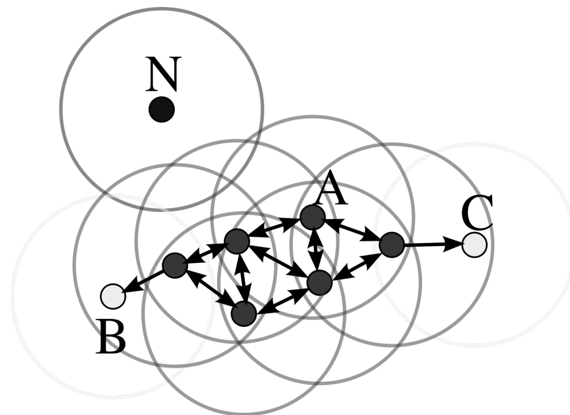


Figure 3.2: Illustration of DBSCAN density algorithm. Core points with high-density are marked with grey, non-core points with lower density marked with white, and outliers with lower density marked with black (Chire, 2011).

a radius of ϵ. If $d_n < minPts$, continue with the next point in phase 1, otherwise proceed to the next phase with x_n as the initial core sample.

$$\begin{aligned} x_n \notin c_k \quad & \text{if } d_n < minPts \\ x_n \in c_k \quad & \text{if } d_n \geq minPts \end{aligned} \tag{3.2}$$

Expanding

Gather the neighbors N_ϵ of x_n as

$$N_\epsilon(x_n) : \{x_i | d(x_n, x_i) \leq \epsilon\}. \tag{3.3}$$

For each point $x_i \in N_\epsilon(x_n)$, calculate the density d_i of x_i Increase the neighborhood set by evaluating the density of the newly visited points x_i.

$$N_\epsilon(x_n) \cup \{x_j | d(x_i, x_j) \leq \epsilon\} \quad \text{if } d_i \geq minPts. \tag{3.4}$$

If $d_i < minPts$, x_i is only added to c_k as a non-core sample. When $|N_\epsilon(x_n)| = 0$, proceed with the next point in phase 1.

Figure 3.2 shows a simple illustration of the DBSCAN algorithm phases, where any of the grey core samples could have been picked as the initial core sample. If any of the points B, C, or N would have had its density calculated, the density would have been too small for any of them to have become core samples. Figure 3.3 shows a real example of DBSCAN using three clusters.

3.2 Classification algorithms

The objective of a classification algorithm is to identify the category a new observation belongs to on the basis of a training set containing classified instances.

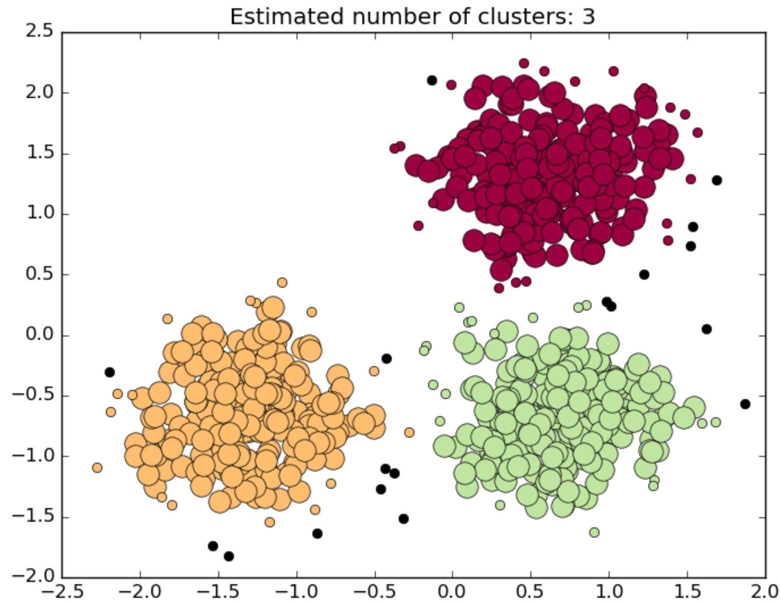


Figure 3.3: DBSCAN example showing three clusters, with black outliers. Thick points are core samples, and thin points are non-core samples (scikit learn, 2014).

3.2.1 Decision tree

A decision tree algorithm extracts key data features that can be used to distinguish one group of instances from another. Based on these features, a tree of decisions can be built in order to classify instances. In our implementation, we use a decision tree algorithm called Classification And Regression Tree (CART), introduced by Breiman et al. (1984). The measure used to determine the optimal data feature is called impurity. Below are two common impurity variants, Gini and Entropy. p_i is the probability of choosing an instance of class i from all the instances available at the current node.

$$p_i = \frac{1}{N}|\{x_j|x_j \in c_i\}| \tag{3.5}$$

$$H_{Gini} = \sum_{i \in C} p_i(1 - p_i) = \sum_{i \in C}(p_i - p_i^2) = \sum_{i \in C} p_i - \sum_{i \in C} p_i^2 = 1 - \sum_{i \in C} f_i^2 \tag{3.6}$$

$$H_{Entropy} = -\sum_{i \in C} p_i \log_2 p_i. \tag{3.7}$$

To build a decision tree, the following algorithm is used:

1. The data at node m is represented by Q, each node has a split criteria $\theta = (j, t_m)$ consisting of a feature j and a threshold t_m. For each node, partition the data Q into $Q_L(\theta)$ and $Q_R(\theta)$ subsets

$$\begin{aligned} Q_L(\theta) &= (x, y)|x_j \leq t_m \\ Q_R(\theta) &= Q \setminus Q_L(\theta). \end{aligned} \tag{3.8}$$

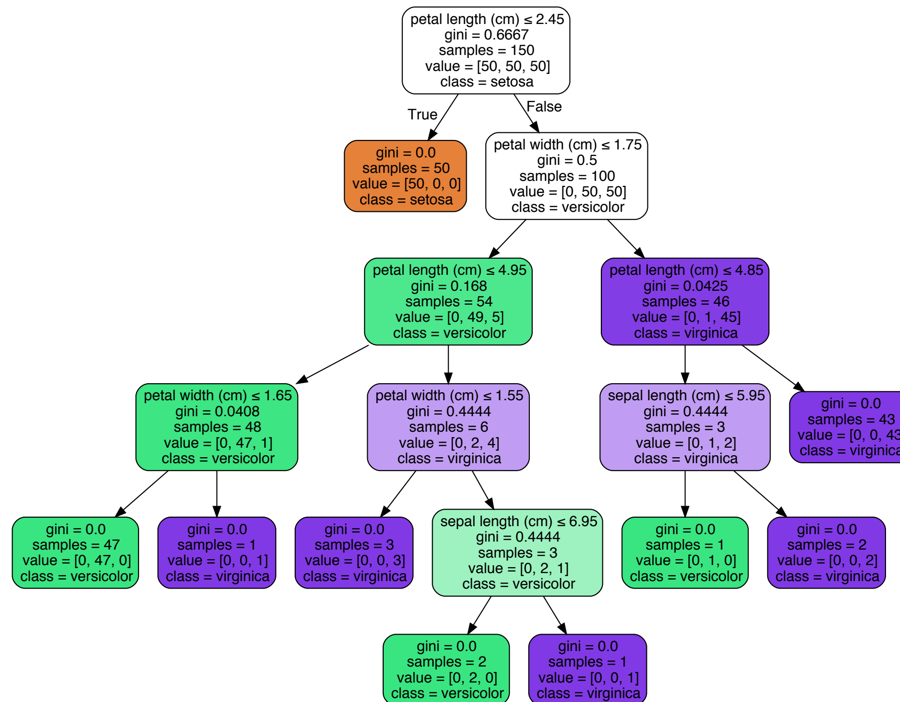


Figure 3.4: An example of a decision tree using scikit-learns built in data set, each node provides its current feature, threshold, impurity and sample information

2. The impurity of this split G is computed using one of the above described impurity measures

$$G(Q, \theta) = \frac{n_L}{N} H(Q_L(\theta)) + \frac{n_R}{N} H(Q_R(\theta)). \tag{3.9}$$

3. Finally the parameters θ that minimize this impurity are selected, and the algorithm is recursively performed for subsets Q_L and Q_R until the chosen stopping criteria is met.

