Chalmers University of Technology

TDA602 - Group 30

Language-Based Security

Machine Learning based Web Application Firewall

Authors:
Filip Granqvist
Oskar Holmberg

Supervisor:
Andrei Sabelfeld

August 8, 2017

Contents

1 Introduction **1**
1.1 Purpose . . . 1
1.2 Delimitations . . . 1

2 Technical background **1**
2.1 Intrusion detection and prevention systems . . . 2
2.2 Web application firewalls . . . 2
2.3 Machine learning . . . 2
2.3.1 Classifiers . . . 2
2.3.1.1 Logistic regression classifier . . . 3
2.3.1.2 Support vector machine . . . 4
2.3.1.3 Multinomial Naive Bayes . . . 5
2.3.1.4 Random forest . . . 6
2.3.1.5 AdaBoost . . . 6
2.3.1.6 Artificial neural network . . . 6
2.3.2 Bag-of-words . . . 7
2.3.3 Principle component analysis . . . 7
2.3.4 Training, validation and test data . . . 8
2.3.5 Performance metrics . . . 9

3 Method **10**
3.1 Preparatory work . . . 10
3.2 Workflow . . . 11
3.3 Programming language . . . 11
3.4 Documentation . . . 12
3.5 Version control . . . 12

4 Implementation **12**
4.1 Data wrangling . . . 13
4.2 Data cleaning . . . 13
4.3 Feature engineering . . . 13
4.3.1 Bag of words feature space . . . 14
4.3.2 Custom feature space . . . 15
4.3.3 Length . . . 16
4.3.4 Non-printable characters . . . 17
4.3.5 Punctuation characters . . . 18
4.3.6 Minimum byte . . . 18
4.3.7 Maximum byte . . . 19
4.3.8 Mean byte . . . 20
4.3.9 Standard deviation byte . . . 20
4.3.10 Distinct bytes . . . 21
4.3.11 SQL keywords . . . 22

4.3.12 Javascript keywords 22
4.4 Model selection 23
4.5 Demonstration website 24

5 Results **25**
5.1 Classifiers 25
5.2 Feature spaces 30

6 Discussion **31**
6.1 Developer tools 31
6.2 Deficiencies 32
6.3 Future work 33

7 Conclusion **33**

References **35**

Appendices **I**

A.1 Github repository **I**

A.2 Complete table of results **I**

A.3 Contributions **III**

1 Introduction

In the recent couple of years there has been a significant increase of application layer based attacks [1]. According to the Open Web Application Security Project attacks such as Cross site scripting(XSS) and SQL injections are the kind of attacks applications are usually most susceptible to [2]. There are several measures to take into account to mitigate these kind of attacks. Sanitizers and Content Security Policies are examples of these protection mechanisms. Another way to mitigate application layer attacks, which this report will focus on, is Web Application Firewalls(WAF) with intrusion detection. With the rapid growth of data, machine learning has been starting to play a significant role within application firewalls.

Intrusion detection systems (IDS) are typically run under some predefined rules, and could thus be circumvented by an attacker. To make an intrusion detection more dynamic and prone to detect new threats as well machine learning can be applied for this task. To make high probability predictions of whether the incoming traffic is malicious or not, just looking at the packet-level isn't always sufficient. In this project we will therefor look directly into the payload on an application level, extracting features found valuable.

1.1 Purpose

The purpose of this project is to investigate the potential of using machine learning techniques for intrusion detection on an application level. The algorithms to be tested are classifiers trained on HTTP payload data which will take a payload as input and decide whether the input contained malicious code or not.

A subproblem to be solved during the project is feature engineering highly relevant features for malicious injections and XSS. In other words, we want to investigate what are good and bad features to use in machine learning classifiers when trying to detect malicious code.

1.2 Delimitations

The most critical web application security risks ranked by OWASP are injections and cross-site scripting. These malicious attacks are also the types which offers the most available data on the web, so we will restrict our classifiers to be trained on XSS-, SQL- and Shell-injections.

Another restriction made to simplify our implementation is that we only consider the address field for GET HTTP requests, and the content field for POST HTTP requests. This means that injections placed in other fields, such as HTTP header injections, will go undetected. [3]

2 Technical background

In this section the different techniques to realize the project will be explained. It will also go through the general principles of what defines a Web Application Firewall and Intrusion Detection Systems.

2.1 Intrusion detection and prevention systems

Historically there have been two major types of intrusion detection systems(IDS) worth mentioning, signature/misuse-based detection and anomaly-based detection. The former commonly uses blacklisting to define malicious inputs, and let everything else pass without an alarm, i.e. default permit. A problem with this technique of identifying attacks is that new threats(zero-day vulnerabilities) will go unnoticed and pass the IDS as normal traffic. The latter technique, anomaly-based detection, does quite the opposite - it defines what is ordinary network traffic and gives alarm for everything else, i.e. default deny.

Intrusion detection systems are usually just built to monitor traffic that flows through it. When they detect anomalies, they can either log the data or create an alert. Some of those IDSs are however extended to also include prevention, i.e. intrusion prevention. This means that when potentially malicious traffic flows through the system, they can take actions and react to the packet [4]. A shortcoming for those general type of systems is that they can't understand packets at an application level.

2.2 Web application firewalls

As there recently has been a substantial increase of attacks on the application level(OSI level 7), the need for intrusion detection to understand application data has raised. In addition to the intrusions described in section 2.1, a Web application Firewall has the ability to investigate the layer 7 of a packet. This gives a more sophisticated system for detecting threats such as SQL injection, cross-site scripting and buffer overflows.

2.3 Machine learning

Machine learning is a subfield of computer science, and more specifically, a subfield of artificial intelligence. The subject in question is all about building models that learn from data and make predictions without being explicitly programmed. The underlying theories are statistical techniques that can fit the data it is presented to, and then guess the most reasonable output for a new unseen input data point. The technique of presenting data input along with their true labels while training the model is called supervised learning, which is the technique heavily used in this project.

2.3.1 Classifiers

Classifiers are supervised machine learning algorithms specialized in assigning labels to data, i.e. classifying input data into different categories.

The input data from the data sample usually isn't in its original format when presented to the classifier. Instead, different meaningful characteristics are extracted from the data sample and made into a vector. The vector of extracted characteristics is equivalent to the name feature vector, which is commonly used in machine learning terminology.

An example classifier using three different feature inputs is shown in figure 1. Instead of using every pixel of the flower image as input, they are contracted into three features. Classifiers always requires numbers as input, so in this case the color "Red" has to be an Enum and not a string.

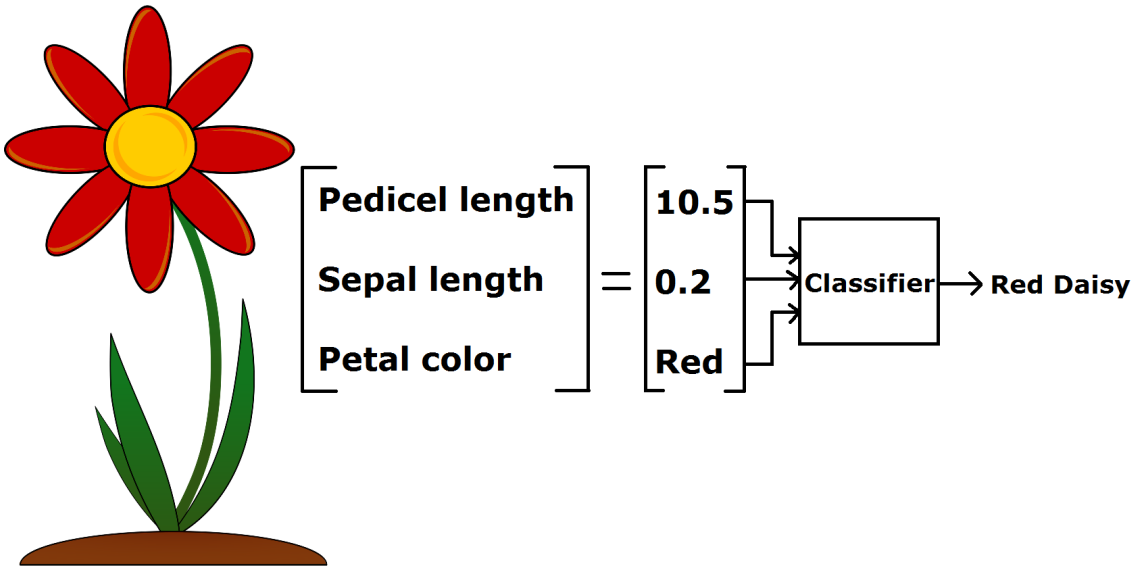


Figure 1: An example of a flower classifier using three different features

One of the biggest challenges when building a classifier is to come up with a good set of features to extract from the data samples that can help distinguish between the different classes. As an example from the flower case; a bad feature could be the pedicel color, because they are green on most flower species and thereby don't provide any variation between classes.

The following subsections will briefly explain about 7 different classifier algorithms and how each one of them make use of the input data set to learn structure from it, and thereafter make a classification of a new input data point.

2.3.1.1 Logistic regression classifier

As the name states, this classifier uses regression to fit boundaries between classes. The regression line can be an arbitrary function of any order which is then used as input to the sigmoid function. The sigmoid function creates a boundary from the regression line which separates two classes. Usually, the boundary between class zero and class one is drawn where the sigmoid function equals 0.5. The equation used for creating a boundary at 0.5 is shown in Equation 1 and an example of separating a one-dimensional data set using this equation is shown in Figure 2. [5] [6]

$$y(x, f) = \begin{cases} 1 \ if \ \frac{1}{1+e^{-f(x)}} > 0.5 \\ else \ 0 \end{cases} \tag{1}$$

$$where \ f = \ regression \ function$$

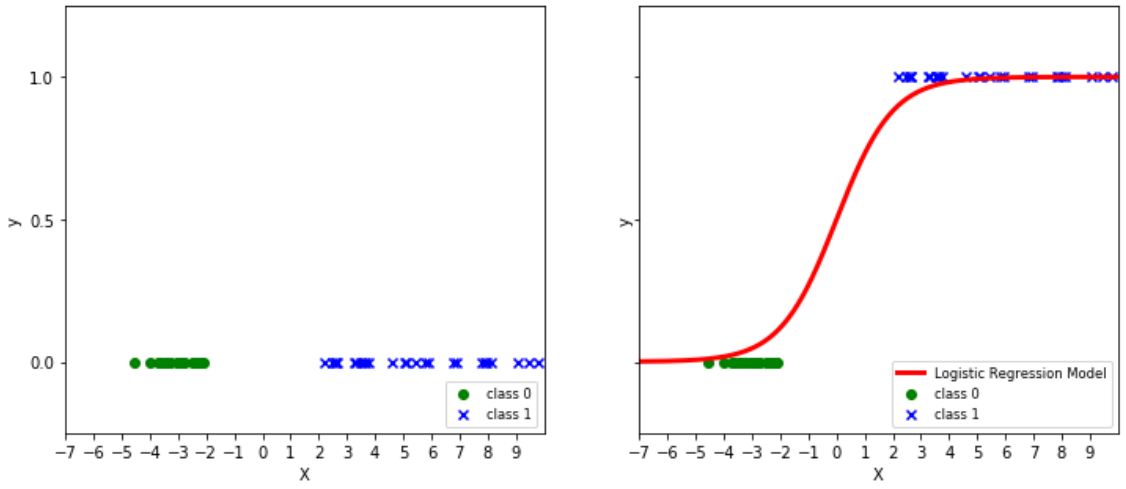


Figure 2: An example of a logistic classifier on a one-dimensional data set

2.3.1.2 Support vector machine

Support vector machine (SVM) is a powerful classifier that is recognized as a good choice of model when fitting high-dimensional data. The underlying theory of fitting a separating line in the feature space is to maximize the margin between the closest points to the line of each class. An example is shown in Figure 3, where the margin that is maximized is the distance between the dotted lines and the red line.

