Machine learning for cybersecurity

Artur Back de Luca
Department of Computer, Control, and Management Engineering
Università di Roma La Sapienza

November 10, 2019

Introduction

In the last decade, the world has witnessed a staggering growth in digital inclusion. The population percentage that now uses the internet has grown more than 70 percent in the last 10 years - countries like Namibia, for instance, watched this same percentage rise from 11.6% to 51% (The World Bank, 2019). This rapid adoption allowed societies to communicate seamlessly, spurring the dissemination of more representative narratives at social and political scales while also connecting people and experiences at an individual scale.

However, this massive gathering around the internet has also given new proportions to a long-standing challenge in technology. Cyberthreats are now one of the most urgent concerns in the world according to the Global Risks Report 2018 by the World Economic Forum (WEF), staying behind only extreme weather events and natural disasters, in terms of imminence (World Economic Forum, 2019). Among the variety of forms in which these attacks can manifest, the familiar malware abides among one of the most popular cyber threats, according to the ISACA report (Downs, 2019).

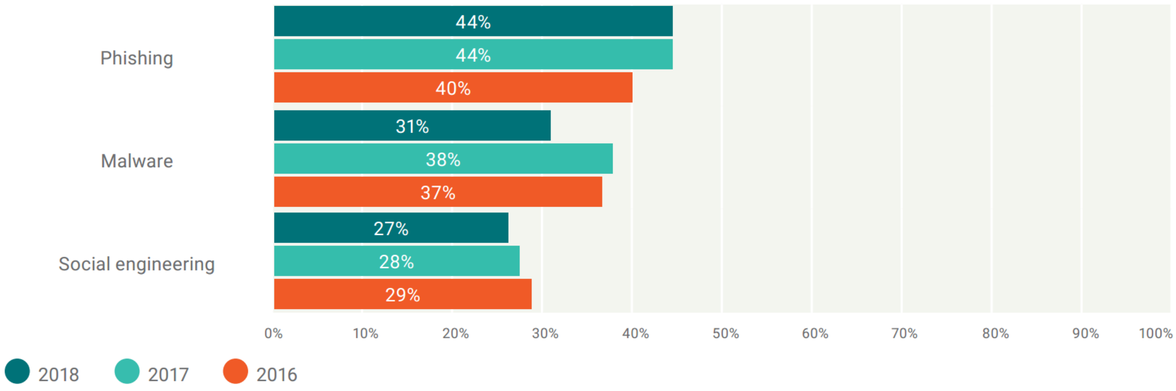

Figure 1: Comparison of the most popular attacks from 2016 to 2018. Source: (Downs, 2019)

Malware is short for *malicious software*, an umbrella term for programs designed to breach devices behind users' backs and cause damage that can range from displaying frequent and inconvenient ads to essentially blocking your personal data and demanding a ransom. According to a report by Verizon in 2015, every second, five attacks are made by malware, accounting for 170 million strikes across organizations (Verizon, 2016).

In this troublesome context, machine learning either can aggravate or mitigate this issue. As the WEF points out, new malware employs machine learning techniques to avoid detection. Conversely, at the same time, in malware analysis, techniques are being developed and implemented to better detect these threats (World Economic Forum, 2019). In general, the objective of malware analysis is to identify and typify certain structures that compose a malware, by dissecting its binary code and observing its behavior ((Zhou and Jiang, 2012)). This process can be segmented into two approaches:

Static analysis is the process of studying different sections of the binary file (post-compiled) without actually executing it. However, malware instructions are often unintelligible, so examiners attempt to revert the compiled code to a comprehensible language, although this is often cumbersome, as malware designers intentionally insert syntactic errors that hamper the decoding process. This technique is called *obfuscation* and is quite notorious among software developers as a way to prevent reverse engineering.

To avoid this predicament, *dynamic analysis* is employed, as it investigates the threat's behavior while it is being executed. This is performed in a controlled environment to innoculate operating systems from malware's harming behavior, allowing analysts to debug commands and check their subsequence consequences, thereby identifying how a particular attack takes place. Although dynamic analysis avoids typical obfuscation, as it doesn't decode the malware's binary code, it is susceptible to other evasive techniques, such as delaying the execution of payloads or checking for active debuggers.

To illustrate how machine learning can be employed in malware analysis, consider the task of disassembling a malicious program. To properly obtain the binary code in its intelligible form, it is crucial to know which compiler generated the executable program. Without this information, it is rather impractical to identify a correspondence between a set of incomprehensible commands and standard library functions. This problem is known as the *compiler provenance* (CP) problem, first presented by Rosenblum (Rosenblum et al., 2010), and its solution consists of a mapping between the *idioms* of the binary code, i.e. short sequences of instructions, and the correspondent source compiler.

Additionally, a similar problem concerns identifying the *optimization degree* (OD) employed during compilation. These assure that the code generated can be executed faster, but it is also can pose an obstacle to the disassembling task. Considering this setting, an inference model can determine these relationships, learning from observations comprising of idioms, their source compiler and their degree of optimization. The dataset used in this example is an excerpt from (Massarelli et al., 2019) and consists of a variable set of instructions, two possible degrees of optimization (`high` or `low`), and three compiler provenances: `icc` (icc-17 and icc-19), `gcc` (gcc-3.4, gcc-4.7, gcc-4.8, gcc-4.9 and gcc-5.0) and `clang` (clang-3.8, clang-3.9, clang-4.0, clang-5.0) - containing 30.000 observations. The subsequent sections of this document further discuss the chosen strategies to solve the given task.

Description of the procedure

The systematic approach taken to tackle this problem is composed of several best practices which, to some extent, are applicable to a wide array of different data science projects. The results showed in the following sections were obtained by using `python-3.6.8`, as well as other specifications which are further described in (Back de Luca, 2019)

Initial setting

Metric selection

The metrics used for the evaluation have to compromise between the interpretability of the results as well as the validity of its assertions under the given problem's circumstances. A well-known example of inadequate metric assertion is the usage accuracy in problems with great class imbalance. For the current task, the distribution of classes in compiler provenance and optimization degree are considered balanced (1:1:1) and marginally imbalanced (1:3), respectively, according to the classification presented by (He and Ma, 2013). For this reason, accuracy can and will be used for the sake of clarity. Moreover, to avoid overlooking potentially imbalanced sets of observations during cross-validation, which will be later discussed, confusion matrices will also be employed.