Figure 3.4 shows an example of a finished decision tree.

3.2.2 K-nearest-neighbors

The k-nearest-neighbors algorithm is a relatively simple classification algorithm. Its input is a training set of instances which consists of I numerical features. It is non-parametric in the sense that the distribution of these instances is irrelevant. The algorithm predicts the class label of a new observation by finding the k nearest training set instances to the new observation. The instance is assigned the class which is most common amongst these k nearest neighbors. Additionally, the algorithm is unique in the sense that does not have a training phase.

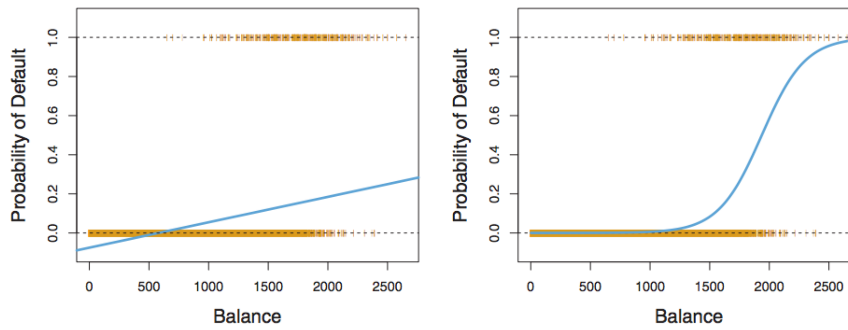


Figure 3.5: An example of how linear regression (left) and logistic regression (right) differ in their probability function when using the simpler binary classification.

3.2.3 Logistic regression

Logistic regression developed by Berkson (1944) is a classification algorithm that is harder to visualize compared to decision trees or K-nearest-neighbors, as it has a more algebraic approach to the problem. The algorithm models the probability that an instance belongs to a particular class using a logistic function. In our case, it means that it models the probabilities that a parameter belongs to each of the classes c_i, where the sum of all probabilities must be 1. Logistic regression derives from linear regression, where the probability of a class instance is represented by the linear function

$$p(X) = \beta_0 + \beta_1 X. \tag{3.10}$$

Logistic regression however uses the logistic probability function

$$p(X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}}. \tag{3.11}$$

Figure 3.5 illustrates how these algorithms finds different representations, the most important difference is that the logistic function always stays inside the probability space [0, 1]. It also attains an S-shaped curve, which contributes to obtaining sensible predictions. The equation in (3.11) is equivalent to another function

$$\log \frac{p(X)}{1 - p(X)} = \beta_0 + \beta_1 X \tag{3.12}$$

where the left-hand side is called logit. As can be seen, the logistic regression model has a logit that is linear in X. This is the reason why logistic regression is also sometimes referred to as log-linear classification (Pedregosa et al., 2011).

The regression coefficients β_0 and β_1 in (3.11-3.12) are estimated using the maximum likelihood function, based on the available training data. It chooses β_0 and β_1 to maximize the function

$$l(\beta_0, \beta_1) = \prod_{i \in c_k} p(x_i) \prod_{x_j \notin c_k} (1 - p(x_j)). \tag{3.13}$$

This function utilizes Equation 3.11 and uses the available training data to map the instances based on classification. The instances of one class should receive higher probability of belonging to that class compared to the other classes instances.

3.2.4 Support vector machine

Support vector machines is an umbrella term for three different classifiers, the *maximal margin classifier*, the *support vector classifier*, and the *support vector machine*. Each of which is an extension of the former. All of these classifiers are based on *hyperplanes* to classify instances. A hyperplane is a flat subspace that in a p-dimensional space is represented in $p - 1$ dimensions, e.g. for a 2-dimensional space, a hyperplane is a line. These hyperplanes are used by the classifiers to separate instances of different classes. How these hyperplanes are used differ among the classifiers:

Maximal margin classifier

The maximal margin classifier (Vapnik and Lerner, 1963) is a simple and intuitive classifier that utilizes an *optimal separating hyperplane*, which is a separating hyperplane that is placed as far away from the training instances as possible, while still separating them into classes, i.e. maximizing the margin of the hyperplane to the training instances. The classifier can then classify instances based on which side of the hyperplane they lie. The training instances closest to the hyperplane are called *support vectors*, since they support the hyperplane in the sense that if one of them were moved slightly, the hyperplane would likely also have to be moved. This also implies that other instances that are not support vectors have no impact on the separating hyperplane. The fact that this classifier solely relies on the support vectors can lead to overfitting when operating in higher dimensions, it also renders the classifier unusable in the case when classes are not completely separated.

Support vector classifier

The support vector classifier (Cortes and Vapnik, 1995) reduces the impact that single instances can have upon the hyperplane, and simultaneously allowing the classifier to not have a perfectly separating hyperplane. This is done by introducing what is called a *soft margin*. Instead of trying to find the largest possible margin so that all instances is not only on the correct side of the hyperplane, but also on the correct side of the margin, the support vector classifier allows for some instances to be within the margin, or even on the wrong side of the hyperplane. By allowing a small reduction in training instance accuracy, a greater robustness to individual instances can be achieved. Due to that the support vector classifier is still a linear classifier, it can have issues when the instances have no linear boundary between classes.

Support vector machine

The support vector machine extends the support vector classifier by increasing the feature space using something called *kernels* (Boser et al., 1992). The purpose of this feature space expansion is to enable the classifier to implement a non-linear hyperplane. Without going into to much technical detail, the solution to finding the optimal hyperplane in a support vector classifier, only involves the inner products of the instances. The inner product of two instances x_i and x_j is given by

$$K(x_i, x_j) = \sum_{k=1}^{p} x_{ik} x_{jk} \tag{3.14}$$

where K is referred to as a kernel. Equation 3.14 is linear in the features, which is why the support vector classifier is linear. Equation 3.14 is also the same function

that is used by the *linear kernel* of the support vector machine. The linear kernel can be changed to

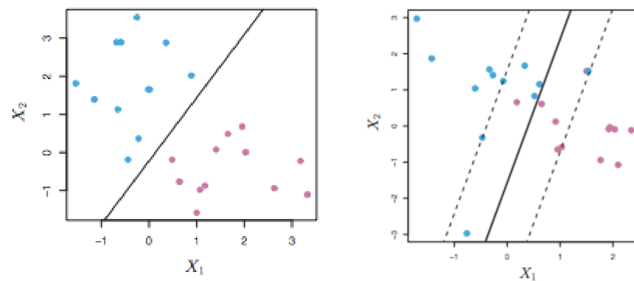
$$K(x_i, x_j) = (1 + \sum_{k=1}^{p} x_{ik}x_{jk})^d \tag{3.15}$$

which is what is called the *polynomial kernel*. By using the polynomial kernel with $d > 1$, the support vector machine can produce a more flexible boundary and therefore also classify instances that the linear kernel is not capable of. Another kernel is the *radial kernel* which is given by

$$K(x_i, x_j) = \exp(-\gamma \sum_{k=1}^{p} (x_{ik} - x_{jk})^2) \tag{3.16}$$

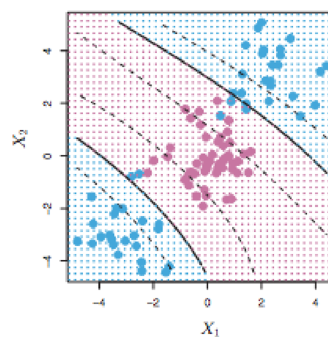
where γ is a positive constant. When the support vector classifier utilizes a non-linear kernel such as Equation 3.15 or Equation 3.16, the resulting classifier is the support vector machine.