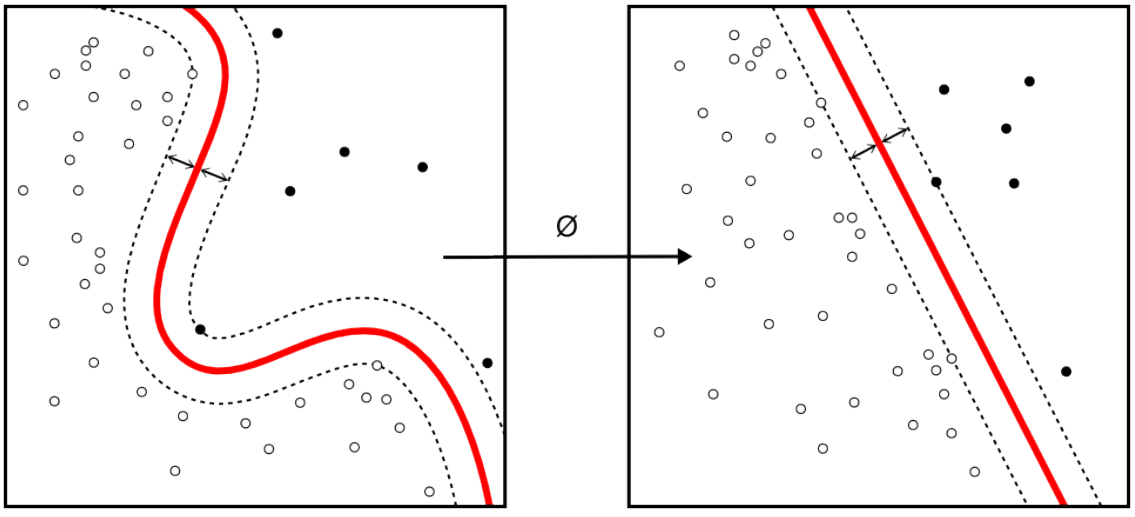


Figure 3: Two examples of SVM on a two-dimensional data set with their maximized margins.

An important argument of the SVM classifier is which kernel function to use. This kernel function drastically changes the properties of the separating line. Figure 4 demonstrates how a data set not linearly separable by a linear kernel can be classified much better using a radial basis function as

kernel. [7]

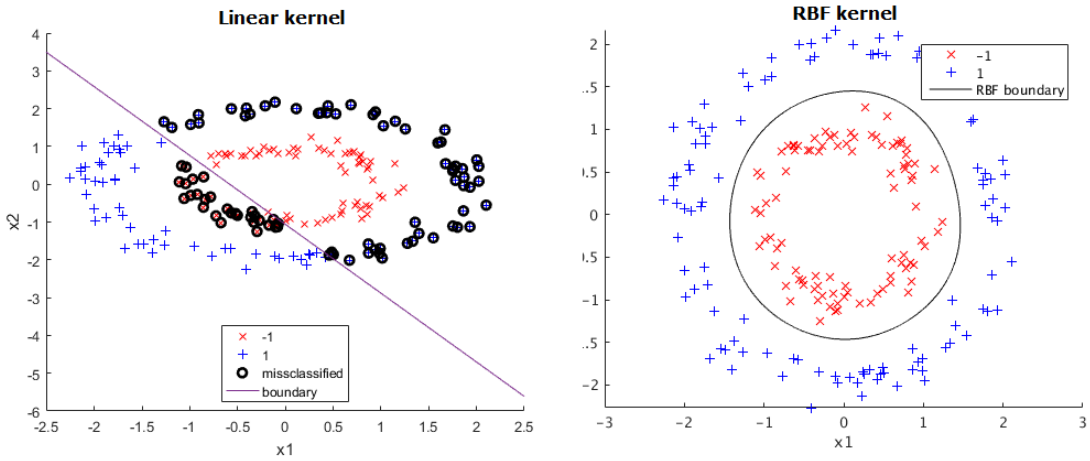


Figure 4: An example of how a RBF kernel can dominate over a linear kernel.

2.3.1.3 Multinomial Naive Bayes

A Bayes classifier uses Bayes rule, shown in Equation 2, to assign the probability of a data point being in a particular class. A threshold can then be decided on how much the probability the result of Equation 2 must be in order to classify the input data point as the positive class.

$$p(C_i|\vec{F}) = \frac{p(\vec{F}|C_i)p(C_i)}{\sum_j p(\vec{F}|C_j)P(C_j)} \tag{2}$$

$$\textit{where } p(C_i|\vec{F}) = \textit{ Probability of a data point with features } \vec{F} \textit{ belonging to class } C_i$$

$$p(\vec{F}|C_i) = \textit{ The likelihood of features } \vec{F} \textit{ given class } C_i$$

$$p(C_i) = \textit{ The probability of class } C_i \textit{ occuring in the data set, i.e. our prior knowledge about the data set}$$

$$\sum_j p(\vec{F}|C_j)P(C_j) = \textit{ A normalizing factor to ensure that } p(C_i|\vec{F}) \textit{ is a propely defined density}$$

The naive part of the classifier is that $p(\vec{F}|C_i) = \prod_j p(F_j|C_i)$, i.e. the likelihood of the feature vector can be represented as independently multiplying the likelihood of each feature. Bayes classifiers are very popular among text classification problems, and the naive part thereby assumes that the words of the text are independent from each other. This is clearly not the case for text but still performs very well. The multinomial Naive Bayes classifier also takes into account the number of occurrences of a particular word in the input string. [6]

2.3.1.4 Random forest

To describe a random forest classifier, a definition of a decision tree first has to be made. Basically, a decision tree is a series of Yes/No questions asked about the data sample, like in Figure 5. The nodes represent the questions and the edges represent the answers. A random forest is self explanatory; it consists of multiple randomly generated decision trees. The mode of all the decision trees will be the classification output from the random forest. [8]

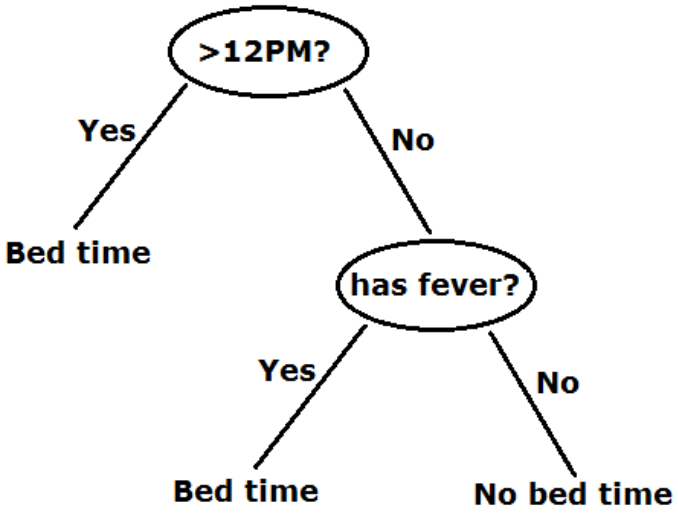


Figure 5: An example of a decision tree

2.3.1.5 AdaBoost

Like random forests, the AdaBoost classifier combines multiple classifiers together. AdaBoost is best used when combining weak classifiers, such as decision trees of only height 1. The algorithm starts by training a model from the training data, and then creating a second model with the purpose of correcting the errors from the first model. This additive procedure is performed until a maximum number of models are created or a perfect prediction is achieved. [9]

2.3.1.6 Artificial neural network Artificial neural networks (ANN) have exploded in popularity the recent years due to the increasing amount of data available and the processing power that is needed to train these networks. The argument why this technique is more powerful than any other machine learning classifier is that it can represent any function, e.g. have a structure of arbitrary complexity to be able to fit even the most complex data.

The type of ANN discussed in this report is of the simplest structure, a feed forward neural network, shown as an example in Figure 6. This particular example has three inputs, one hidden layer with three "neurons" and two outputs, but the fact is that a network can consist of any number of layers and neurons. As Figure 6 might explain, the input of a neuron in layer l is a linear combination of the outputs from the neurons in layer $l-1$. Each of these inputs in every layer is weighted, and the weighted combination of the inputs to a neuron is transformed to an output of zero or one depending on the neuron's activation function. Deciding all the weight matrices $W^{(l)}$ of each layer

l is the learning process. Learning the weights requires a labeled data set and is performed using a backpropagation algorithm, but this is outside the scope of this report. [10]

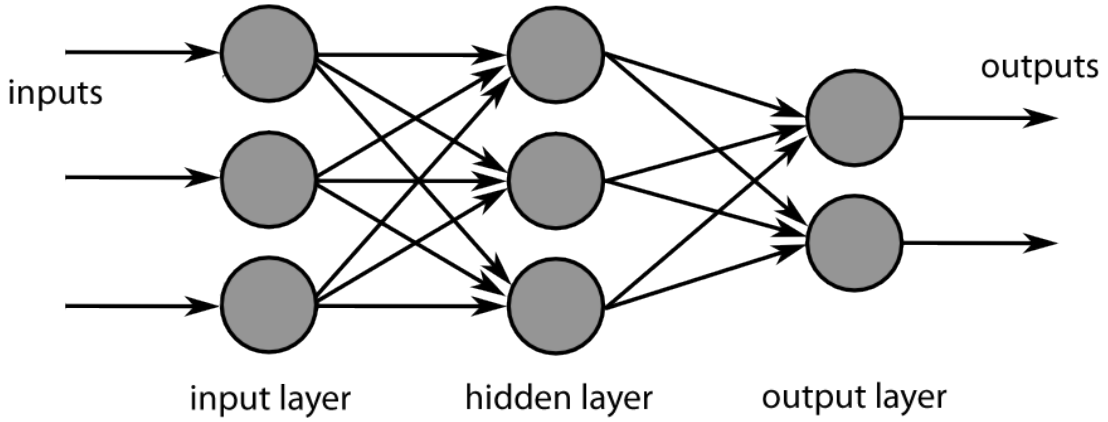


Figure 6: An example of an artificial neural network

2.3.2 Bag-of-words

As machine learning classifiers always need numbers as input, raw text data is not suitable to use directly into a model. A popular technique to solve the problem with text data input is called bag-of-words, which transforms the text string into a vector of word counts. The bag-of-words transformation results in every unique word in the entire data set being represented as its own feature. A feature vector of a data point simply is zero for every feature except for the features representing words that exists in the string. The values of those features are equal to the number of occurrences of the words in the string. An example transformation using bag-of-words is shown in Figure 7, where the data set contains two strings and the feature space is every unique word in the data set. [11]

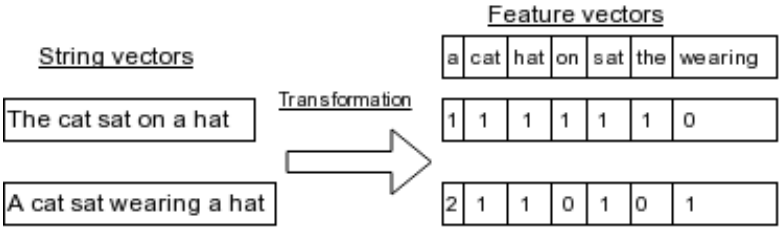


Figure 7: An example of two strings transformed into its common feature space

2.3.3 Principle component analysis

Principle component analysis (PCA) falls in the category of unsupervised learning algorithms. Its a dimensionality reduction technique that projects a data set from its original feature space into a new reduced feature space. The new features are linear combinations of the original features. The

goal of the transformation process is to maximize the variance preserved. For example in Figure 8, when projecting a three dimensional hand onto a two dimensional plane, the variance is maximized when the area of the palm is faced towards the plane. PCA can be used for compressing feature space, feature selection and visualization. This report uses PCA for visualizing high dimensional data.

Figure 8: Projecting a hand onto a two dimensional plane

2.3.4 Training, validation and test data

When training and evaluating a machine learning model, a common practice is to split the data set into three separate parts; training set, validation set and a test set. The purpose of dividing the data set is to minimize the risk of curve-fitting. This phenomena occurs when the model is evaluated using the same data it has been trained on, which means that the model will be good at predicting already known data points, but the performance of predicting new unseen data points is unknown, and usually performing poorly. [12]

A second mistake that can result in a curve-fitted model is basing the selection of hyperparameters on the same data as the model is being optimized on. Hyperparameters are settings of the machine learning model that is algorithm-specific, for example the number of hidden layers in an ANN or the number of weak classifiers to use in AdaBoost. In summary:

1. Train a set of models with different hyperparameters on the training data
2. Evaluate the models on the validation data and pick the best model
3. The selected model can now have its true performance measured using the training data that is yet not seen by the model or the person/algorithm that selected the best model by the validation data.

A popular technique commonly used when optimizing a model is to use cross-validation. The principle of this technique is to re-use the training and validation data by setting the validation data portion to an interval inside the old training data and reseting the validation data to training data. This can be done multiple times for better accuracy of the validation performance. In other words, it removes any eventual bias the validation data had by making a new portion of validation

data for multiple iterations. This is easily understood with a visualization, so an example is shown in figure 9 [13].