		Predicted label	
		Positive	Negative
True label	Positive	TP *True Positives*	FP *False Positives*
	Negative	FN *False Negatives*	TN *True Negatives*

Table 1: Illustration of a confusion matrix for a binary classification problem

There are several ways to try to render the results of a confusion matrix in a single value. The F-measure, for instance, is a harmonic weighted mean between *precision* and *recall*, two proportions derived from the matrix.

$$F_\beta = \frac{2}{\alpha.\text{recall}^{-1} + (1-\alpha).\text{precision}^{-1}}$$

where

$$\text{precision} = \frac{TP}{TP+FP}, \quad \text{recall} = \frac{TP}{TP+FN} \text{ and } \alpha = \frac{1}{1+\beta^2}$$

The score ranges from 0 (worst) to 1 (best) and if $\beta = 1$, i.e. $\alpha = 0.5$, the measure is famously known as the *F1-score*, where both precision and recall have equal importance. The F-measure stems from Rijsbergen's *Effectiveness* and (Sasaki, 2007) provides a quite comprehensive investigation on the topic.

Furthermore, a lesser-known yet effective measure, the Matthews Correlation Coefficient (MCC), also known as the phi coefficient of a confusion matrix, measures the degree of association between two variables, or in our case, the predictions and the true values. These two variables are considered positively associated if most of the data falls along the main diagonal. Likewise, as more data gets distributed along the antidiagonal, the measure decreases towards 0, signalizing a random prediction, or even further towards -1, indicating a completely negative association between variables. Because of this, as discussed in (Powers, 2011), MCC is useful as it provides an understanding of the direction and intensity of the variables' relationship. Furthermore, as indicated in (Bekkar et al., 2013), MCC is considered one of the best singular assessment metrics, and for these reasons, it will chosen to compare the performance between the different approaches tested.

Model assessment

Once the metrics have been selected, an important next step to consider is how these will be applied. There is an extensive general understanding of what methods are more effective and unbiased to evaluate a model, which will not be further expanded here (for a comprehensive discussion on the topic, see (Hastie et al., 2009)). However, it is important to mention that depending on the type of evaluation method that is employed, the performance estimate can be more or less subject to uncertainties and biases. For this reason, it suffices to mention that since a large quantity of data is available, cross-validation with a high number of folds has been chosen, as it will evaluate each setting multiple times with different training and testing sets, in attempt to abate the inherent uncertainties of assessing a model with a limited quantity of observations.

Additionally, to build the following confusion matrices, the dataset has been split into two parts: a training set, accounting for 2/3 of the total dataset and a validation set with the remaining observations. All these operations were performed with a random state seed, arbitrarily set to 42, that is fed to the pseudorandom generators to ensure reproducibility.

Baseline setting

Finally, to suitably compare the different approaches, a baseline setting is needed, and this is comprised of a model and a feature representation tactic. It is worth mentioning that the same baseline approach will be used for both problems (CP and OD). First, the approach chosen to extract features consists of simply counting the number words of each observation, as well as counting how many of those are distinct. This tactic disregards any hypothesis of general ordering among instructions and only accounts for the assumption that different compilers and optimizers may generate binary code of different lengths.

Secondly, the baseline model should be simple and fast, without too many configurable parameters. Thus, accordingly, the Naive Bayes algorithm is oftentimes employed. The model, however, requires the specification of the probability distribution of the input variables. Normally, for text classification, the multinomial distribution is used as it models the frequency of observations of a particular instance. Additionally, the Naive Bayes classifier applies a sort of regularization to the extent that it assumes that all the features all independent within each class, that is, the within-class covariance matrix is diagonal. This, in turn, makes the model much more simple than its counterparts, such as the LDA, considerably improper for this type of analysis, as the model greatly increases with the number of features, a typical consequence of text classification (Hastie et al., 2009).

In this initial setting, the average accuracy for CP is 35.20% and MCC is 0.04. Conversely, for the OD problem the accuracy is 59.35%, the MCC is 0.07, and the confusion tables for both tasks are the following.

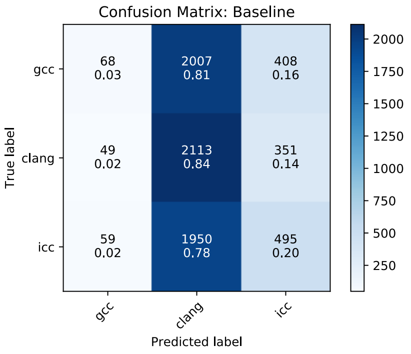

Figure 2: Confusion matrix for the baseline setting - compiler provenance

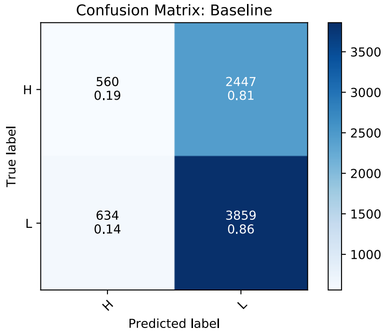

Figure 3: Confusion matrix for the baseline setting - optimization degree

Feature extraction

The first phase of the project consists of defining better alternatives to represent the input data. For both problems, the data is identical: the list of commands of the binary files. This representation, however, is not suitable for any model as every observation is very distinct from each other, including examples from the same class. This happens essentially because the programs compiled may differ in their nature, thus explaining the variability.

The strategy in this problem is to try to capture the similarities among the examples, hopefully also depicting the distinctions between observations from different classes. However, the number of distinct keywords in the dataset poses a predicament to this type of strategy. One way to alleviate this problem is to exclude the keywords that could be potentially irrelevant to the task. For instance, by analyzing the dataset, it is easy to identify common mnemonics. On the other hand, there are certain positional arguments that follow initial commands that possess great variability. It is very likely that these are elements particular to each type of task, and therefore could be eliminated.

Another strategy that can be combined with the aforementioned consists of transforming the variable input commands in a table containing certain mnemonics. This, in turn, can be further subdivided into different approaches:

1. Create a table with a *one-hot encoding*, composed of binary data that represents if a certain keyword is found in that particular set of instructions or not

2. Create a frequency table that counts how many times a certain mnemonic is found in that set of instructions

    (a) These mnemonics can be further expanded considering not only a single word (or a unigram) but the combination of subsequent keywords as well (such as bigrams, trigrams and so on).

From the above mentioned strategies, eight distinct approaches are described below:

1. `Mnemonic check`: create a one-hot encoding table representing the presence of a single keyword.

2. `tf-idf`: frequency-inverse document frequency table, in which each value represents the relative frequency of a particular keyword in the set of instructions considering its frequency in all other observations.

3. `Unigrams`: the absolute frequency for a single keyword (unigram) in a set of instructions.