Figure 3.6 shows an example of these algorithms hyperplanes.



(a) Maximum margin classifier with an optimal separating hyperplane.

(b) Support vector classifier with a soft margin hyperplane.



(c) Support vector machine using a polynomial kernel of degree 3.

Figure 3.6: Illustration of the three support vector machines (James et al., 2013).

Chapter 4

Evaluation Measures

This chapter describes the algorithms behind the measures used to produce the results in Chapter 5. By explaining both clustering and classification measures, it aims to provide an understanding to why these measures were used.

4.1 Clustering evaluation

The goal when clustering data is to receive a high intra-cluster similarity, while simultaneously receiving a low inter-cluster similarity, i.e. we want one cluster for each class in the dataset. For small datasets, this can easily be evaluated manually by simply looking at the contents of each cluster, but for larger datasets, this could become cumbersome. To enable the evaluation of larger datasets, we have selected a few external evaluation criteria, recommended by Manning et al. (2008), to ensure cluster quality: *purity*, *normalized mutual information*, *rand index*, and *F1 measure*.

4.1.1 Purity

To compute purity, each cluster is given a class based on which class that is most frequent in that cluster. Then the number of correctly assigned instances is counted for each cluster and this total number is divided by the total number of instances N. Formally:

$$\text{purity}(\Omega, \mathbb{C}) = \frac{1}{N} \sum_{k} \max_{j} |w_k \cap c_j| \tag{4.1}$$

Where $\Omega = w_1, w_2, ...$ is the set of clusters and $\mathbb{C} = c_1, c_2, ...$ is the set of classifications given to the aforementioned clusters.

4.1.2 Normalized mutual information

The problem with purity is that it improves with an increasing amount of clusters, if we have N clusters, then we will have the maximum purity. Normalized mutual information (NMI) handles this by penalizing an increasing numbers of clusters. The calculation of NMI is the following:

$$\mathrm{NMI}(\Omega, \mathbb{C}) = \frac{2 \cdot I(\Omega; \mathbb{C})}{[H(\Omega) + H(\mathbb{C})]}, \tag{4.2}$$

where I is the mutual information, and H is the entropy.

4.1.3 Rand index

Rand index (RI), introduced by Rand (1971), sees the problem as a series of decisions, one for each pair of instances. For each pair of instance, if these are similar and assigned to the same cluster, they are considered a true positive (TP). If these are dissimilar and are assigned to different clusters, they are considered a true negative (TN). Intuitively, if a pair of dissimilar instances are assigned to the same cluster, they are a false positive (FP), and if a pair of similar instances are assigned to different clusters, they are a false negative (FN). RI simply measures the accuracy of these decisions:

$$\mathrm{RI} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}. \tag{4.3}$$

4.1.4 F1 measure

RI sees both types of false decisions as equally wrong, even though false negatives might be worse. F1 measure (F1) penalizes false negatives and is evaluated using precision (P) and recall (R):

$$\mathrm{P} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \qquad \mathrm{R} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}, \qquad \mathrm{F1} = 2 \cdot \frac{\mathrm{P} \cdot \mathrm{R}}{\mathrm{P} + \mathrm{R}}. \tag{4.4}$$

4.2 Classification evaluation

When training a model, it is always better to use as much of the available data as possible. This however limits the testing to to be done on the same data. To learn the parameters of a predictive function and then test the function on the same data is methodologically a mistake. The model would remember the data and simply return the input label, producing a false evaluation score. In machine learning, this is called overfitting. If, on the other hand, the available data is split into one training and one testing set, the prediction will be unbiased, but the evaluation score might end up nowhere near the true model score.

A common practice to solve both of these problems is by using a procedure called cross validation (CV) (Mosteller and Tukey, 1968). The basic approach to CV, called k-fold CV, splits the available data into k smaller sets of equal size, trains the model on $k - 1$ sets, and tests it on the last set. This is repeated k times, until all k subsets have been used for testing, resulting in k evaluation scores.

To evaluate our model, we used a particular CV procedure, called LeaveOneOut-CV (LOO). LOO trains the model using all the instances except one i.e. for n instances it trains the model using $n-1$ instances, and tests the model on the last instance. This is equivalent to a k-fold CV where $k = n$. With this procedure, we were able to precisely extract the results from the model, used to fill out the confusion matrix described in Section 4.2.1.

In addition to evaluating models based on the accuracy of the predictions, we considered how the models differ in time consumption. Since we have a time requirement in our goals, both training and classification time is to be considered. To accomplish this, we performed 10 simple 5-fold CV and measured the average time passed. This allows us to evaluate complete model time, as well as accumulating deviating times.

4.2.1 Confusion matrix

We created a confusion matrix to evaluate how well the classifiers labeled the documents. A confusion matrix is a visual representation of the exact results achieved from at least one test, where it illustrates the actual classes of all the instances, and how they were predicted by the model. A generic example of a confusion matrix is the binary classification based matrix. Table 4.1 shows an example of this matrix. From this representation there are several measures that can be used to evaluate a classifier. Some examples of these measures are: *average accuracy*, *precision*, and *recall*. Table 4.2 shows the confusion matrix we used, that is based on a multi class classification.

	Predicted pos	**Predicted neg**
Actual pos	True positive	False negative
Actual neg	False positive	True negative

Table 4.1: Binary confusion matrix example

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	True SQL	False XSS	False XBD	False BEN
	XSS	False SQL	True XSS	False XBD	False BEN
	XBD	False SQL	False XSS	True XBD	False BEN
	BEN	False SQL	False XSS	False XBD	True BEN

Table 4.2: Multi class confusion matrix example

Average accuracy

The average accuracy (ACC) of a classifier is the simplest and most intuitive way to measure its effectiveness. It calculates the average correctness of the classifications by dividing the correct predictions with all the predictions.

$$\text{ACC} = \frac{\sum_{i=1}^{l} \frac{tc_i}{tc_i+fc_i}}{l} \tag{4.5}$$

where l is the total number of classes, tc are the correct classifications, and fc are the sum of the incorrect classifications.

Precision

Precision, or Positive Predictive Value (PPV), is a measure of per-class effectiveness. From all the predictions that have been made as the same class, it calculates the ratio of instances that have been classified correctly.

$$\text{PPV} = \frac{tc_i}{c_i} = \frac{tc_i}{tc_i + \sum fc_i} \tag{4.6}$$

where tc_i are all the correct classifications of class i, c_i are all classification classified as class i, and fc_i are all classifications incorrectly classified as class i.

Recall

Recall, or True Positive Rate (TPR), is another way to measure per-class effectiveness. From all the items that belong to a class, it calculates the ratio of instances that have been classified correctly.

$$\text{TPR} = \frac{tc_i}{I_i} = \frac{tc_i}{tc_i + \sum I_{fc_i}} \tag{4.7}$$

where tc_i are all the correct classifications of class i, I_i are all the instances that belong to class i, and I_{fc_i} are all the instances of class i that have been classified incorrectly.