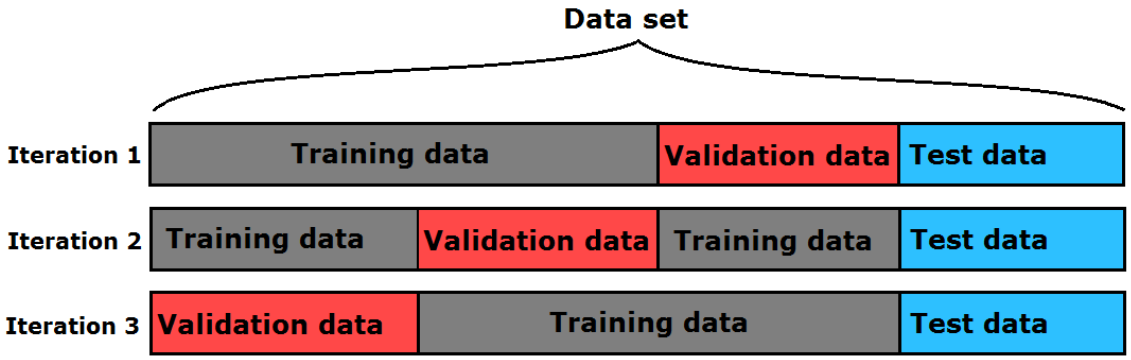


Figure 9: Cross-validation example with a hold out set for final testing.

2.3.5 Performance metrics

When evaluating models from the results of validation data or test data some performance metric has to be compared. The easiest is to sort the results by accuracy, e.g. the percent of successful classifications, but in cases where the distribution of classes among the data points is very skewed this metric might be deceptive. Other common performance metrics are calculated from the confusion matrix. A confusion matrix of a binary classification problem is shown in Figure 10, and an explanation of the contents of the matrix is displayed in Table 1.

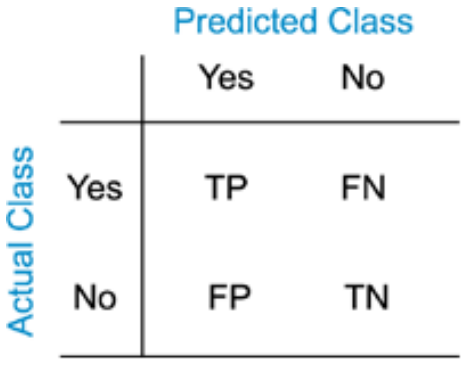


Figure 10: A binary confusion matrix.

Abbreviation	Explanation
TP	Number of data points with class 1 also classified as 1
FP	Number of data points with class 0 but classified as 1
TN	Number of data points with class 0 also classified as 0
FN	Number of data points with class 1 but classified as 0

Table 1: Explanation of confusion matrix content

The performance metrics used in this report additional to accuracy are: sensitivity, specificity, f1-score and AUC. The equation for sensitivity is shown in Equation 3 and can be interpreted as "the percent of class 1 data points that were classified correctly". [14]

$$sensitivity = \frac{TP}{(TP + FN)} \tag{3}$$

Specificity is shown in Equation 4 and can be interpreted as "the percent of class 0 data points that were classified correctly". [14]

$$specificity = \frac{TN}{(TN + FP)} \tag{4}$$

F1-score from Equation 5 is an all-around metric that is good to use instead of accuracy when the distribution of labels are skewed among the data set.

$$\begin{gathered} \textit{F1-score} = \frac{2 * precision * recall}{precision + recall} \\ where\ recall = sensitivity \\ precision = \frac{TP}{(TP + FP)} \end{gathered} \tag{5}$$

Area under the curve, AUC, is a metric extracted from the Receiver operating characteristic (ROC). The ROC curve of a model is a plot with the true positive rate ($sensitivity$) as the y-axis and the false positive rate ($1 - specificity$) as the x-axis. The AUC metric is simply the percent of the graph that is under the ROC curve ($0 < x\text{-}axis < 1$ and $0 < y\text{-}axis < 1$). [15]

3 Method

3.1 Preparatory work

Using machine learning techniques for intrusion detection and firewalls have become a heavily researched area the recent years, so a bit of preparatory work had to be done before starting any implementation by reading up on the subject and any recent published results. There weren't any clear patterns on what makes a successful implementation of this kind, so we gave ourselves free hands using only machine learning roadmaps.
According to Konrad Rieck in his paper *Machine Learning for Application-Layer Intrusion Detection*, he argued that generic features such as length of payload, number of security related keywords and numeric information regarding the byte distribution of the payload can be of great use [16]. Mutaz Alsallal tried some of those features in practice in his paper *Applying Machine Learning to Improve Your Intrusion Detection System* [17]. The IDS showed a great potential of detecting incoming threats, however there was a substantial lack of data.

3.2 Workflow

The work flow of this project have been very much like how machine learning projects in general are done, or even more broadly speaking, the process of data analysis. The project can roughly be divided into five different ordered phases: defining the question or problem, data wrangling and data cleaning, feature engineering, training the data and evaluating it. Even though the process seems highly sequential, this was not always the case. For example extracting features helped us find new information about the data, which made extra data cleaning necessary, and even refining the initial question. Evaluating the different classifiers and the features made us go back to both the data and the features. Some of the misclassified data were obviously out of place which forced another round of data cleaning. A graphical representation of the work flow is shown in Figure 11.

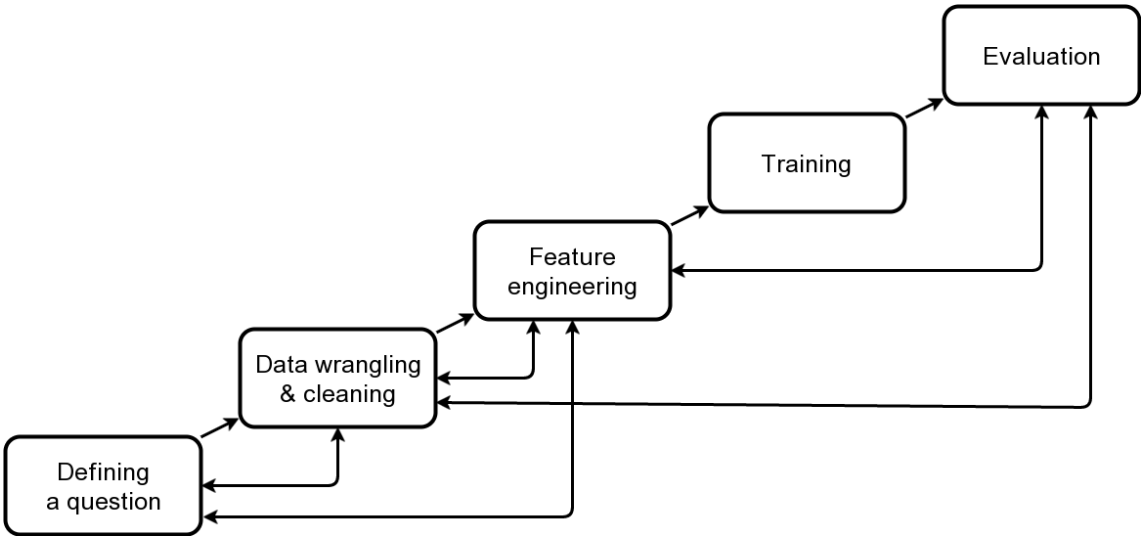


Figure 11: Visualization of the work flow of the project.

3.3 Programming language

The programming language in question was going to be python. The language is very suitable for data analysis, and with the libraries explained below it has become the most popular programming language for machine learning. [18]

These are the main libraries extensively used throughout the project to drastically decrease the amount of code we needed to write ourselves:

NumPy	Numerical Python, a library that adds support for large multidimensional matrices and a collection of high-level mathematical functions. Website: http://www.numpy.org/
Pandas	Builds upon NumPy and provides high-performance table-like data structures well suited for handling and manipulating large sets of data. Website: http://pandas.pydata.org/
Matplotlib	An extensive plotting library that allows for easy visualization of any kind of data analysis. Website: https://matplotlib.org/
Scikit-learn	The swiss army knife of machine learning for python. This library contains all the machine learning algorithms, validation tools and model selection tools we need for the project. Website: http://scikit-learn.org/stable/index.html

This is a security project, so we will not focus on implementing the machine learning algorithms themselves, but only using them for practical implementations. Thanks to scikit-learn this was achievable.

3.4 Documentation

All experiments were coded and tested in Jupyter Notebooks (website: http://jupyter.org/), which is an interactive python environment for data science. With its integrated support for Pandas, Matplotlib, markup language, plots and tables, a much more appealing and understandable presentation of the flow of the code can be made. The reader can also execute the code parts step-by-step to follow our development process.

The notebooks will include results of each part, code documentation and description of how the implementation is structured. Any other documentation about files and how to run our code is presented in README.md files.

3.5 Version control

The version control system used for this project was Git, and more specifically, GitHub. This enabled efficient collaboration and ease for downloading the latest version of the project and training models on multiple computers simultaneously (but separately). Address to the repository can be found in Appendix A.1.

4 Implementation

The following section will explain how the theory and algorithms in the technical background were used; from finding and extracting good data, to classifying it using Python.

4.1 Data wrangling

After a long period of vacuuming the web for good data, we ended up with four different categories, legal input, XSS-attacks, SQL injections and command injections. The data was mostly found from different github repositories. However the data obtained was not always in the format that was necessary for this project, i.e. only the input string. Some of the data were for example inside HTTP POST- and GET requests. Python scripts were used to convert the data in the format that fit this project.

4.2 Data cleaning

After the data wrangling phase described in section 4.1, it was time to clean the dataset. A problem after retrieving data from several sources was the amount of duplicates. A python script was created removing all duplicates, the input that was way to short (and therefore not causing any real attack), empty data points and null data. For the malicious data some manual work was done also. Some of the data points were still in the wrong format or were still not enough to create any harm for a real system.

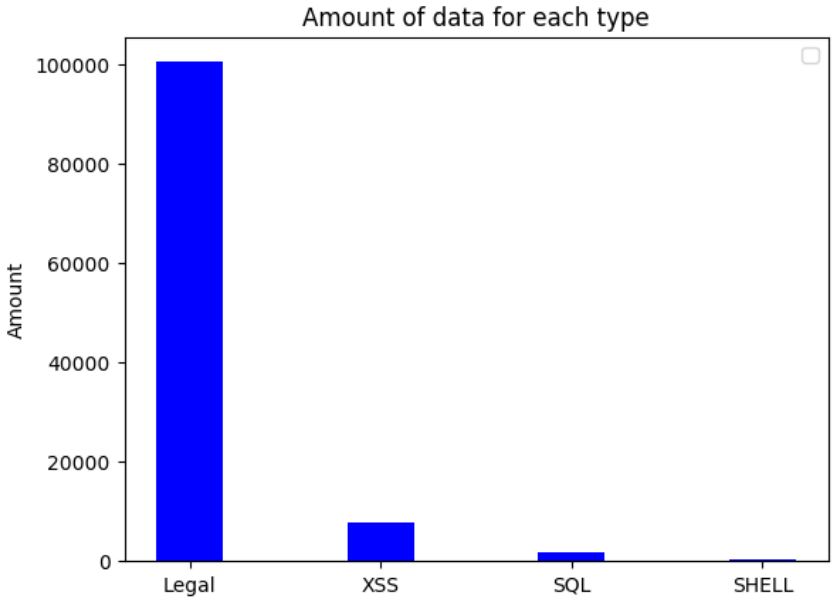


Figure 12: Distribution of the length of the payload.

4.3 Feature engineering

For converting the input data to numeric features two type of techniques were used. A bag-of-words technique and another technique were custom features such as length and byte distribution of the data were used.

4.3.1 Bag of words feature space

We tried a number of well-known bag-of-words approaches to transform our payloads into more suitable feature spaces; a count vectorizer and a TF-IDF vectorizer. The first is using the same approach as explained in section 2.3.2, and the latter is using a weighted version. The weights are automatically set to empathize less common words. [19]

As our payload data samples aren't structured like a word sequence as regular text documents are, additional preprocessing had to be made before transforming the payloads using the bag-of-words vectorizers. We used the N-grams approach to transform our payload data samples into "words" of size N. In particular, we used 1-gram (unigram), 2-gram and 3-gram, and an example of how each is applied is shown in Table 2.