4. `Unigrams-trigrams`: the absolute frequency for unigrams, bigrams and trigrams in a set of instructions.

5. `Unigrams-trigrams (two words)`: the absolute frequency for unigrams, bigrams and trigrams not discarding the first parameter in a set of instructions.

6. `Unigrams-trigrams (two words) - simplified`: the absolute frequency for unigrams, bigrams and trigrams not discarding the first parameter in a set of instructions but considering only the three first and last instructions.

7. `Fourgrams (two words)`: the absolute frequency for fourgrams in a set of instructions.

8. `Fourgrams (two words) - simplified`: the absolute frequency for fourgrams in a set of instructions keeping the 15000 most frequent fourgrams for each class.[1]

[1] Of a total of approximately 3.7 million features

Preliminary results

The approaches aforementioned are here presented for both tasks in question. All the confusion matrices are exposed in the Appendix section

Compiler provenance

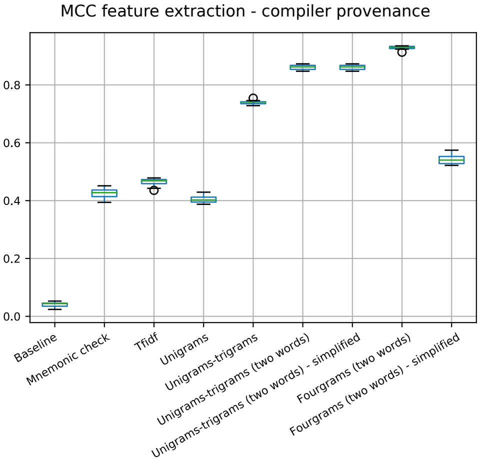

Figure 4: Boxplot for the feature extraction strategies in the compiler provenance task

						Unigrams	Unigrams		
		Mnemonic			Unigrams-	-trigrams	-trigrams		Fourgrams -
	Baseline	check	tf-idf	Unigrams	trigrams	(2w)	(2w) - simp.	*Fourgrams*	simp.
Accuracy	35.21%±0.4%	60.54%±1.0%	63.85%±0.9%	82.45%±0.8%	90.71%±0.4%	90.71%±0.6%	90.71%±0.6%	*95.21%±0.4%*	69.25%±0.1%
MCC	0.04 ±0.01	0.42±0.02	0.46±0.01	0.40±0.01	0.74±0.01	0.86±0.01	0.86±0.01	*0.93±0.01*	0.54±0.02

Table 2: Feature extraction results for the compiler provenance task

Optimization degree

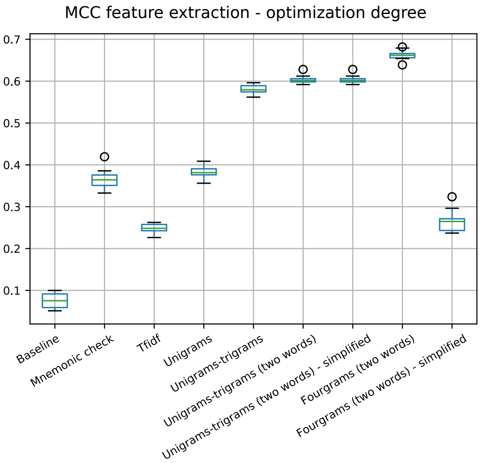

Figure 5: Boxplot for the feature extraction strategies in the optimization degree task

						Unigrams	Unigrams		
		Mnemonic			Unigrams-	-trigrams	-trigrams		Fourgrams -
	Baseline	check	tf-idf	Unigrams	trigrams	(2w)	(2w) - simp.	*Fourgrams*	simp.
Accuracy	59.35%±0.5%	69.93%±1.0%	65.07%±0.3%	69.87%±0.6%	78.31%±0.4%	79.09%±0.5%	79.09%±0.5%	*82.13%±0.6%*	65.16%±0.9%
MCC	0.07±0.02	0.37±0.02	0.24±0.01	0.38±0.01	0.58±0.01	0.60±0.01	0.60±0.01	*0.66±0.01*	0.26±0.03

Table 3: Feature extraction results for the optimization degree task

It is noticeable by comparing both results that under the same circumstances, the compiler provenance task presents greater results than optimization degree, except for the Baseline setting. The reasons for this remain uncertain, although it is likely that this is due to the inherent structure of the OD problem, namely that analyzing the frequency of words may not be as fruitful as in its counterpart. Moreover, another plausible reason for this discrepancy is the class imbalance present in the binary case, that despite not initially posing a serious threat to the task, can hamper performance.

Additionally, the two approaches to reduce the complexity of the model produce distinct effects. In the first one, by keeping the first and last 3 mnemonics seems to produce no different results than considering all keywords. Yet it doesn't increase the performance of the estimator, at least it significantly reduces the complexity of the dataset representation. Conversely, the results of the Fourgrams simplification indicates a great hindrance to performance. In spite of cutting down a great portion of the features (3.7 million features to 15 thousand), this setting yields much poorer results than those generated with the integral set of features, which, in turn, demonstrates the complexity of the task. And based on the results obtained, the feature extraction technique that is used in the subsequent sections the *Fourgrams* approach, as it yielded the best results so far.

Model selection

Having established the dataset representation, the next step in the process is to define the most suitable model for each task, namely the one that yields the best result. Among the plethora of models for the given problems, only three models are compared, primarily for their simplicity and robustness.

1. Naive Bayes classifier with multinomial feature distribution (*Baseline setting*)
2. Support Vector Machine with linear kernel (SVM-linear)
3. Decision Tree (DT)

Apart from the baseline, which was already discussed, the Support Vector Machine with linear kernel was selected by its capacity to effectively divide the decision space and different kernels were also considered due to the potential non-linearity of the problem. However, because of the high-dimensionality encountered, these would later show to be intractable with the computational facilities available. Finally, the Decision Tree classifier was also chosen mainly for its straightforward principle and resulting interpretability.