4.2.2 Learning curve

A learning curve (Ebbinghaus, 1913) is a graph that illustrates a measure of performance over a varying amount of learning effort. The term *learning curve* is used in two different contexts within machine learning, one concerning neural networks, and one that is more generally adaptive. We used the general approach to learning curve, where the graph illustrates the predictive generalization performance as a function of the number of training examples used. Figure 4.1 shows an example of a learning curve.

We began by dividing the dataset into two sets of equal size, one training set, and one testing set. During each iteration, we increased the dataset size used to train the classifier, trained several models using different parts of the training set, and used these models to predict the instances of the testing set. The average ratio of correct predictions was the performance measure for that specific dataset size. By using a training size over a range from 1% to 50% of the complete dataset, this created a curve that illustrates the learning performance of the used classifier.

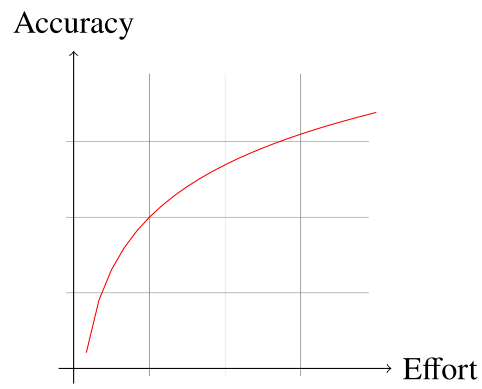


Figure 4.1: A learning curve illustrating the increase in performance while the effort increases.

Chapter 5
Results

This chapter presents the results of the clustering algorithms as well as the classification algorithms explained in Chapter 3. The results are gathered using the measures described in Chapter 4.

5.1 Clustering results

During the clustering analysis, we explored two different clustering algorithms: one that requires the desired numbers of clusters beforehand (k-means), and one that does not (DBSCAN). Tables 5.1-5.2 shows the results of these algorithms. Note that we consider the data to consist of four different groups.

As we can see from the results, the DBSCAN algorithm performs best with an epsilon $\epsilon = 0.4$. Since all our measurements agree on this, this is our optimal DBSCAN clustering. However, the resulting numbers of clusters is 9, which is more than double the actual number.

The k-means algorithm performs best with 6 clusters, which is closer to our interpretation. This clustering also receives higher evaluation scores than the best DBSCAN clustering, on all measures except F1 measure.

Table 5.1: Results of the clustering analysis, using DBSCAN ϵ between 0.2 – 1.0.

ϵ	Purity	NMI	RI	F1	Number of clusters
0.2	0.8815	0.7262	0.7636	0.8601	10
0.4	**0.9381**	**0.8077**	**0.8617**	**0.9326**	9
0.6	0.8235	0.701	0.6904	0.7627	5
0.8	0.834	0.7317	0.7359	0.7705	4
1.0	0.5	0.0	0.0	0.3333	1

Table 5.2: Results of the clustering analysis, using k-means with k between $4 - 8$.

k	Purity	NMI	RI	F1
4	0.9328	0.8086	0.8695	0.9144
6	**0.9452**	**0.8462**	**0.9105**	**0.9269**
8	0.9448	0.845	0.9093	0.9265

5.2 Classification results

During the classification analysis, we explored four different classification algorithms, and three ways of representing features. Based on the evaluation measures described in Section 4.2, we have concluded a number of results.

Figures 5.1-5.3 show the learning curves. As can be seen, the learning curves behave as expected. With little data, we are still able to achieve a relatively high accuracy. The models using keyword feature representation learn quicker compared to models using n-gram feature representation.

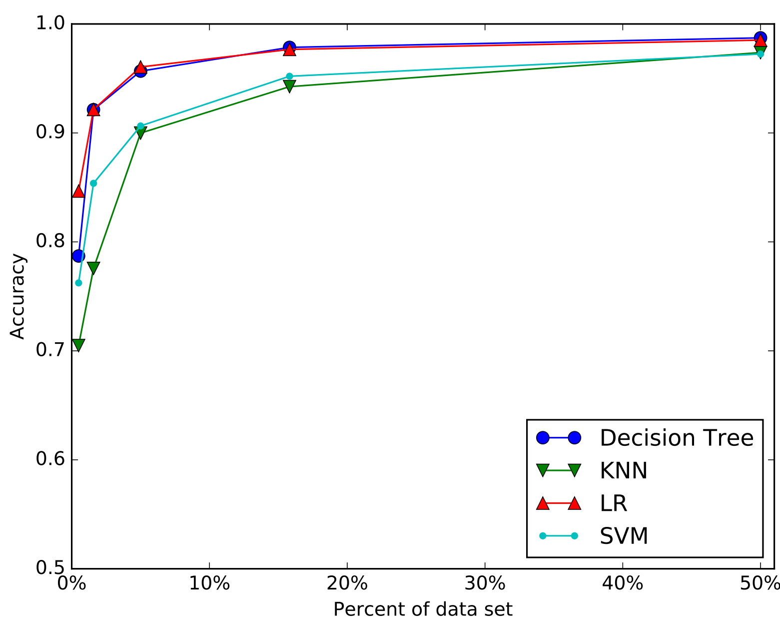


Figure 5.1: Learning curve of the different algorithms using a 1-gram representation of the parameters.

Table 5.3 shows the results of the time measurements. As we can see, the decision tree model is the fastest, independent of feature representation. Furthermore, all models are faster using the keyword feature representation, except for k nearest neighbors where 1-gram representation is faster.

Table 5.4 shows the average accuracy of all models and feature representations. The worst result is logistic regression using 1-grams, which achieved an accuracy of 97.1%. KNN using keyword feature representation achieved the best result with an accuracy of

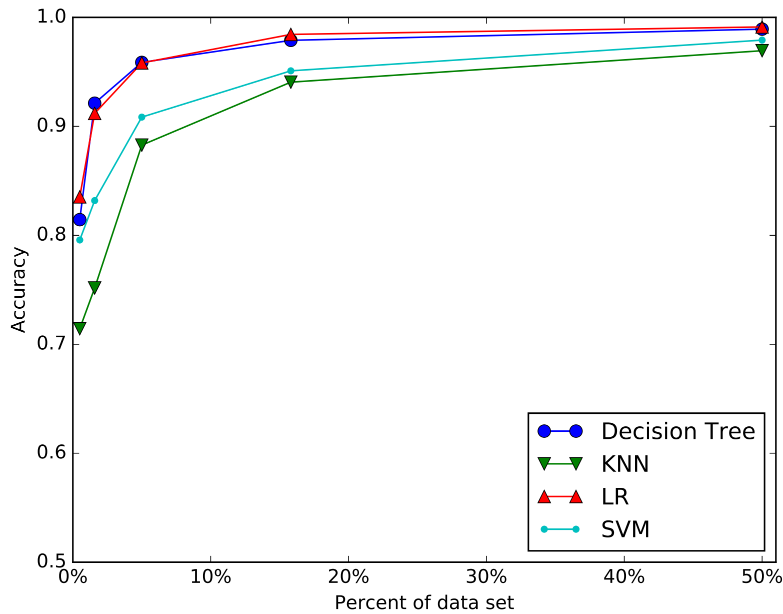


Figure 5.2: Learning curve of the different algorithms using a 1-2-gram representation of the parameters.