N-gram	Payload input	Payload Output
1-gram	"<script>"	['<','s','c','r','i','p','t','>']
2-gram	"<script>"	['<s','sc','cr','ri','ip','pt','t>']
3-gram	"<script>"	['<sc','scr','cri','rip','ipt','pt>']

Table 2: Example of how the string "<script>" would be transformed when using 1-gram, 2-gram and 3-gram

Each of these N-grams were combined for each vectorizer, which resulted in six different feature spaces:

- 1-gram count vectorizer (175 features)
- 2-gram count vectorizer (4357 features)
- 3-gram count vectorizer (55424 features)
- 1-gram TF-IDF vectorizer (175 features)
- 2-gram TF-IDF vectorizer (4357 features)
- 3-gram TF-IDF vectorizer (55424 features)

As the 3-gram vectorizers might show, they resulted in feature spaces of 55424 features, so going higher than 3-grams weren't an option for us because of limited computing power.

After defining the bag-of-words feature spaces, PCA was used to project these spaces into two dimensions. This was done for visualization purposes only to be able to see if the data behaved randomly or if there was structure between the non-malicious and malicious payloads in the feature spaces. Two extracts of the six feature spaces are shown in Figure 13 and Figure 14, where we can clearly see that the data doesn't behave randomly. In fact, quite a few data points can be classified by hand when only looking at these two dimensions. All six feature spaces had a promising look when visualized using PCA, so every one of them advanced to the next step, which was training and model selection.

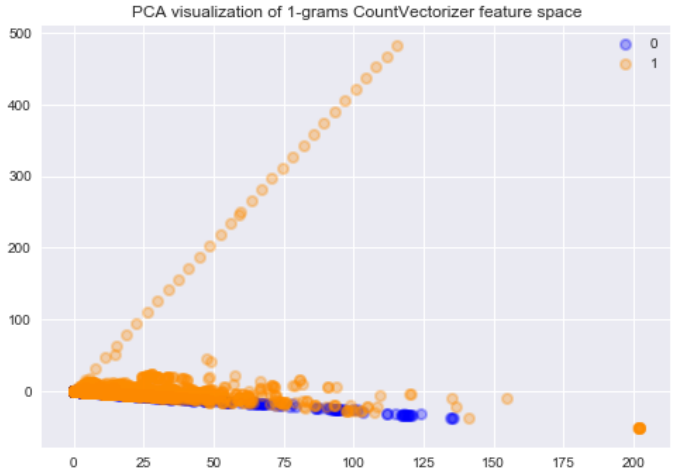


Figure 13: Visualization of 1-gram Count feature space by projecting data set onto 2 dimensions with PCA

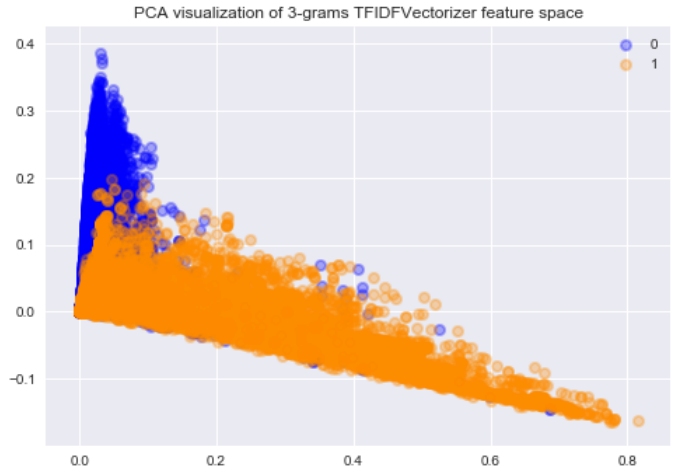


Figure 14: Visualization of 3-grams TF-IDF feature space by projecting data set onto 2 dimensions with PCA

4.3.2 Custom feature space

In addition to the bag-of-words feature space described above, we also tried classifying using ten customized features that was presumed to be of potential usage. Before training the classifiers the seven best features was selected using chi square feature selection. An overview of the process from payload input to a feature vector for the custom features can be seen below.

$$\text{Payload input} = \text{'} < \text{script} > \text{'} \Rightarrow \begin{bmatrix} \text{Length} \\ \text{Non-printable chars} \\ \text{Punctuation chars} \\ \text{Minimum byte} \\ \text{Maximum byte} \\ \text{Mean byte} \\ \text{Standard deviation byte} \\ \text{Distinct bytes} \\ \text{SQL Keywords} \\ \text{Javascript Keywords} \end{bmatrix} = \begin{bmatrix} 8 \\ 0 \\ 2 \\ 60 \\ 116 \\ 97.875 \\ 21.95 \\ 8 \\ 0 \\ 1 \end{bmatrix} = \text{Input to classifier}$$

To visualize the distribution of malicious versus non malicious data the PCA method was used, projecting the data sets onto two dimensions. The result can be seen in figure 15. For examle an "island" of malicious data points can be seen at the right hand side of the figure.

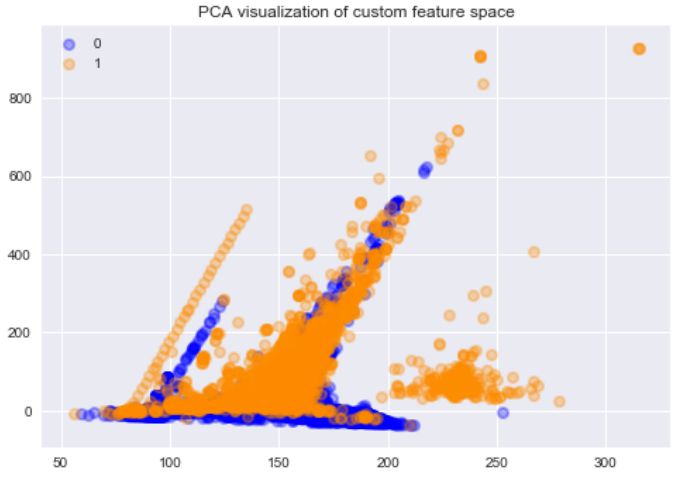
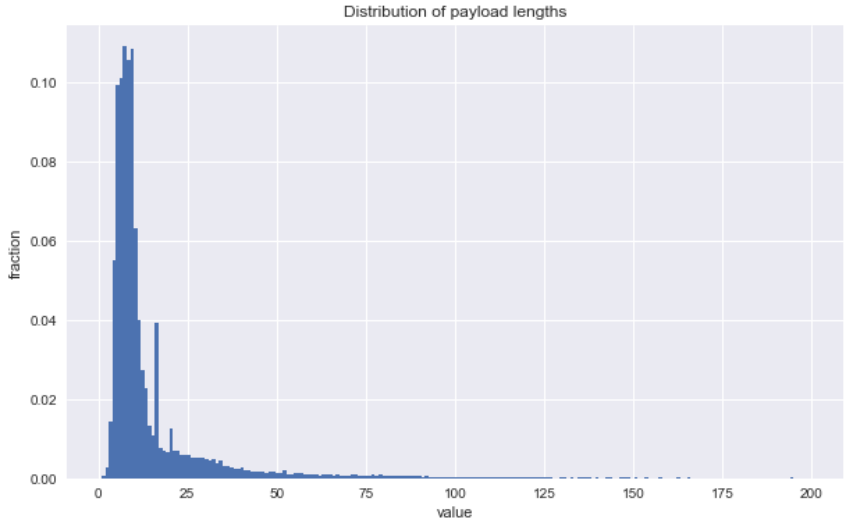


Figure 15: Visualization of custom feature space by projecting data set onto 2 dimensions with PCA

4.3.3 Length

The first custom feature to be implemented was the length of the input. Distribution of the payload lengths can be seen in Figure 16. The distribution had a mean of 16.5 and a standard deviation of 32. Minimum payload length was just one, and the maximum was 974 characters long.



3

Figure 16: Distribution of the length of the payload.

4.3.4 Non-printable characters

Non-printable characters are for example tabs, line breaks and null characters, i.e. characters that does not represent a written symbol. Mean was 0.0074, indicating those are not usually in any of input strings. The maximum value of a payload was 30 non-printable characters. Distribution of non-printable characters within payload can be seen in figure 17. In our data, this seems to be used little to none for both malicious and non-malicious data.

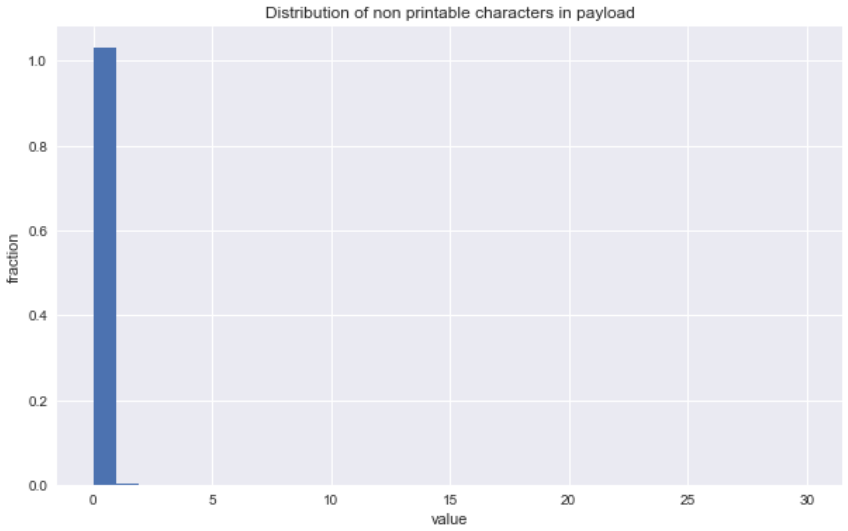


Figure 17: Distribution of non printable characters within payload.

4.3.5 Punctuation characters

This feature describe the number of punctuation characters within payload. This includes characters such as < and ', which are commonly used within both sql injections and cross-site scripting attacks. Looking back at the distribution of malicious versus non-malicious data, this feature seems to reflect this, looking at the distribution in figure 18.

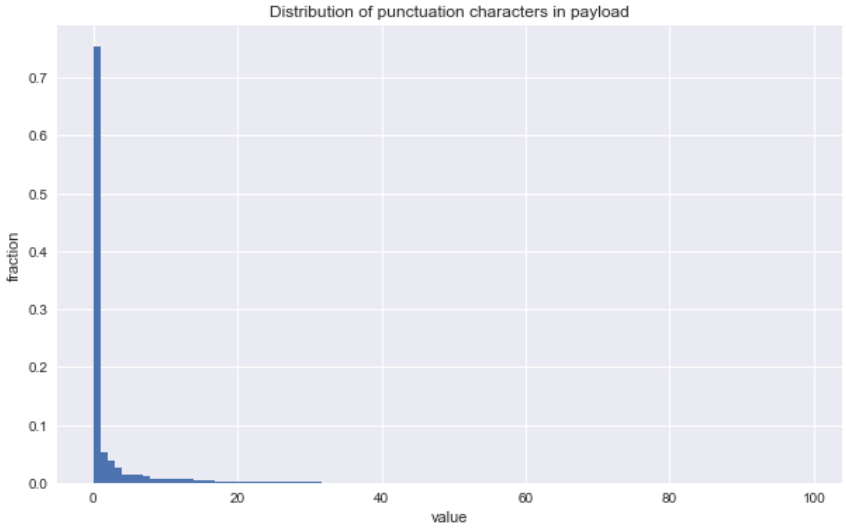


Figure 18: Distribution of punctuation characters within payload.

4.3.6 Minimum byte

A feature describing the minimum byte within the input. The distribution can be seen in figure 19. The average minimum byte is 71 and the standard deviation 26. Comparing the table with the byte value of utf-8 standards, it is not unreasonable that it will have a big impact due to the peak around 100 bytes is normal characters from the alphabet.