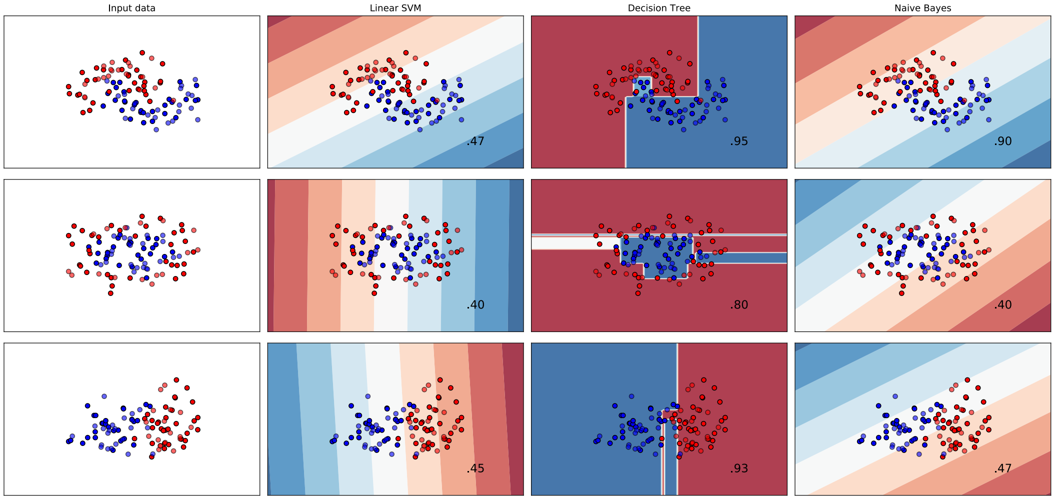

Figure 6: Illustrarion of the decision boundaries for different models and binary classification tasks

All of the models implemented were taken from the `scikit-learn` (version 0.21.3), a machine learning for Python. Consequentially, all the parameters defined for each model was taken by default, namely:

1. Naive Bayes ($\alpha = 1.0$, `uniform and learnable priors`)
2. SVM-linear (`C=1.0, L2 penalty, squared hinge loss, and uniform class weights`)
3. DT (`Gini split criterion, minimum of 2 samples to split, max depth undefined, best split strategy`)

Preliminary results

The experiments for the model selection stage are here presented for both tasks in question. All the confusion matrices are exposed in the Appendix section.

Compiler provenance

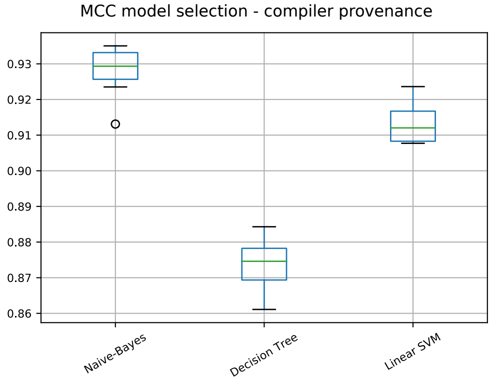

Figure 7: Boxplot for the feature extraction strategies in the compiler provenance task

	Naive Bayes	Linear SVM	Decision Tree
Accuracy	95.21%±0.4%	94.21%±0.4%	91.57%±0.4%
MCC	0.93 ±0.01	0.91±0.01	0.87±0.01

Table 4: Model selection results for the compiler provenance task

Optimization degree

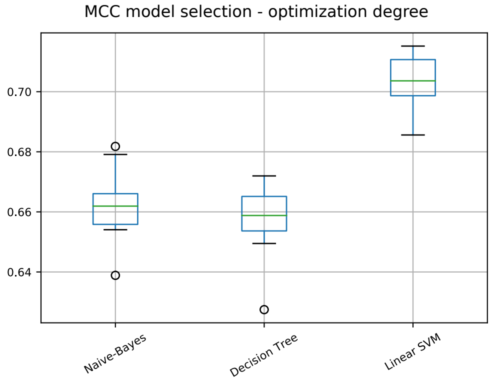

Figure 8: Boxplot for the feature extraction strategies in the optimization degree task

	Naive Bayes	Linear SVM	Decision Tree
Accuracy	82.13%±0.6%	85.80%±0.4%	83.50%±0.6%
MCC	0.66 ±0.01	0.70±0.01	0.65±0.01

Table 5: Feature extraction results for the optimization degree task

It is possible to identify a divergence in terms of performance for both tasks. As in the feature extraction step, the optimization degree has presented slightly worse results than its counterpart, but here the best models diverge: in CP the model with the best performance was the baseline Naive Bayes with multinomial distribution, as for the OD, the Linear SVM was the best model found. Thus, for the parameter tunning section, the tasks follow different directions.

Parameter tunning

Finally, having selected the most suitable models, the next steps comprise of finding the best configuration of parameters for each task. In this stage, we select the some of the model parameters and their correspondent ranges and apply a grid-search to generate all the possible configurations. However, since the combination of parameters generates several instances of the problem, the number of cross-validations was reduced from 10 to 5, conforming the computations to the available processing capacity.

Compiler provenance

The model selected for the compiler provenance case was the Multinomial Naive Bayes model. By design, the model is parameterized only by one single argument, the smoothing factor α, which increases the frequency of every feature to generate a probability distribution less noisy if this had been generated solely based on the observations. From this, we establish three acceptable values for alpha: 0 (no smoothing), 1 and 1.5 and from it, and obtain the following results.

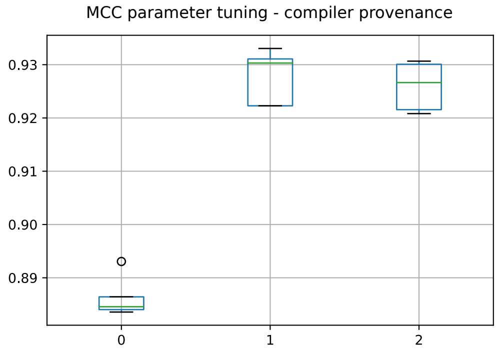

Figure 9: Boxplot for the parameter tuning results for the compiler provenance task

	0. $\alpha = 0$	1. $\alpha = 1$	2. $\alpha = 1.5$
Accuracy	92.41%±0.2%	95.18%±0.3%	95.05%±0.3%
MCC	0.89 ±0.00	0.93±0.00	0.93±0.00

Table 6: Parameter tuning results for the compiler provenance task

The two configurations $\alpha = 1$ and $\alpha = 1.5$ yielded very similar results while the strategy of ignoring the smoothing factor has generated poorer results. Based on the results obtained, and considering Accuracy as tiebreak, we select alpha with the value of 1 and obtain the following final confusion matrix.