Table 5.3: Time (s) measurements for all models and feature representations.

Features	DTC	KNN	LR	SVM
1-gram	0.1917	1.9981	2.8634	1.6003
1-2-gram	5.7893	65.9405	10.9168	62.0825
keyword	**0.1034**	2.6094	0.6845	0.6690

99.51%. Furthermore, all the models achieved their highest accuracy using the keyword feature representation. Tables 5.5-5.8 shows the details of the keyword model evaluation. Whereas, the n-gram model details can be found in Appendix B.

Table 5.4: Accuracy (%) for all models and feature representations.

Features	DTC	KNN	LR	SVM
1-gram	99.08	98.57	97.10	98.26
1-2-gram	99.16	98.31	98.91	98.64
keyword	99.48	**99.51**	99.01	99.24

Figure 5.4 shows a partial decision tree generated by the DTC model. The tree is a flowchart-like structure and the starting node is the top one. All non-leaf nodes start with a feature condition. If this condition holds true, the left node is chosen as the next node.

This is illustrated in Figure 5.4 where the start node has the feature condition `' <= 0.5`. This means that if there is less than 1 occurence of the single-quote in our instance, the next decision node is the left one.

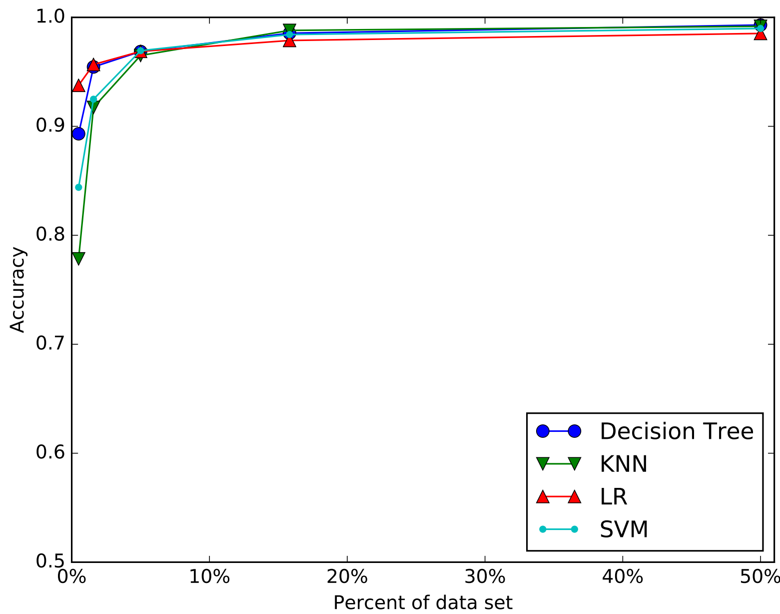


Figure 5.3: Learning curve of the different algorithms using our own keyword representation of the parameters.

Table 5.5: Confusion matrix and the resulting evaluation measures, using the decision tree model with keyword features.

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	2051	3	0	1
	XSS	4	190	9	1
	XBD	0	6	715	0
	BEN	6	1	0	2973

	SQL	**XSS**	**XBD**	**BEN**
Precision	99.51	95.00	98.76	99.93
Recall	99.81	93.14	99.17	99.77
	Accuracy: 99.48%			

Additionally, each node in the diagram contains other information such as the impurity (Gini), number of samples (samples), the class distribution in the node (values), and the majority class (class).

Table 5.9 summarizes the properties of trees generated by the DTC model. 1-grams results in the largest trees, with the highest average depth and node counts – while the keyword representation has the highest maximum depth. 1-2 grams has the lowest number of nodes, lowest average depth and lowest maximum depth.

Table 5.6: Confusion matrix and the resulting evaluation measures, using the k nearest neighbors model with keyword features.

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	2052	2	0	1
	XSS	3	190	10	1
	XBD	2	4	715	0
	BEN	5	1	0	2974

	SQL	**XSS**	**XBD**	**BEN**
Precision	99.52	96.45	98.62	99.93
Recall	99.85	93.14	99.17	99.80
	Accuracy: **99.51%**			

Table 5.7: Confusion matrix and the resulting evaluation measures, using the logistic regression model with keyword features.

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	2037	2	0	16
	XSS	7	175	9	13
	XBD	2	3	712	4
	BEN	3	0	0	2977

	SQL	**XSS**	**XBD**	**BEN**
Precision	99.41	97.22	98.75	98.90
Recall	99.12	85.78	98.75	99.90
	Accuracy: 99.01%			

Table 5.8: Confusion matrix and the resulting evaluation measures, using the support vector machine model with keyword features.

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	2042	5	0	8
	XSS	6	184	10	4
	XBD	0	6	715	0
	BEN	5	1	0	2974

	SQL	**XSS**	**XBD**	**BEN**
Precision	99.46	93.88	98.62	99.60
Recall	99.37	90.20	99.17	99.80
	Accuracy: 99.24%			

Table 5.9: Comparison of the properties of the decision trees generated by different feature collection methods.

Features	Nodes	Average Depth	Max Depth
1-grams	163	7.3	21
1-2-grams	**105**	**6.7**	**15**
Keyword	123	6.9	22

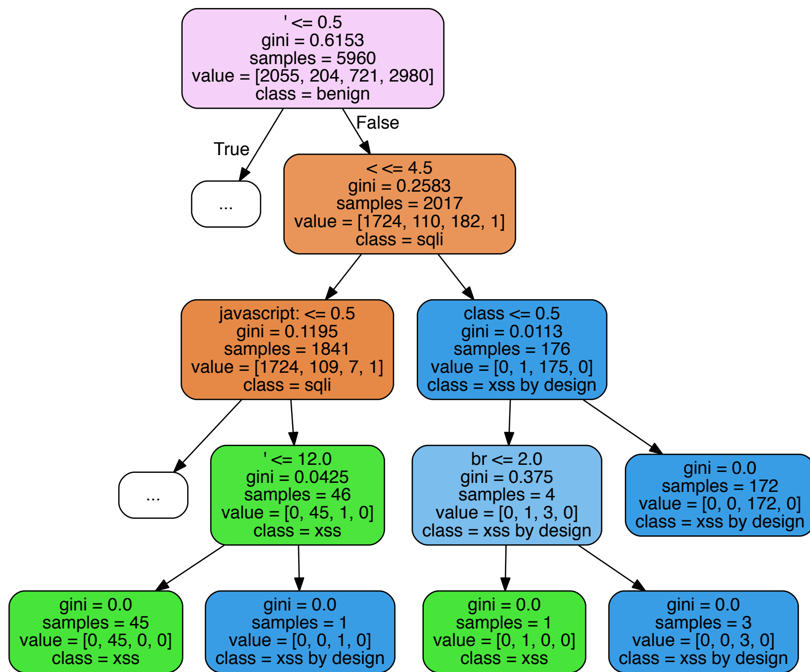


Figure 5.4: Reduced decision tree generated by the DTC-model. The original tree can be found in Appendix A and has 123 nodes with a maximum depth of 10 and the average depth is 6.9 nodes.

Chapter 6

Discussion

This chapter evaluates the results presented in Chapter 5, and discusses their validity and relevance. Furthermore, it discusses possible applications of this project, and compares the result to the set goals.

6.1 Results

Given that we already knew that we had four distinct classes in our dataset, the clustering analysis felt somewhat unnecessary. We used this part of the process largely to learn more about our data. The clustering helped us to confirm that the data set could be split into four groups. We know from the results that XSS and XBD are separable. Moreover, Table 5.2 shows that for $k = 4$ we are given reasonable results. Furthermore, the results are not much lower than for $k = 6$, which is the best clustering result.