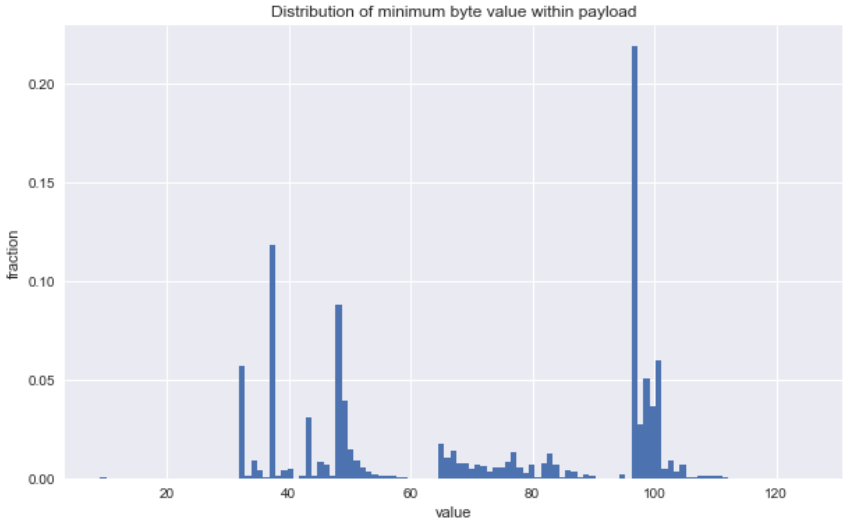


Figure 19: Distribution of minimum byte within the payload.

4.3.7 Maximum byte

Similar to the minimum byte, a maximum byte feature was created. Looking at the distribution in figure 20, most of the data seems concentrated around 110-120 bytes, which are normal characters. The average maximum byte in an input is 109 bytes with a standard deviation of 20.

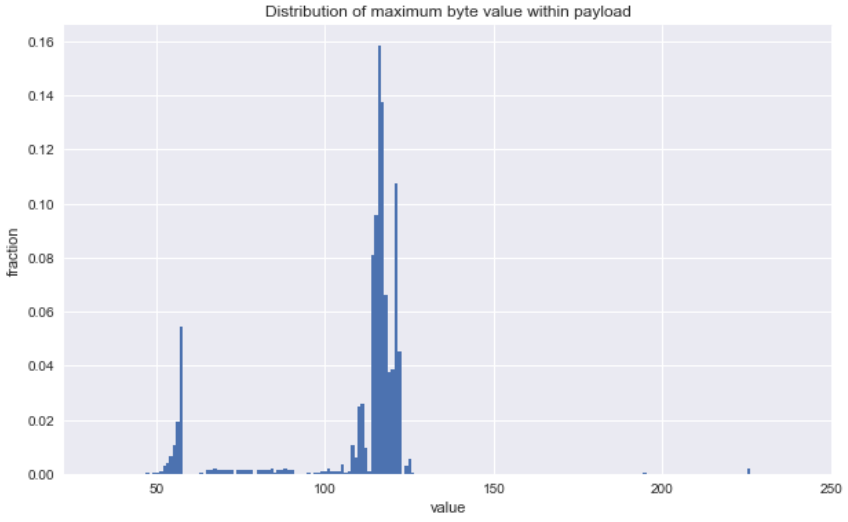


Figure 20: Distribution of maximum byte within the payload.

4.3.8 Mean byte

A feature describing the mean byte value of the input character values. The distribution of the whole payload can be seen in figure 21. As expected theres a peak around 110, i.e. normal characters.

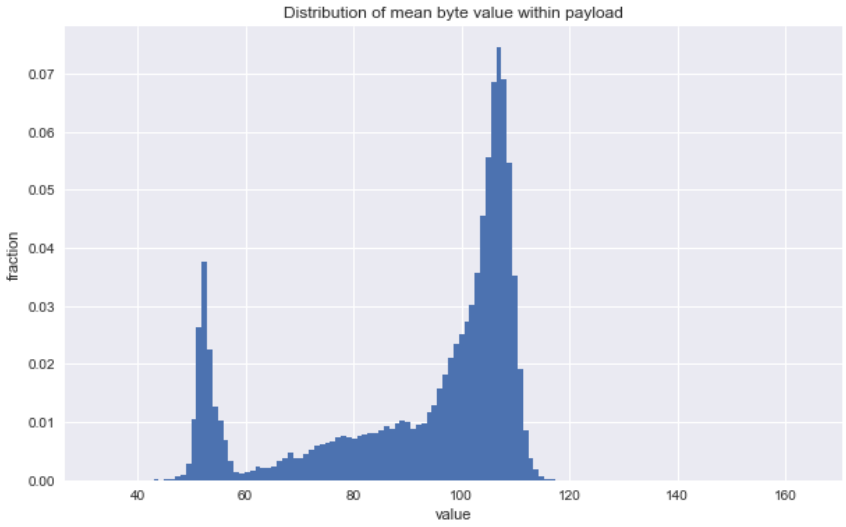


Figure 21: Distribution of mean byte within payload.

4.3.9 Standard deviation byte

This feature define the standard deviation within the input. The distribution can be seen in figure 22. The average standard deviation of the entire payload is 12.

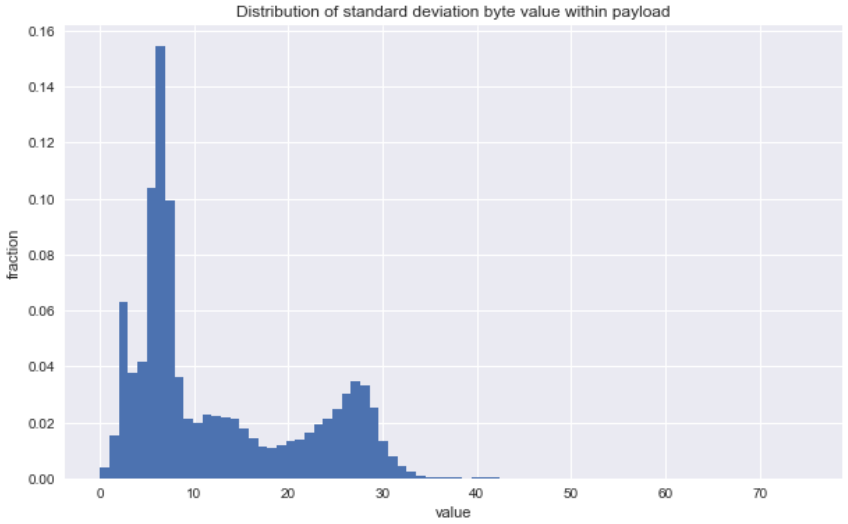


Figure 22: Distribution of the payloads standard deviation.

4.3.10 Distinct bytes

Feature describing the number of distinct bytes within an input string. Distribution can be seen in figure 23. The average number of distinct bytes is 9.5 with a standard deviation of 7.4.

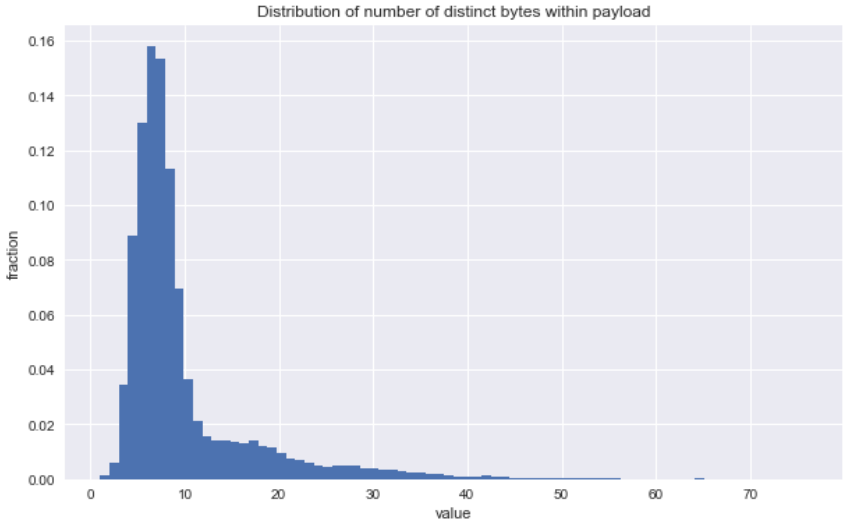


Figure 23: Distribution of number of distinct bytes within payload.

4.3.11 SQL keywords

As the malicious data sets were mostly from SQL injections and cross-site scripting, a feature describing the number of SQL keywords inside an input. The distribution can be seen in figure 24. The mean was 0.2 SQL related words, but with a maximum of 15 words.

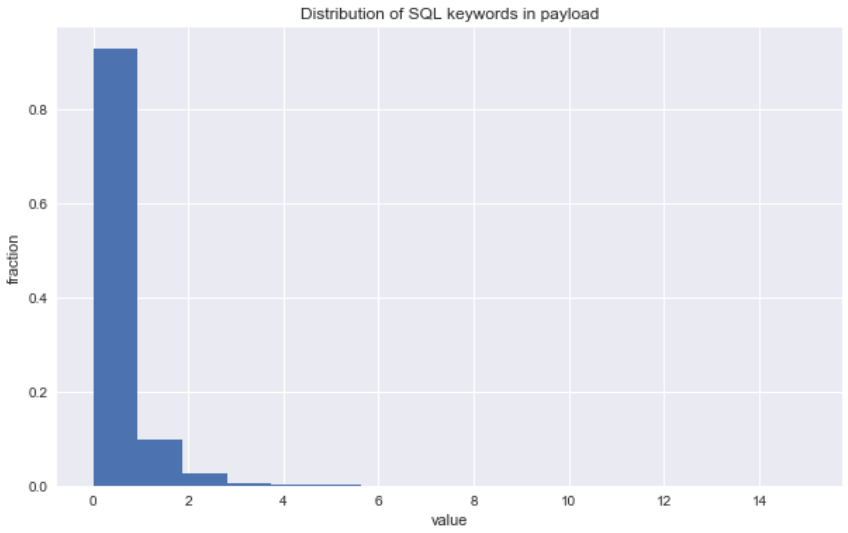


Figure 24: Distribution of SQL related words inside the payload.

4.3.12 Javascript keywords

A feature describing the number of JavaScript related words for an given input. The distribution is shown in figure 25. In average there is 0.35 javascript related words for a data point, with a maximum of 16 words.

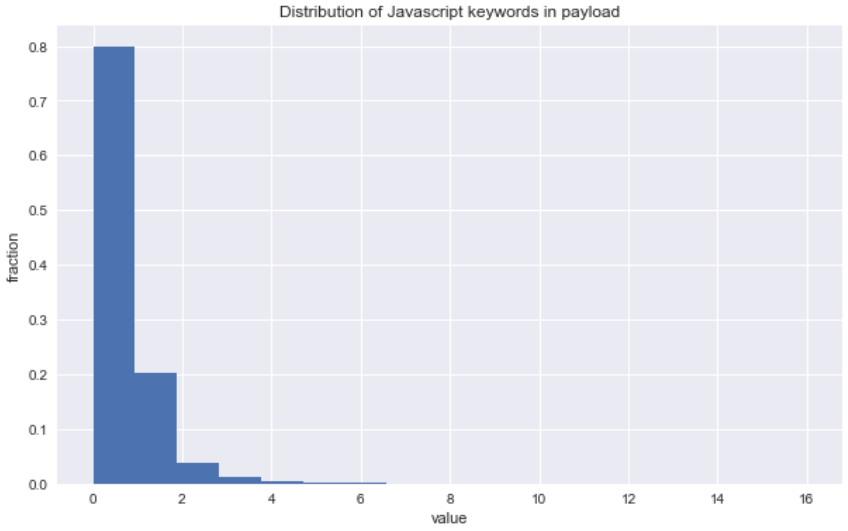


Figure 25: Distribution of JavaScript related words inside the payload.

4.4 Model selection

After creating our data set, cleaning it and constructing features, the next step was to select and train our models. The classifiers chosen are listed below and they were trained on all of our seven feature spaces separately:

- Feed-forward artificial neural network (MLPClassifier)
- Random forest classifier
- Multinomial Naive Bayes classifier (MultinomialNB)
- Decision tree
- Support vector machine (SVM)
- Stochastic gradient decent classifier (SGDClassifier)
- Logistic regression
- AdaBoost

Our model construction pipeline follows the approach explained in section 2.3.4 and a simplified

description of the process is described in Algorithm 1.

Algorithm 1: Our process of transforming the data set, optimizing a model and calculating its true performance on a test set

input : $Clfs$ as a list of classifiers,
$feature\_spaces$ as a list of feature spaces to train each classifier in,
$dict\_of\_params$ as a collection of hyperparameters to optimize for each classifier,
D as a data set of labeled payloads
output: T as a table of performance metrics for each model

1 **for** $fs \in feature\_spaces$ **do**
2 **for** $Clf \in Clfs$ **do**
3 $D_{train}, D_{test} \leftarrow$ Split D into train and test data ;
4 $M_{best} \leftarrow null$;
5 **for** $params \in dict\_of\_params[Clf]$ **do**
6 $D'_{train} \leftarrow fs.transform(D_{train})$;
7 $M_{temp} \leftarrow cross\text{-}validation\_train(D'_{train})$;
8 $M_{best} \leftarrow best\_model(M_{best}, M_{temp})$;
9 **end**
10 $test\_result \leftarrow M_{best}.predict(D_{test})$;
11 $Performance\_metrics \leftarrow get\_performance\_metrics(test\_result)$;
12 $T[fs\ Clf] \leftarrow Performance\_metrics, M_{best}$;
13 **end**
14 **end**

The hyperparameters and their list of values that were included when optimizing are not presented here but only in the code. When optimizing a model in any of the bag-of-word spaces, there was an additional parameter introduced, *min_df*, that mentions how many times a word has to occur in the data set before it becoming a feature.