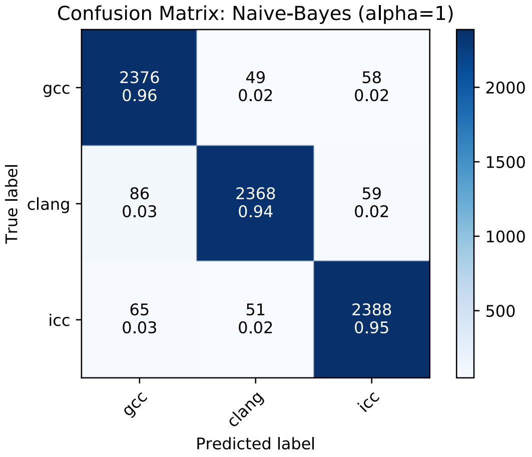

Figure 10: Confusion matrix of the conclusive setting for the compiler provenance task

Optimization degree

For the optimization degree task, the model selected (SVM-linear) has numerous parameters to be adjusted. This, in turn, complicates parameter optimization as complexity exponentially increases with the number of parameters considered as well as with the increase discretization of their range of values. In order to avoid these issues, only two possible values of a small portion of parameters were considered. The selected parameters to tune are the following:

- *Penalty parameter C:* the parameter dictates the contribution of the regularization factor to the total cost function. The higher the penalty parameter the more influence the regularization will have. The values tested in this case are $C = 1$ and $C = 10$
- *Class weight*: since the distribution of observations is slightly uneven, class weight can penalize more the misclassifications of the underrepresented class. The values tested are an even setting of weights 1:1 and an uneven setting favoring the least represented class 1:1.5.

Note that during this experiment, other parameters were also considered, such as the type of penalization employed (L1-norm or L2-norm). However, in this setting, where the number of features is much greater than the number of observations, the determination of the support vectors comprises of solving the dual problem, which in turn makes the use of L1-norm unfeasible. So the cases in which L1-norm was used were discarded due to this unfeasibility. Additionally - for the same reasons as in the compiler provenance task - the number of splits performed in the cross-validation stage is 5, and from this setting the following results are obtained.

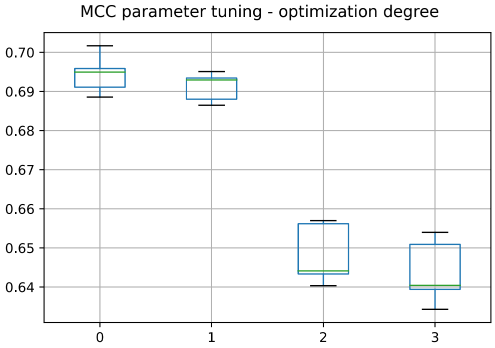

Figure 11: Boxplot for the parameter tuning results for the optimization degree task

	0. (C=1, 1:1)	1.(C=1, 1:1.5)	2.(C=10, 1:1)	3.(C=10, 1:1.5)
Accuracy	85.35%±0.2%	85.19%±0.2%	83.07%±0.3%	82.86%±0.4%
MCC	0.69 ±0.00	0.69±0.00	0.65±0.01	0.65±0.01

Table 7: Parameter tuning results for the optimization degree task

It is conspicuous the penalization factor exerts a much greater influence than the weightening factor. Nevertheless, the cases in which uneven weights were used produced slightly lower results than the even ones. So, according to the experiments shown, the most suitable model under these circumstances is the one with parameters $C = 1$ and equal weights, which yields the following confusion matrix.

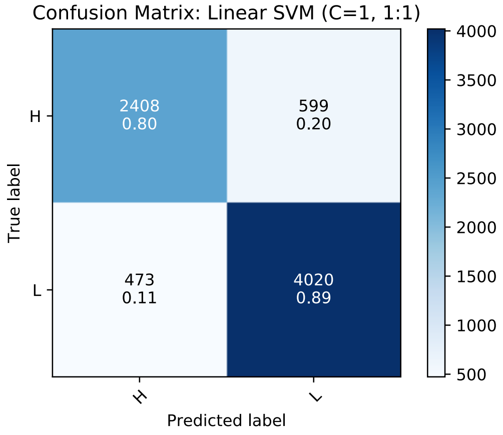

Figure 12: Confusion matrix of the conclusive setting of the optimization degree task

Conclusion

It becomes evident that both problems have presented very distinct performances, despite sharing similar inputs and preprocessing steps. Naturally, the inherent structure of both task is distinct, and so the optimization degree problem could have benefited from alternative approaches than those developed in the present work. Strategies for feature extraction, such as using ngrams of different sizes, or by applying different models for the inference stage, like ensemble methods (e.g. random forests, adaptive boosting) and different linear models such as logistic regression could perhaps be more suited for the task in hand. Yet one must carefully consider the complexity of the methods employed due to the already cumbersome feature space to avoid stagnant performances.

Additionally, despite commonly regarded as merely a baseline approach that is generally surpassed, the Naive Bayes model has yielded outstanding results. This illustrates the power of generative classifiers and their capacity to model the joint distribution when bolstered with a good quantity of data.

Lastly, the cases here discussed display a very constructive application in machine learning that can potentially generate a positive impact at a large scale. By deepening research on malware analysis, more sophisticated approaches may eventually arise, not only tailored to effectively tipify malware but also to surpass the current hindrances placed by perpetrators. Therefore, it imperative to stimulate and further expand the current research on the topic.

References

Back de Luca, A. (2019). artur-deluca/compiler_provenance. original-date: 2019-11-08T13:53:28Z.

Bekkar, M., Djemaa, H. K., and Alitouche, T. A. (2013). Evaluation Measures for Models Assessment over Imbalanced Data Sets. *Journal of Information Engineering and Applications*, 3(10):27–38–38.

Downs, T. F. (2019). State of cybersecurity 2019. Technical report, ISACA.

Hastie, T., Tibshirani, R., and Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction*. Springer series in statistics. Springer, New York, NY, 2nd ed edition.

He, H. and Ma, Y. (2013). *Imbalanced learning foundations, algorithms, and applications*. IEEE Press ; Wiley, Piscataway, NJ; Hoboken, New Jersey. OCLC: 880822636.

Massarelli, L., Di Luna, G. A., Petroni, F., Querzoni, L., and Baldoni, R. (2019). Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. In *Proceedings 2019 Workshop on Binary Analysis Research*, San Diego, CA. Internet Society.

Powers, D. M. (2011). Evaluation: from Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation.

Rosenblum, N. E., Miller, B. P., and Zhu, X. (2010). Extracting Compiler Provenance from Program Binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 21–28, New York, NY, USA. ACM. event-place: Toronto, Ontario, Canada.

Sasaki, Y. (2007). The truth of the F-measure.

The World Bank (2019). Individuals using the Internet (% of population).

Verizon (2016). Data Breach Investigations Report 2016. Technical report, Verizon.

World Economic Forum (2019). *Global risks 2019: insight report*. OCLC: 1099890423.