Unfortunately, this test only gives an overall performance measure. It does not provide detailed information such as individual performance per class, nor does it provide an indication of false positives or negatives. However, we still consider this approach to be valid if little is known about the data.

In order to fully perceive our results, we used n-grams as a baseline. N-grams were chosen since this is an explored approach which requires little to no feature engineering. Additionally, n-grams can be applied to a wide variety of unspecific text based problems.

All learning curves behaved in the way we expected. The lowest accuracy was around 70% using about 1% the dataset. The learning curve for the keyword representation was the quickest to plane out. This shows that the keyword representation is a good approach for our problem. This does not imply that the n-gram approach was bad, just that the keyword representation learns quicker. What we learn from these curves is that these approaches are useful even when the available data is limited.

If we look at the n-gram representation time measurements, it is clear that the number of features has a direct impact on the performance. However, our keyword representation

has more features than 1-gram, but is still faster for all models but KNN. This implies that the time performance is not only related to the number of features, but also to the number of relevant features. This test evaluated both training and classification performance for a great number of instances. As such, practically doing a single classification, with a trained model, would take less time.

The accuracies in Table 5.3 show that keyword features is an approach in the right direction. Even though it is marginally better than the baseline accuracies, we consider the marginal improvement important. Since small changes in accuracy can have a great effect on usefulness, any increase in accuracy is highly valued.

Accuracy can be misleading, since some misclassifications are worse than others. A false positive, for example, classifying a BEN as a XSS is more acceptable than a false negative. This is better demonstrated in the confusion matrices and its accompanying precision/recall tables. In this case for XBD and BEN, we strive to achieve high precision, and for SQL and XSS, we strive to achieve high recall, since these values indicate that a malicious parameter has been classified as benign. More specifically, it appears that XSS is our weak link in regard to recall. DTC and KNN perform best in regard to recall with both reaching a value of 93.14%. This could be caused by a lack of XSS samples, especially in the balance between XSS and XBD. All the models seem to have a difficulty to distinguish XSS from XBD. This is most likely due to that the classes are textually similar.

In terms of decision trees, 1-2-grams seem to generate a smaller tree even though it has a greater number of features. This is likely due to that 1-2-grams have more discriminating features, and thus the tree can easier select its decision conditions. It seems that tree size and time performance are not directly correlated. Time performance seems to be affected by both number of features and discriminating features. Practically, if the decision tree is already constructed, the performance of this tree is more related to its size. Additionally, only features which are used as conditions in the tree would need to be collected in the feature representation.

6.2 Possible applications

In this section, we explore how the models could be applied in the Triggerfish Platform. Each model that we have evaluated has different properties, not only in the sense of the evaluated metrics such as accuracy and time, but also in how the models are fitted and used. Moreover, each model can present the results in a different way. For example, a DTC model can produce a decision tree while an LR model presents the result of its classification as a list of classes and their probabilities. Additionally, the different models have a different resource impact, in both memory and computing time, which is also an aspect to be discussed in this section.

6.2.1 Clients

As mentioned in the introduction, the clients should be light on performance in both time and disk space used and therefore a machine learning model used in the client must be lightweight. As a suggestion to reduce redundant calculations in all clients, a fitted model should be supplied in the client. This also reduces the disk space used by a model since

a fitted model is smaller than the training data used to create the model. If needed, the procedures for automatically updating the model should be implemented in the client. Consequently, a training module should be developed for the backend, which can update the client with new models.

As can be seen in the time measurement test in Table 5.3, the DTC-model is the fastest model to complete its cross-validation. Furthermore, to use the DTC-model as a classifier one only needs to have the actual decision tree, which, in turn, does not necessarily use all features which are collected by the feature-vectorizer. This makes the model lightweight in disk space. Moreover, as we can see in Table 5.5, the DTC model has good scores when it comes to classification accuracy.

6.2.2 Backend

In the backend, resources are less constrained, therefore a slower model is acceptable. Additionally, disk space is not constrained here – thus the whole training set can be stored. The LR model could be used in the backend for its probability measures. KNN is another possibility, since it scored the highest accuracy in our tests – it does however not provide a real confidence measure in the form of a probability.

The frontend should display the probability measures as a confidence level for a classification. Consequently, the analysis server would have to have a LR or KNN classification filtor to do the actual classification for injection attacks. Furthermore, the frontend could be used to collect feedback from the users. This feedback would be used as additional training data.

For training models, a backend module should be developed. This is done in order to not change the current modular design of the backend, and to not increase complexity in other modules. The responsibilities of this module would be limited to: training a model for the analysis server, training a model for the clients, and serving updates to the client models.

6.3 Achievements

The purpose of this thesis was to explore the possibility to adapt machine learning theory to code injection attempts, more specifically, to the Triggerfish platform. To do this, we established interim goals and in this section we explain how these goals have been accomplished.

Reach an *acceptable* accuracy In order to answer whether this goal was reached, we chose appropriate evaluation measures. Additionally, we established a baseline using two n-gram representations. We attained a maximum accuracy of 99.51% using the keyword representation. This was higher than the baseline, which implies that this goal has been reached. Furthermore, all models are able to distinguish between the XSS and XBD classes.

Account for model performance and size requirements Our results from the time measurements (Table 5.3) show the performance of our models. Furthermore, the results are discussed in Section 6.1 and it is clear that the keyword representation is faster

than the baseline. Additionally, Section 6.2 discusses the size requirements of the different models.

Describe how these models can be implemented into Triggerfish Section 6.2 discusses how the models can be implemented into the existing Triggerfish architecture. Furthermore, specific examples of models are motivated based on their performance and characteristics.

Find a model that provides a second opinion and a degree of confidence Out of all our evaluated models we found that the LR model provides a real estimate of probability for each class. All the models could function as a second opinion to the Triggerfish platform. Furthermore, both SVM and KNN provide measures which could be seen as degrees of confidence – but are not pure probability measures.

Chapter 7
Conclusions

This chapter summarizes the content of this report, and provides suggestions to future work and improvements that can be applied to the project.

7.1 Summary

In this thesis, we have explored the possibility to classify HTTP requests using machine learning techniques. More specifically, we aimed not only to distinguish between benign and malign requests – but also to separate three different code injection classes from each other. Using a standard data mining process, we constructed evaluation measures both to review several classification algorithms, and also three different data representation methods. Finally we succeeded in finding the intentions of HTTP requests by identifying language specific syntax. This lead to the final data representation using language specific keywords, which achieved an accuracy of 99.51% using the k-nearest-neighbors algorithm. Additionally, we have suggested ways of integrating the findings into the Triggerfish platform.

7.2 Future improvements

Even though we consider this thesis project to be a success, there are still a few areas which could be improved. This section discusses some ways that could improve the results of this project, and also ways to further increase the applicability of our results.

7.2.1 Data set improvement

As we have previously mentioned, the data that we used is slightly imbalanced. More specifically, the number of XSS is comparatively low. This might be the reason for the

relatively low recall for the XSS class. If the classes could be balanced, as well as the total cardinality increased, the results may improve.

7.2.2 Feature engineering

Naturally, further manual feature engineering could be done to improve both accuracy and performance of the model, however an automated approach to feature engineering could be attempted. Minimum Redundancy Maximum Relevance (mRMR) (Peng et al., 2005) is a feature selection algorithm to find the optimal features. Given an annotated dataset, it eliminates redundant features and extracts the most discriminating ones. The algorithm could, for example, be used in conjunction with a n-gram range, and reduce the n-gram features to a predetermined number of discriminating features. This approach could potentially remove the need for future manual feature engineering – but would increase the time and memory requirements of the data preparation phase.