Additional to the model itself, the metrics saved in table T for later use were: accuracy, F1-score, confusion matrix, sensitivity, specificity, AUC and points to plot the ROC curve. The metric used to compare the models in line 8 of Algorithm 1 was F1-score. We chose to compare models using F1-score instead of the commonly used accuracy measure because of the unbalanced distribution of labels in the payload data set, see Figure 12. For example, with about 90% of the data being non-malicious, if the classifier would classify everything as non-malicious, it would result in an accuracy of 90%, which can be very misleading!

Eight classifiers trained on seven different feature spaces means that we ended up with a total of 56 trained models. This is not the case because there were a number of classifiers that were hard to train on sparse matrices (bag-of-words feature space), so we ended up in 32 different models. See Appendix A.2 to see which combinations were successful in completing training.

4.5 Demonstration website

After evaluating the performance and selecting the best classifier it was time to put it into an environment it would actually be used in a real scenario, a website. This allowed us to make manual tests as well as demonstrating the classifier. As the language of choice was Python, a simple Python HTTP server was setup, just to classify an input and respond with either malicious

or legal. To make the actual visualization cleaner and more dynamic another server was setup, using Node, Express and Angular 2. On the web server one can manually type in a string, which will then go through the classifier and then either be shown as legal or malicious. The resulting website can be seen in figure 26.

Figure 26: Demonstration website using Node.js and Angular 2. One can type an input inside the text field and after validation it will be displayed as either legal or malicious.

5 Results

The goal of the project was to evaluate the potential of machine learning for intrusion detection on an application level. The result reflecting this question will be presented as a set of classifiers able to identify malicious payloads. As input to the classifiers, two types of constructing feature spaces has been used; the bag-of-words technique and feature extraction of individual features combined into a custom feature space.

A subgoal was to find out which type of features are extraordinary good for deciding if the input is malicious or harmless. To solve this for the custom feature space a shi-squared scoring method was used.

5.1 Classifiers

Table 3 demonstrates the F1-scores of the final optimized models tested on out-of-sample test data. The entries containing N/A are the combinations of features and classifiers that didn't work for the scikit-learn library. Most of the models show an outstanding result with a total mean F1-score of 0.9949. The top performing classifier across all feature spaces were the random forest classifier with a mean F1-score of 0.9989.

Table 4 and Table 5 displays the top four and bottom four performing classifiers. There's a clear pattern on which combination of features and classifiers performed the best, namely 1-gram and

	custom	counts 1grams	count 2grams	count 3grams	tfidf 1grams	tfidf 2grams	tfidf 3grams	Avg Feature
MLPClassifier	0.9960	N/A	N/A	N/A	N/A	N/A	N/A	0.9960
RandomForest	0.9983	0.9991	0.9991	0.9985	0.9991	0.9992	0.9987	0.9989
MultinomialNB	0.9570	0.9950	0.9968	0.9965	0.9951	0.9976	0.9978	0.9908
DecisionTree	0.9939	N/A	N/A	N/A	N/A	N/A	N/A	0.9939
SVM	0.9968	0.9985	0.9988	0.9982	0.9968	0.9982	0.9972	0.9978
SGD	0.9899	N/A	N/A	N/A	N/A	N/A	N/A	0.9899
Logistic	0.9915	0.9980	0.9988	0.9984	0.9979	0.9990	0.9988	0.9975
AdaBoostClassifier	0.9943	N/A	N/A	N/A	N/A	N/A	N/A	0.9943
Avg Classifier	0.9897	0.9976	0.9984	0.9979	0.9972	0.9985	0.9981	0.9949

Table 3: F1-score of each classifier for each feature space

2-gram with random forest classifier. The best performing classifier misclassified payloads only 0.14% of the time, while the worst performing classifier had a misclassification rate of 7.57%

Feature space & classifier	**F1-score**	**accuracy**	**sensitivity**	**specificity**	**AUC**	**conf_matrix**
tfidf 2grams RandomForest	0.9992	0.9986	0.9883	0.9996	0.9994	[[20097 8] [23 1944]]
count 2grams RandomForest	0.9991	0.9984	0.9883	0.9994	0.9991	[[20093 12] [23 1944]]
count 1grams RandomForest	0.9991	0.9984	0.9898	0.9993	0.9994	[[20090 15] [20 1947]]
tfidf 1grams RandomForest	0.9991	0.9983	0.9929	0.9989	0.9992	[[20082 23] [14 1953]]

Table 4: The four classifiers with highest F1-score

Feature space & classifier	**F1-score**	**accuracy**	**sensitivity**	**specificity**	**AUC**	**conf_matrix**
custom DecisionTree	0.9939	0.9889	0.8958	0.998	0.9909	[[20065 40] [205 1762]]
custom Logistic	0.9915	0.9845	0.8622	0.9964	0.9902	[[20033 72] [271 1696]]
custom SGD	0.9899	0.9816	0.8775	0.9918	0.9846	[[19940 165] [241 1726]]
custom MultinomialNB	0.957	0.9243	0.9161	0.9251	0.9713	[[18600 1505] [165 1802]]

Table 5: The four classifiers with worst F1-score

Table 6 and Table 7 show some of the most common payloads classified as false positive and false negative respectively.

Payload
Scripts
$passthru(chr(105).chr(100))$exit()
Play%20On%20%E2%80%93%20Carrie%20Underwood%20%E2%80 %93%20Listen%20free%20and%20discover%20music%20at%20Last.fm
ertl 127.0.0.1 — id—
—/usr/bin/id
Steep%20and%20Cheap%3A%20Outdoor%20Research%20Ellipse %20Jacket%20-%20Women%27s%20-%20%24129.99%20-%2063%25%20off
1 order by 12

Table 6: A sample of payloads commonly classified as false positive

Payload
bfilename
window.alert(”Bonjour !”);
0x77616974666F722064656C61792027303A303A313027
<username>’–
<![
b=”URL(”;
b=‘*/(1)}‘;
—id;
{61}l{65}rt‘1‘
)”.sleep(10).”
<style>*{display:table; border:10px solid red;} </style>
//*
’test
as
charset=utf-
”+document.cookie+”

Table 7: A sample of payloads commonly classified as false negatives

A visualization of the results from Table 3 is shown in Figure 27, where we can see once again that the winner is random forest, with logistic regression and SVM as close runner-ups.

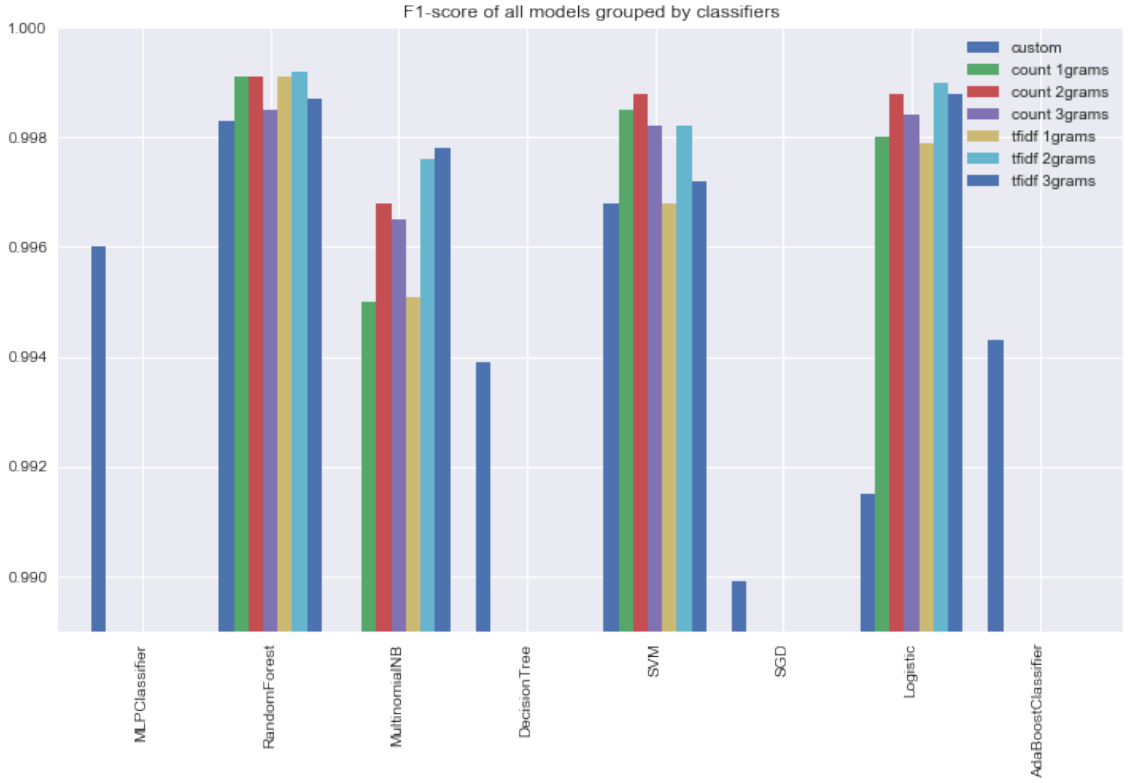


Figure 27: Performance of classifiers measured by F1-score and grouped by classifier

Figures 28, 29 and 30 visualize how the cross-validation performance increased with the increase of dataset size. These can be used to determine whether its worth to gather more data to the dataset or not.

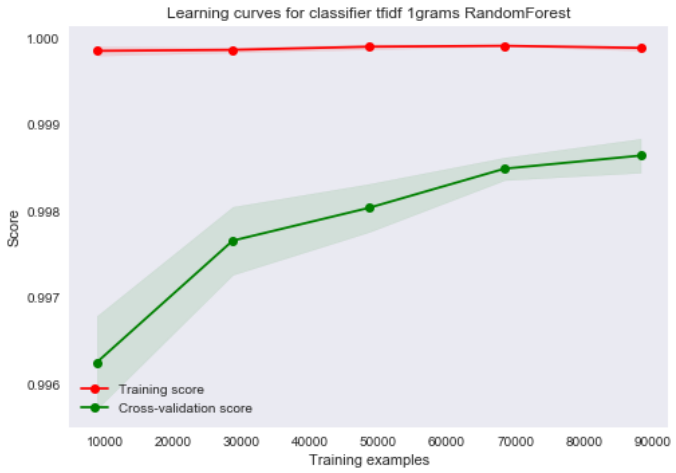


Figure 28: Learning curve for 1-gram TF-IDF random forest

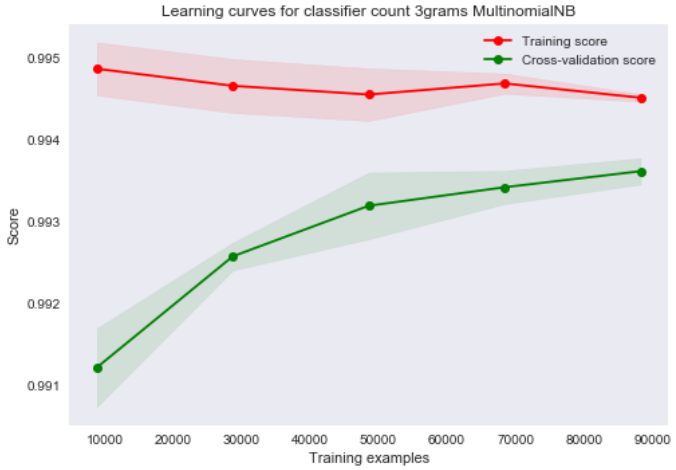


Figure 29: Learning curve for 3-gram Count multinomial Naive Bayes

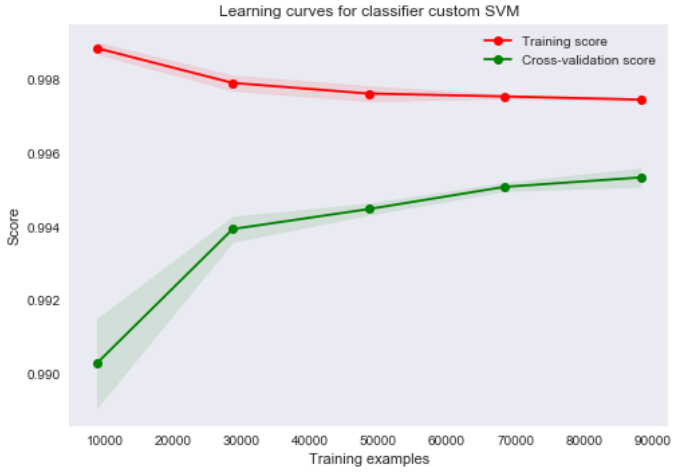


Figure 30: Learning curve for custom feature space SVM

Lastly, the Receiver Operating Characteristic (ROC) curve is drawn in Figure 31 for the top three and bottom three performing models. Recall from Table 4 and Table 5 which these models are. The ROC curve demonstrates two things. The first being a visualization of the area under the curve (AUC), which is so high for the top three models (exact values presented in Table 4) that it is required to zoom in to see the boundaries in the upper left corner. The second thing visualized in the ROC curve is the sensitivity/specificity trade-off, which are the actual lines plotted. The dots plotted are the current sensitivity and specificity for each model.