Zhou, Y. and Jiang, X. (2012). Dissecting Android Malware: Characterization and Evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109, San Francisco, CA, USA. IEEE.

Appendix

Confusion matrices for the feature extraction stage

Compiler provenance

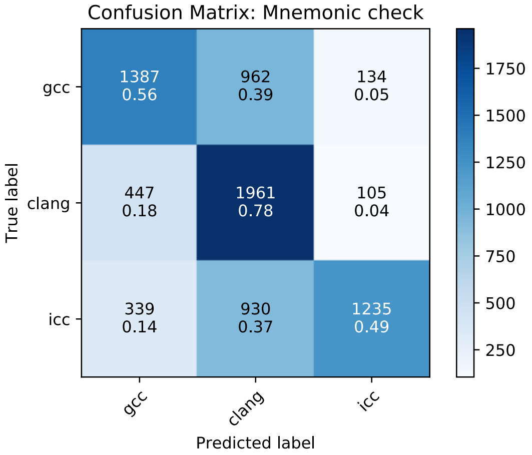

Figure 13: Confusion matrix for the mnemonic one hot encoding strategy in the compiler provenance task

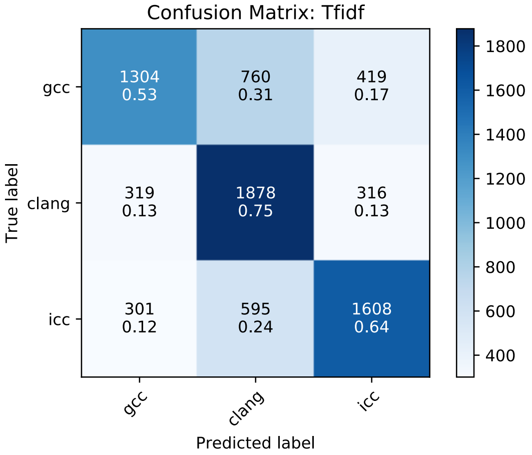

Figure 14: Confusion matrix for the tf-idf strategy in the compiler provenance task

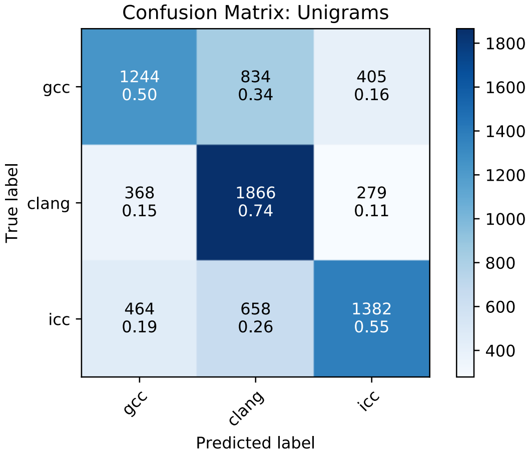

Figure 15: Confusion matrix for the simplified unigrams strategy in the compiler provenance task

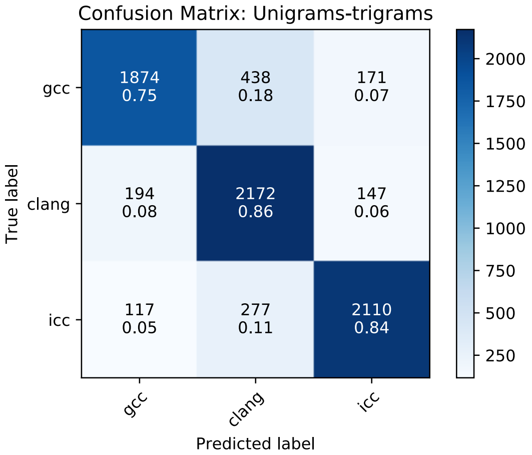

Figure 16: Confusion matrix for the simplified uni/bi/trigrams strategy with one word in the compiler provenance task

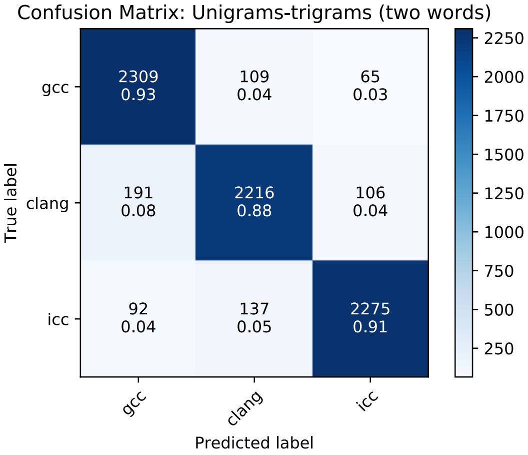

Figure 17: Confusion matrix for the uni/bi/trigrams strategy with two words in the compiler provenance task

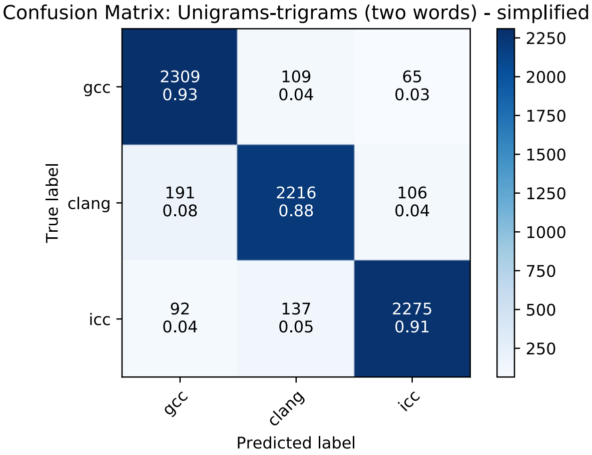

Figure 18: Confusion matrix for the simplified uni/bi/trigrams strategy with two words in the compiler provenance task

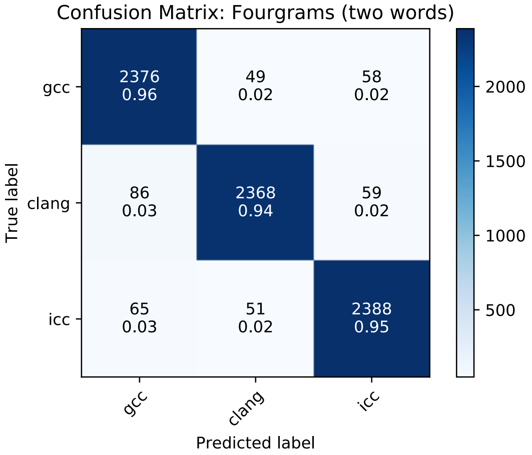

Figure 19: Confusion matrix for the fougrams strategy in the compiler provenance task