7.2.3 Further hyperparameter optimization

Most classification models provided by `scikit-learn` provide several hyperparameters which determine the behavioral characteristics of the model. The existing evaluation framework could be used in conjunction with a grid search to find better hyperparameters.

7.2.4 Adding more classes

The Triggerfish platform is not limited to detecting code injection attacks. As such, more classes could be added to the model. This could for example be detecting remote code execution such as the shellshock attack, or directory traversals. For some cases this would require further feature engineering to deduce new keyword features.

7.2.5 Hybrid trees

The satisfying performance and accuracy of the decision tree makes this a suitable classifier. It could however be possible to improve these results if a decision tree collaborated with a binary classifier, such as SVM. Instead of having a true or false condition in a node, it could contain a SVM classifier with data relevant to this node.

Bibliography

Arthur, D. and Vassilvitskii, S. (2007). K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Berkson, J. (1944). Application of the logistic function to bio-assay. *Journal of the American Statistical Association*, 39(227):357–365.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. *COLT '92 Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152.

Breiman, L., Friedman, J., Stone, C. J., and Olshen, R. A. (1984). *Classification and Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series. Taylor & Francis.

Chapman, P., Clinton, J., Kerber, R., Khabaza, T., Reinartz, T., Shearer, C., and Wirth, R. (2000). *CRISP-DM 1.0, Step-by-step data mining guide*. SPSS Inc.

Cheon, E. H., Huang, Z., and Lee, Y. S. (2013). Preventing sql injection attack based on machine learning. In *IJACT: International Journal of Advancements in Computing Technology*, volume 5 of *9*, pages 967–974.

Chire (2011). Dbscan-illustration. `https://commons.wikimedia.org/wiki/File:DBSCAN-Illustration.svg`. Accessed: 2016-03-01.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3):273–297.

Digifort Sverige AB (2015). Documentation for the Triggerfish Rails client.

Ebbinghaus, H. (1913). *Memory: A Contribution to Experimental Psychology*. Columbia University. Teachers College. Educational reprints. no. 3. University Microfilms.

Ester, M., Kriegel, H., Sander, J., and Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext transfer protocol – http/1.1.

Hamerly, G. and Elkan, C. (2002). Alternatives to the k-means algorithm that find better clusterings. *Proceedings of the eleventh international conference on Information and knowledge management*, pages 603–604.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning with Applications in R*. Springer.

Lloyd, S. (1982). Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

Mosteller, F. and Tukey, J. W. (1968). *Data analysis, including statistics*. Addison-Wesley, Reading, MA.

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

Peng, H., Long, F., and Ding, C. (2005). Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(8):1226–1238.

Rand, W. M. (1971). Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850.

scikit learn (2014). Demo of dbscan clustering algorithm. `http://scikit-learn.org/stable/auto_examples/cluster/plot_dbscan.html`. Accessed: 2016-03-01.

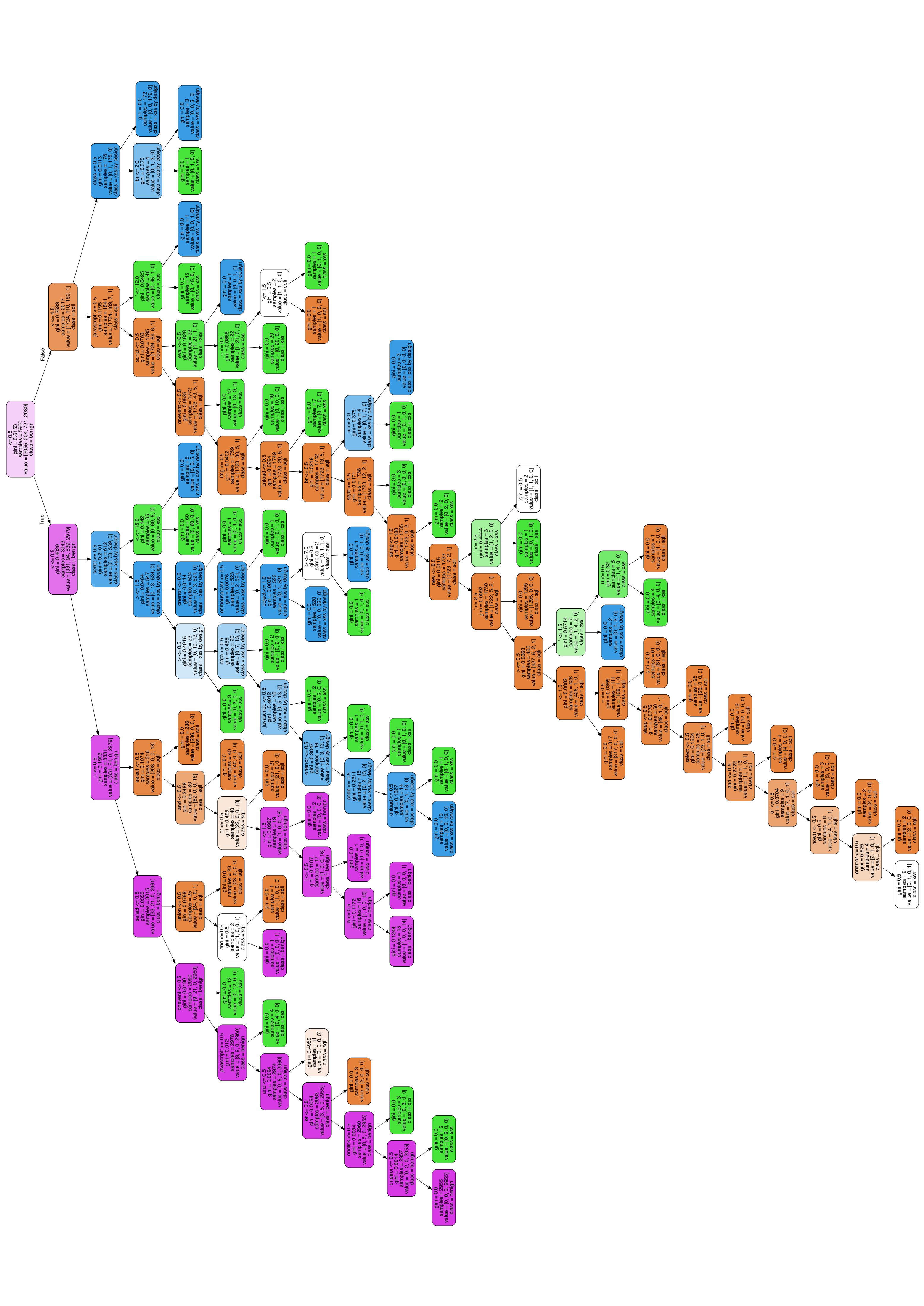
Vapnik, V. and Lerner, A. (1963). Pattern recognition using generalized portrait method. *Automation and Remote Control*, 24:774–780.

Weston.pace (2007). K means example step 1, 2, 3 and 4. `https://commons.wikimedia.org/wiki/File:K_Means_Example_Step_1.svg`, `https://commons.wikimedia.org/wiki/File:K_Means_Example_Step_2.svg`, `https://commons.wikimedia.org/wiki/File:K_Means_Example_Step_3.svg`, `https://commons.wikimedia.org/wiki/File:K_Means_Example_Step_4.svg`. Accessed: 2016-02-29.