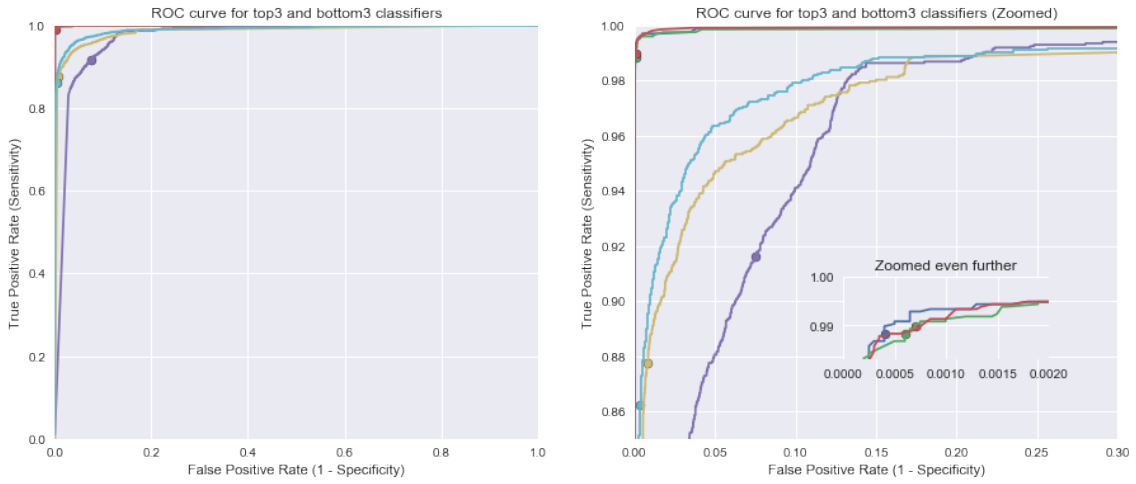


Figure 31: To the left: ROC curve for the top 3 and bottom 3 performing classifiers. To the right: the same graph but zoomed in.

5.2 Feature spaces

To score the value of the different features the library Scikit-learn was used with it's built in function SelectKBest. The algorithm of use was a chi-squared scoring method. Results can be seen in Figure 32. The length of the input and the number of punctuation characters in it had a significant impact compared to the others. The closest ones after that was the number of distinct bytes, minimum byte and the standard deviation byte. Worst of all features was the number of printable characters within a payload. Only the seven best features were chosen for the final training.

Even though the SVM and random forest using the custom features looked promising (Figure 27 and Appendix A.2), the bag-of-words feature spaces were the superior ones across all classifiers they were trained with. On average, the 1-gram underperformed the 2-grams and 3-grams, but there weren't any patterns noticeable if the TF-IDF vectorizer was better than the count vectorizer. It seemed to be redundant of which type of word vectorizer to use.

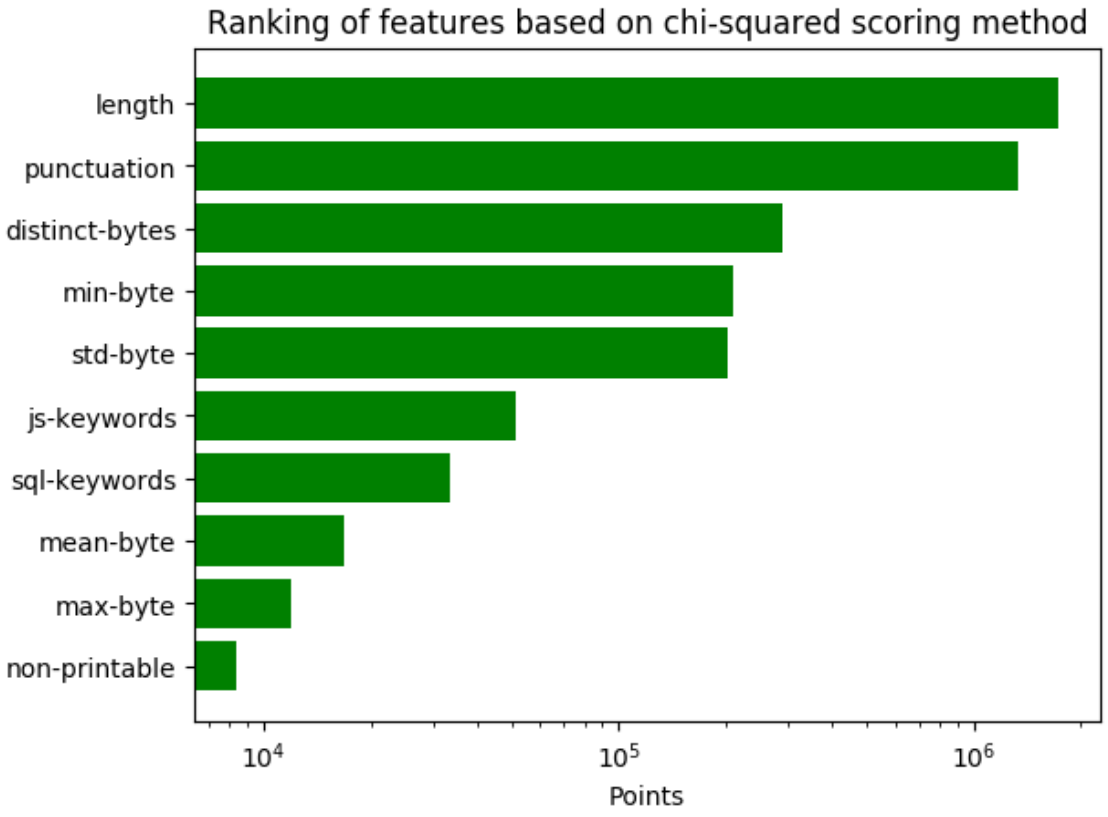


Figure 32: Feature rankings from a chi-squared scoring method. The graph is shown in logarithmic scale.

6 Discussion

In this project we have investigated the potential of supervised learning for detecting threats such as SQL injections and cross-site scripting. The result is a set of classifiers, where of some are exceptionally capable of detecting incoming threats. The classifier with the best F1-score was integrated into a dummy web server for demonstration purposes. Considering the relative fast time of classification, it is not unreasonable to examine integrating a similar solution to a real web server. This chapter will go through the deficiencies of the final product, and which improvements that could be made in order to make the classifier more reliable.

6.1 Developer tools

We are extremely satisfied with the flexibility and the simplicity python's data analysis libraries offered for the project. Its well suited for machine learning tasks like ours and Jupyter Notebooks allowed for a flexible and efficient workflow. The notebooks also resulted in an easy-to-read

presentation of the code.

6.2 Deficiencies

If most of our models performed so well, why are injection attacks still on the OWASP top 10 vulnerability list? We believe that there are a number of reasons why intrusion prevention system projects such as ours can fail in a live setting even though the performance on a test data set is extremely high.

The first and biggest deficiency is the dataset. In our case, we created our own dataset by pooling a large amount of smaller datasets together. Even though the biases in the dataset is lowered by combining multiple sources, we experienced that some of the data were low quality or even mislabeled (for example "bfilename" and "as" in Table 7), so our dataset resulted in a combination of the biases. Some biases in our dataset that both had a negative and positive effect on the performance are:

- Unrealistic distribution of labels, i.e. 10% of the payloads in the dataset were malicious, which is much lower in reality.
- Only SQL-, XSS-, and shell script injections were included as malicious payloads.
- Our non-malicious data were typically not large strings.
- Our non-malicious data came from only two different sources, which meant that the payloads were very domain-specific to those servers that recorded the data. For example, theres a huge amount of Spanish and English words in the non-malicious data, but barely any words from any other language.

The hardest issue to prevent is the non-malicious data being biased towards the domain-specific traffic from the two servers. There is simply no easy way of gathering a balanced non-malicious dataset. There are also more complex non-malicious payloads that can be very hard for the classifier to learn that its not harmful. The most extreme case would be non-malicious payloads containing forum posts about injection attacks to be uploaded on a Internet security forum.

A second possible deficiency would be the scalability. We have integrated our WAF onto a dummy python server and the execution time is very acceptable. Though, the scalability of this implementation have not been tested. Commonly, intrusion detection systems and firewalls are installed directly after the router in a network. But then another problem arises; for those types of firewalls to understand the packet at the application layer, you need reassemble the full packet - which in itself is time consuming. The other way would be installing our classifier directly in the application, but then if a business is running more than one server, it would need to be installed in all environments.

Lastly, a potential upcoming threat to machine learning based firewalls might be counter models. As Sergio Pastrana et al. presents in their paper *Anomalous Web Payload Detection: Evaluating the Resilience of 1-grams Based Classifiers*, they propose a Genetic Programming technique that can learn to bypass a machine learning firewall. Their implementation takes a payload detectable by the firewall as input, and then returns a mutation that will go undetected. For their implementation to work, the data the machine learning firewall was trained on has to be known. Since the sources we gathered data from are all publicly available our WAF will be vulnerable to their bypass technique.

6.3 Future work

What have made machine learning excel in so many areas lately is the rapid growth of data. The potential of our machine learning based WAF is therefor strongly dependent on having a good data set, to make the classifiers as good as possible. Always maintaining the WAF with new data when new threats are discovered should be a high priority. When a zero-day vulnerability is discovered this should be added to the data set.

If it's necessary to take different actions depending on what malicious threat an input is classified into, our classifier can be changed to classify which kind of threat is discovered instead of just labeling malicious or non-malicious. This was however not a necessity within the scope of this project.

Another potential improvement of the web application firewall is to find new valuable features. There is also a possibility of combining the bag of words features with the custom features we have created. Due to the immense feature size difference between those sets the features need to be weighted differently.

As previously discussed, the dataset is not of the highest quality. For example some payloads found were even mislabeled. Thereby, any future work of this project would prioritize further cleaning of the dataset, because "garbage in, garbage out". Also, the Figures 28, 29, and 30 suggests that expanding the dataset would most likely contribute to better performance, since the slope of the cross-validation score is still in an upslope when using the dataset in its current size.

Lastly, further work can be invested in examining and adjusting the sensitivity/specificity trade-off of the models, see the ROC curves in Figure 31. All classifiers selected are probabilistic and currently classifies as malicious when the probability is at least 50%. This threshold can be lowered for higher sensitivity, and as Figure 31 shows, the sensitivity can be increased with reasonably low loss of specificity. In other words, the number of false negatives can be decreased with an increased number of false positives. If any of the models would to be implemented in a real scenario a parameter could be this threshold that the system administrator can adjust depending on whether he/she wants to prioritize a higher sensitivity or maintain the specificity.

7 Conclusion

The main purpose of the project was to investigate the potential of using machine learning for detecting malicious threats with a web application firewall. Considering the limited time of this project and the minimal data sets available online, we believe that machine learning has a huge potential within this area, and even cyber security in general. All the best classifiers (where a 2-grams vectorizer using random forest was the absolute best) had 99.86% accuracy and a sensitivity of 98.83%. A clear pattern was that the random forest classifier outperforms all others on the data set that was acquired. Nevertheless, for the custom features, the support vector machine managed to get a slightly better sensitivity. Even as simple ideas for features such as the bag-of-words technique was proven to have a great potential.