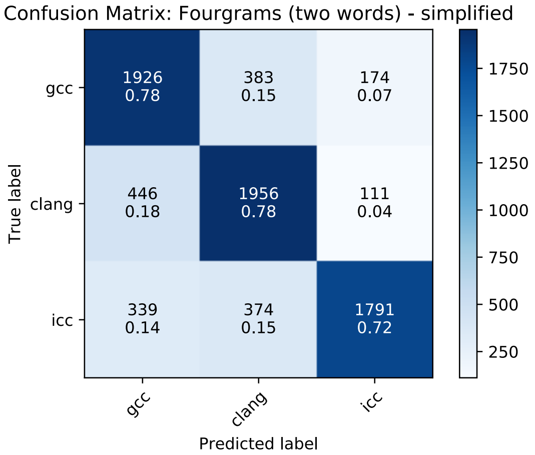

Figure 20: Confusion matrix for the simplified fougrams strategy in the compiler provenance task

Optimization degree

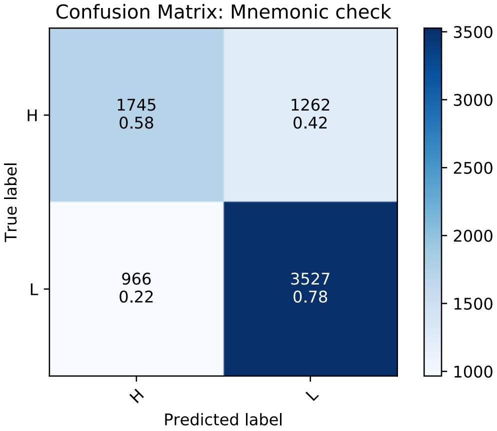

Figure 21: Confusion matrix for the mnemonic one hot encoding strategy in the optimization degree task

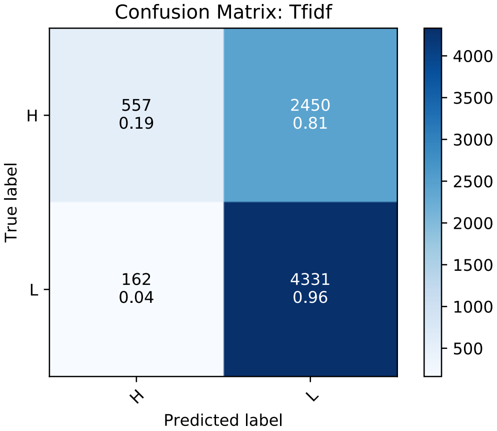

Figure 22: Confusion matrix for the tf-idf strategy in the optimization degree task

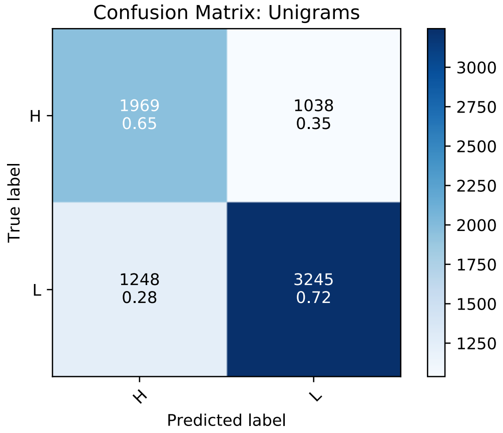

Figure 23: Confusion matrix for the simplified unigrams strategy in the optimization degree task

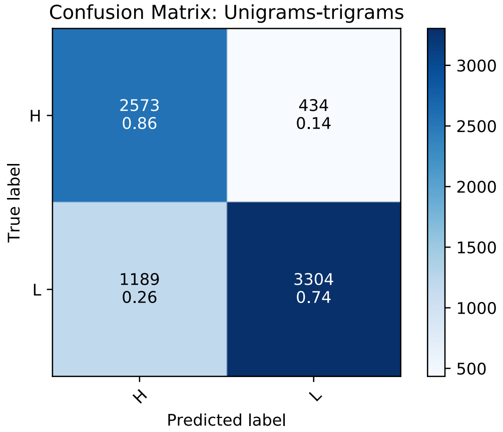

Figure 24: Confusion matrix for the simplified uni/bi/trigrams strategy with one word in the optimization degree task

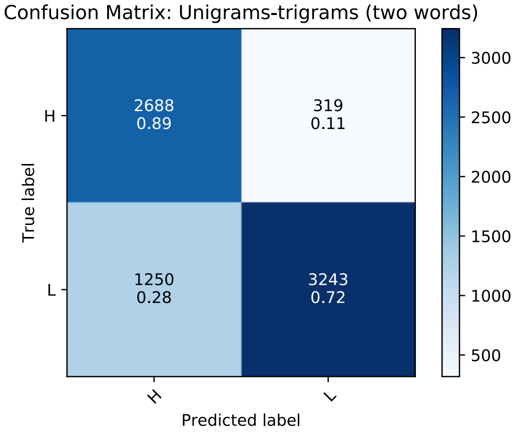

Figure 25: Confusion matrix for the uni/bi/trigrams strategy with two words in the optimization degree task

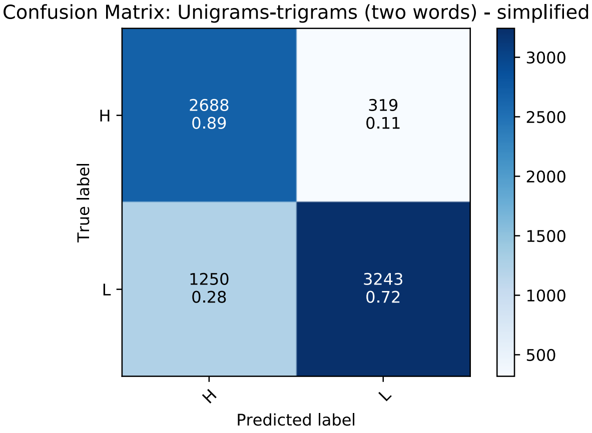

Figure 26: Confusion matrix for the simplified uni/bi/trigrams strategy with two words in the optimization degree task