Wressnegger, C., Schwenk, G., Arp, D., and Rieck, K. (2013). A close look on n-grams in intrusion detection: Anomaly detection vs. classification. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, pages 67–76, New York, NY, USA. ACM.

Appendices

Appendix A
Final Decision Tree



Appendix B

Results

Table B.1: Confusion matrix and the resulting evaluation measures, using the decsion tree model with 1-gram features.

(a)

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	2041	11	0	3
	XSS	9	187	6	2
	XBD	0	8	713	0
	BEN	9	7	0	2964

(b)

	SQL	**XSS**	**XBD**	**BEN**
Precision	99.13	87.79	99.17	99.83
Recall	99.32	91.67	98.89	99.46
	Accuracy: 99.08%			

Table B.2: Confusion matrix and the resulting evaluation measures, using the k nearest neighbors model with 1-gram features.

(a)

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	2018	3	0	34
	XSS	4	183	1	16
	XBD	1	8	697	15
	BEN	1	1	1	2977

(b)

	SQL	**XSS**	**XBD**	**BEN**
Precision	99.70	93.85	99.71	97.86
Recall	98.20	89.71	96.67	99.90
	Accuracy: 98.57%			

Table B.3: Confusion matrix and the resulting evaluation measures, using the linear regression model with 1-gram features.

(a)

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	2009	3	5	38
	XSS	12	155	12	25
	XBD	2	7	695	17
	BEN	11	6	35	2928

(b)

	SQL	**XSS**	**XBD**	**BEN**
Precision	98.77	90.64	93.04	97.34
Recall	97.76	75.98	96.39	98.26
	Accuracy: 97.10%			

Table B.4: Confusion matrix and the resulting evaluation measures, using the support vector machine model with 1-gram features.

(a)

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	2033	5	3	14
	XSS	7	186	6	5
	XBD	4	9	702	6
	BEN	22	15	8	2935

(b)

	SQL	**XSS**	**XBD**	**BEN**
Precision	98.40	86.51	97.64	99.16
Recall	98.93	97.36	98.49	
	Accuracy: 98.26%			

Table B.5: Confusion matrix and the resulting evaluation measures, using the decision tree model with 1-2 grams features.

(a)

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	2045	4	0	5
	XSS	11	184	8	1
	XBD	2	8	711	0
	BEN	7	3	0	2970

(b)

	SQL	**XSS**	**XBD**	**BEN**
Precision	99.03	92.00	98.89	99.80
Recall	99.51	90.20	98.61	99.66
	Accuracy: 99.16%			

Table B.6: Confusion matrix and the resulting evaluation measures, using the k nearest neighbors model with 1-2 grams features.

(a)

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	1997	2	0	56
	XSS	3	188	1	12
	XBD	2	4	697	18
	BEN	2	1	0	2977

(b)

	SQL	**XSS**	**XBD**	**BEN**
Precision	99.65	96.41	99.86	97.19
Recall	97.18	92.16	96.67	99.90
	Accuracy: 98.31%			

Table B.7: Confusion matrix and the resulting evaluation measures, using the logistic regression model with 1-2 grams features.

(a)

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	2038	1	0	16
	XSS	4	191	3	6
	XBD	2	1	701	17
	BEN	2	1	11	2966

(b)

	SQL	**XSS**	**XBD**	**BEN**
Precision	99.61	98.45	98.04	98.70
Recall	99.17	93.63	97.23	99.53
	Accuracy: 98.93%			

Table B.8: Confusion matrix and the resulting evaluation measures, using the support vector machine model with 1-2 grams features.

(a)

		Predicted			
		SQL	**XSS**	**XBD**	**BEN**
Actual	**SQL**	2031	9	3	12
	XSS	1	189	6	8
	XBD	1	6	706	8
	BEN	15	7	5	2953

(b)

	SQL	**XSS**	**XBD**	**BEN**
Precision	99.17	89.57	98.06	99.06
Recall	98.83	92.65	97.92	99.09
	Accuracy: 98.64%			

EXAMENSARBETE Intrusion Detection System by Statistical Learning – Using keyword features to classify HTTP requests
STUDENTER Meris Bahtijaragic, Julian Kroné
HANDLEDARE Pierre Nugues (LTH), Markus Millbourn (Digifort Sverige AB)
EXAMINATOR Jacek Malec (LTH)

Webbsäkerhet genom statistiskt lärande

POPULÄRVETENSKAPLIG SAMMANFATTNING **Meris Bahtijaragic, Julian Kroné**

I takt med att tillgängligheten ökar på nätet så ökar även behovet av smidiga säkerhetslösningar. Detta examensarbete utforskar möjligheten att tillämpa statistiskt lärande för att upptäcka intrångsförsök.

Ett webbserverintrång är när en användare får obehörig tillgång till information. För att uppnå detta används vanligtvis kodinjektioner. En kodinjektion är när en attackerare får en server till att köra kod som den ej är avsedd till att köra. Med en lyckad kodinjektion kan en attackerare stjäla, manipulera, eller förstöra lagrad information på en server.

I dag finns det över 3 miljarder internetanvändare världen över. Mycket av deras användningsstatistik lagras, men istället för att använda denna data till att lära upp och förbättra säkerhetssystem så används istället manuellt framställda regler. System som upptäcker intrångsförsök kallas för Intrusion Detection Systems (IDS). Dessa system analyserar data som skickas från användare för att identifiera intrångsförsök. I vårt arbete används data från ett befintligt IDS för att lära en maskin att se mönster. Denna maskin kan sedan användas för att upptäcka webbserverintrång.

Precis som vid ett läkarbesök nöjer man sig inte enbart med att få veta om man är frisk eller sjuk, utan man vill också veta vad är det för sjukdom och hur den kan behandlas. På samma sätt är det vid ett intrångsförsök på en webbserver. Det är inte tillräckligt att enbart få veta om det har skett, utan man vill även ha mer information. Vad är det för typ av intrångsförsök, hur det påverkar mig, och hur stor är sannolikheten att det är ett faktiskt intrångsförsök och inte ett så kallat falskt larm.

De mönsterbaserade IDS som finns på marknaden idag identifierar ofta attacker genom metoder som enbart kan visa att ett intrångsförsök har skett eller ej. Ett problem för dagens mönsterbaserade IDS är att det förekommer nätverkskommunikation som ser elakartad ut, men i själva verket är godartad. Detta ger upphov till falska larm. I detta examensarbete utforskas datadrivna metoder som reducerar denna typen av falska larm.

Digifort är ett Lundabolag vars främsta tjänst är ett IDS som heter Triggerfish som upptäcker intrångsförsök riktade specifikt mot webbapplikationer. I detta examensarbete utvärderas hur våra framtagna algoritmer kan tillämpas i Digiforts plattform Triggerfish. Detta görs eftersom maskinerna som testats har olika egenskaper. Exempelvis har en sorts maskin förmågan att förmedla hur säker den är på sitt svar, en annan maskin har med hjälp av ett beslutsträd möjlighet till snabb återkoppling.

Med hjälp av Triggerfish har exempel på kodinjektioner samlats in, dessa kategoriserades till en början manuellt beroende på dess typ. Den kategoriserade datan används sedan för att lära upp våra maskiner. Eftersom mängden data bidrar till en maskins förmåga att kunna kategorisera händelser så användes dessa maskiner sedan till att kategorisera ny data och utöka datamängden. Genom detta kom vi slutligen fram till ett antal rimliga metoder för att känna igen och kategorisera injektionsattacker, varav den bästa med 99.51% precision.