Even though solutions to mitigating threats such as white listing and sanitizers will likely pick up most the threats out there, there will come a time when a zero-day vulnerability will be able

to bypass these kinds of solutions. Therefor we deem that having machine learning based WAF in combination with these techniques will be sufficient to mitigate most threats on the web.

A subproblem to solve within the scope of this project was to find relevant features for detecting malicious threats. In addition to the bag-of-word features ten custom features were found. Using a chi-squared based scoring function we were able to cherry pick the best features to make a classifier as accurate as possible.

The length of the input, the number of punctuation characters and the number of distinct bytes were the three best features that were found. The number of printable characters was however the worst one, and looking back at the distributions, this is not a surprise. Comparing different similar features we can notice some interesting differences. For example the minimum byte of a payload is far more deciding than it's maximum byte. The standard deviation and the number of distinct bytes has also far more impact than the mean.

References

[1] A. Saha and S. Sanyal, "Application Layer Intrusion Detection with Combination of Explicit-RuleBased and Machine Learning Algorithms and Deployment in CyberDefence Program," nov 2014, [Retrieved: 2017-05-31]. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1411/1411.3089.pdf

[2] J. Manico. (2015) Owasp application security verification standard 3.0. [Hämtad: 2016-02-07]. [Online]. Available: https://www.owasp.org/images/6/67/OWASPApplicationSecurityVerificationStandard3.0.pdf

[3] C. L. A. K. R. H. S. Orrin. Http request smuggling. [Visited: 2017-05-11]. [Online]. Available: http://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf

[4] J. McMillan. (2009) Idfaq: What is the difference between an ips and a web application firewall? [Retrieved: 2017-05-11]. [Online]. Available: https://www.sans.org/security-resources/idfaq/what-is-the-difference-between-an-ips-and-a-web-application-firewall/1/25

[5] M. Humphry. Continuous output - the sigmoid function. [Visited: 2017-05-11]. [Online]. Available: http://www.computing.dcu.ie/~humphrys/Notes/Neural/sigmoid.html

[6] M. G. Simon Rogers, *A First Course in Machine Learning.* Chapman & Hall, 2016, ch. Classification, pp. 167–180.

[7] B. S. Jean-Philippe Vert, Koji Tsuda. A primer on kernel methods. [Visited: 2017-05-11]. [Online]. Available: http://members.cbio.mines-paristech.fr/~jvert/publi/04kmcbbook/kernelprimer.pdf

[8] L. Breiman. Random forests. [Visited: 2017-05-11]. [Online]. Available: https://link.springer.com/article/10.1023%2FA%3A1010933404324

[9] J. Brownlee. Boosting and adaboost for machine learning. [Visited: 2017-05-11]. [Online]. Available: http://machinelearningmastery.com/boosting-and-adaboost-for-machine-learning/

[10] S. Haykin, *Neural Networks: A Comprehensive Foundation.* Prentice Hall, 1998.

[11] A. Zheng, *Mastering Feature Engineering: Principles and techniques for data scientists.* O'Reilly, early release ed., ch. Basic Feature Engineering for Text Data, pp. 15–31.

[12] S. L. Arlinghaus, "Phb practical handbook of curve fitting," 1994.

[13] M. G. Simon Rogers, *A First Course in Machine Learning.* Chapman & Hall, 2016, ch. Generalization and over-fitting, pp. 32–34.

[14] K. Markham. Evaluating a classification model. [Visited: 2017-05-11]. [Online]. Available: https://github.com/justmarkham/scikit-learn-videos/blob/master/09_classification_metrics.ipynb

[15] M. G. Simon Rogers, *A First Course in Machine Learning.* Chapman & Hall, 2016, ch. Assessing Classification Performance, pp. 196–201.

[16] K. Rieck, "Application Layer Intrusion Detection with Combination of Explicit-RuleBased and Machine Learning Algorithms and Deployment in CyberDefence Program," vol. 25, jan 2009, [Retrieved: 2017-05-11]. [Online]. Available: https://depositonce.tu-berlin.de/bitstream/11303/2496/2/Dokument_38.pdf

[17] M. Alsallal. Applying machine learning to improve your intrusion detection system. [Visited: 2017-05-11]. [Online]. Available: https://securityintelligence.com/applying-machine-learning-to-improve-your-intrusion-detection-system/

[18] J.-F. Puget. The most popular language for machine learning and data science is .... [Visited: 2017-05-11]. [Online]. Available: http://www.kdnuggets.com/2017/01/most-popular-language-machine-learning-data-science.html

[19] A. Zheng, *Mastering Feature Engineering: Principles and techniques for data scientists.* O'Reilly, early release ed., ch. The Effects of Feature Scaling: From Bag-of-Words to Tf-Idf, pp. 33–50.

Appendices

A.1 Github repository

https://github.com/grananqvist/TDA602_ApplicationIPS

Final collection of trained models with their metrics in a pickle file (trained_classifiers.p). The file was too big for Github
https://1drv.ms/f/s!Aj1zBHCOJiQFgbQwDswBYtpzB1Pulg

A.2 Complete table of results

Feature space & Classifier	F1-score	accuracy	sensitivity	specificity	auc	conf_matrix	params
tfidf 2grams RandomForest	0.9992	0.9986	0.9883	0.9996	0.9994	[[20097 8] [23 1944]]	'vect__min_df': 5 , 'clf__n_estimators': 6
count 2grams RandomForest	0.9991	0.9984	0.9883	0.9994	0.9991	[[20093 12] [23 1944]]	'vect__min_df': 1 , 'clf__n_estimators': 60
count 1grams RandomForest	0.9991	0.9984	0.9898	0.9993	0.9994	[[20090 15] [20 1947]]	'vect__min_df': 1 , 'clf__n_estimators': 60
tfidf 1grams RandomForest	0.9991	0.9983	0.9929	0.9989	0.9992	[[20082 23] [14 1953]]	'vect__min_df': 40 , 'clf__n_estimators': 60
tfidf 2grams Logistic	0.999	0.9981	0.9853	0.9994	0.9995	[[20093 12] [29 1938]]	'vect__min_df': 1 , 'clf__C': 1000
count 2grams Logistic	0.9988	0.9979	0.9797	0.9997	0.9981	[[20098 7] [40 1927]]	'vect__min_df': 20 , 'clf__C': 10
tfidf 3grams Logistic	0.9988	0.9977	0.9776	0.9997	0.9996	[[20099 6] [44 1923]]	'vect__min_df': 1 , 'clf__C': 1000
count 2grams SVM	0.9988	0.9977	0.9792	0.9996	0.9986	[[20096 9] [41 1926]]	'vect__min_df': 5 , 'clf__gamma': 0.001 , 'clf__C': 100 , 'clf__kernel': 'rbf'
tfidf 3grams RandomForest	0.9987	0.9976	0.9751	0.9998	0.9991	[[20100 5] [49 1918]]	'vect__min_df': 5 , 'clf__n_estimators': 60
count 1grams SVM	0.9985	0.9973	0.9802	0.999	0.9979	[[20085 20] [39 1928]]	'vect__min_df': 5 , 'clf__gamma': 0.001 , 'clf__C': 100 , 'clf__kernel': 'rbf'
count 3grams RandomForest	0.9985	0.9972	0.972	0.9997	0.9991	[[20098 7] [55 1912]]	'vect__min_df': 5 , 'clf__n_estimators': 60
count 3grams Logistic	0.9984	0.9971	0.9705	0.9998	0.9981	[[20100 5] [58 1909]]	'vect__min_df': 2 , 'clf__C': 10
custom RandomForest	0.9983	0.9969	0.9781	0.9987	0.9986	[[20079 26] [43 1924]]	'n_estimators': 40
tfidf 2grams SVM	0.9982	0.9967	0.9675	0.9996	0.9991	[[20096 9] [64 1903]]	'vect__min_df': 5 , 'clf__gamma': 0.001 , 'clf__C': 100 ,

							'clf__kernel': 'rbf'
count 3grams SVM	0.9982	0.9967	0.969	0.9994	0.9976	[[20093 12] [61 1906]]	'vect__min_df': 5 , 'clf__gamma': 0.001 , 'clf__C': 100 , 'clf__kernel': 'rbf'
count 1grams Logistic	0.998	0.9964	0.9634	0.9996	0.9969	[[20097 8] [72 1895]]	'vect__min_df': 40 , 'clf__C': 10
tfidf 1grams Logistic	0.9979	0.9961	0.9675	0.9989	0.998	[[20083 22] [64 1903]]	'vect__min_df': 1 , 'clf__C': 1000
tfidf 3grams MultinomialNB	0.9978	0.9961	0.9766	0.998	0.9991	[[20064 41] [46 1921]]	'vect__min_df': 2
tfidf 2grams MultinomialNB	0.9976	0.9957	0.969	0.9983	0.998	[[20070 35] [61 1906]]	'vect__min_df': 2
tfidf 3grams SVM	0.9972	0.995	0.9471	0.9997	0.9981	[[20098 7] [104 1863]]	'vect__min_df': 5 , 'clf__gamma': 0.001 , 'clf__C': 100 , 'clf__kernel': 'rbf'
tfidf 1grams SVM	0.9968	0.9942	0.9492	0.9987	0.9964	[[20078 27] [100 1867]]	'vect__min_df': 5 , 'clf__gamma': 0.001 , 'clf__C': 100 , 'clf__kernel': 'rbf'
count 2grams MultinomialNB	0.9968	0.9941	0.9781	0.9957	0.9893	[[20018 87] [43 1924]]	'vect__min_df': 1
custom SVM	0.9968	0.9941	0.9853	0.995	0.9977	[[20004 101] [29 1938]]	'kernel': 'rbf' , 'C': 100 , 'gamma': 'auto'
count 3grams MultinomialNB	0.9965	0.9936	0.9817	0.9947	0.9977	[[19999 106] [36 1931]]	'vect__min_df': 1
custom MLPClassifier	0.996	0.9928	0.9548	0.9965	0.9976	[[20035 70] [89 1878]]	'hidden_layer_sizes': (300,200,150,150) , 'momentum': 0 , 'learning_rate': 'invscaling' , 'alpha': 0.01 , 'learning_rate_init':0.001
tfidf 1grams MultinomialNB	0.9951	0.9911	0.9273	0.9973	0.9927	[[20051 54] [143 1824]]	'vect__min_df': 1
count 1grams MultinomialNB	0.995	0.9909	0.9624	0.9937	0.9763	[[19979 126] [74 1893]]	'vect__min_df': 20
custom AdaBoostClassifier	0.9943	0.9897	0.9304	0.9955	0.9978	[[20014 91] [137 1830]]	'learning_rate': 1.0 , 'n_estimators': 100
custom DecisionTree	0.9939	0.9889	0.8958	0.998	0.9909	[[20065 40] [205 1762]]	'min_samples_split': 2
custom Logistic	0.9915	0.9845	0.8622	0.9964	0.9902	[[20033 72] [271 1696]]	'C': 100
custom SGD	0.9899	0.9816	0.8775	0.9918	0.9846	[[19940 165] [241 1726]]	'learning_rate':'optimal'
custom MultinomialNB	0.957	0.9243	0.9161	0.9251	0.9713	[[18600 1505] [165 1802]]	'alpha': 1.0

A.3 Contributions

Work	**Oskar**	**Filip**
Proposal	X	X
Project - Preparatory work	X	X
Project - Data wrangling	X	X
Project - Step3A: Feature engineering custom features	X	
Project - Step3B: Feature engineering bag-of-words		X
Project - Step3C: Feature space visualization		X
Project - Step4: Model selection		X
Project - Step5: Visualization		X
Project - Training models	X	X
Project - Demonstration web server	X	
Presentation powerpoint/prezi	X	X
Report - 1: Introduction	X	
Report - 2.1: Intrusion detection and...	X	
Report - 2.2: Web application firewalls	X	
Report - 2.3: Machine learning		X
Report - 3.1: Preparatory work	X	
Report - 3.2: Workflow	X	
Report - 3.3: Programming language		X
Report - 3.4: Documentation		X
Report - 3.5: Version control		X
Report - 4.1: Data wrangling	X	
Report - 4.2: Data cleaning	X	
Report - 4.3: Feature engineering	X	X
Report - 4.4: Model selection		X
Report - 4.5: Demonstration website	X	
Report - 5.1: Classifiers		X
Report - 5.2: Feature spaces	X	
Report - 6: Discussion	X	X
Report - 7: Conclusion	X	