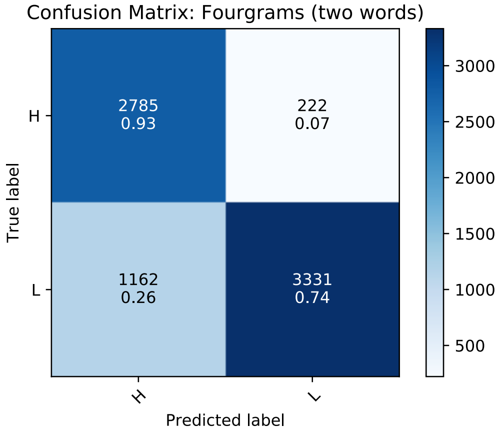

Figure 27: Confusion matrix for the fougrams strategy in the optimization degree task

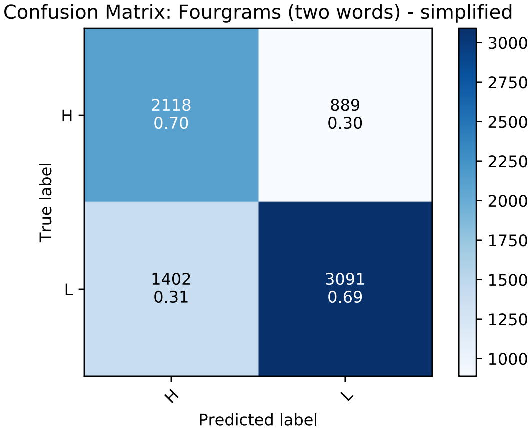

Figure 28: Confusion matrix for the simplified fougrams strategy in the optimization degree task

Confusion matrices for the model selection stage

Compiler provenance

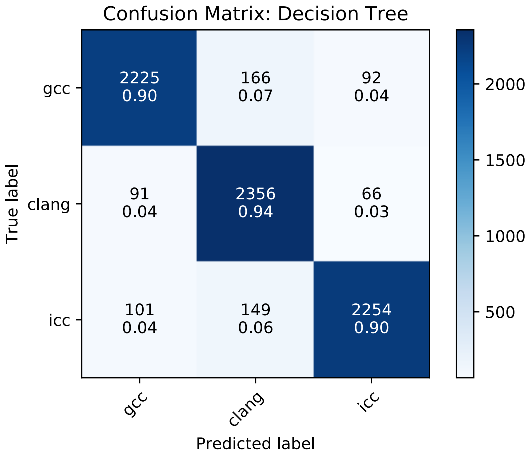

Figure 29: Confusion matrix of a Decision Tree for the compiler provenance task

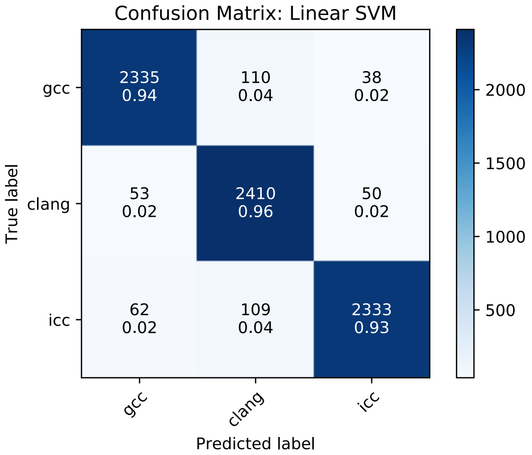

Figure 30: Confusion matrix of a Linear Support Vector Machine for the compiler provenance task

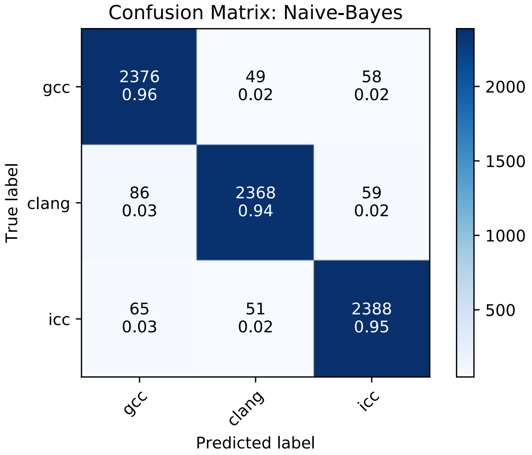

Figure 31: Confusion matrix of a Naive-Bayes classifier for the compiler provenance task

Optimization degree

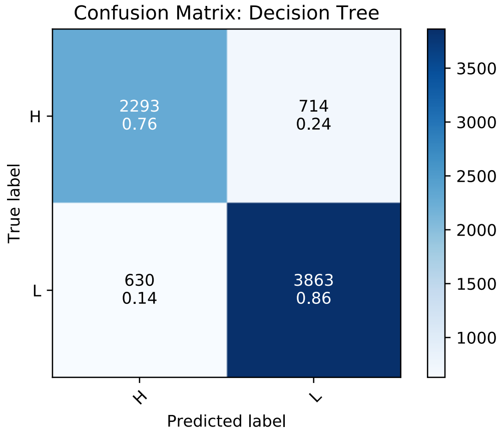

Figure 32: Confusion matrix of a Decision Tree for the optimization degree task

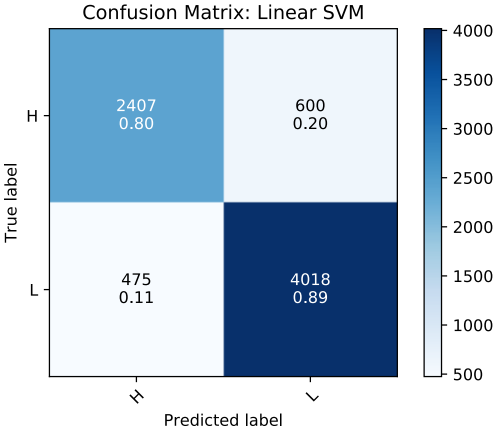

Figure 33: Confusion matrix of a Linear Support Vector Machine for the optimization degree task

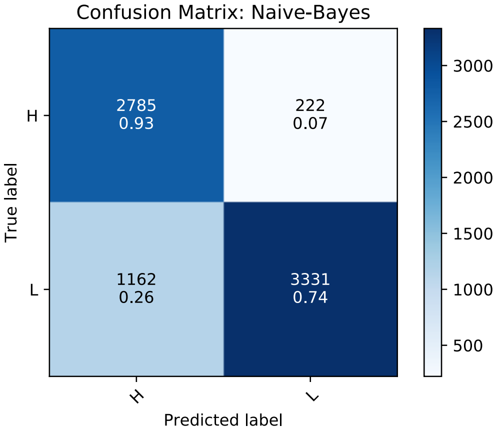

Figure 34: Confusion matrix of a Naive-Bayes classifier for the optimization degree task