University of Bari "Aldo Moro"
Department of Computer Science

Master's Degree in Computer Science
Curriculum Studies in Artificial Intelligence

Malicious URL detection using Machine Learning techniques and Apache Spark

Big data course

Team members:
Simone GRAMEGNA

Professor:
Michelangelo CECI

Academic Year 2023/2024

Abstract

The rapid increase in malicious activities on the internet has necessitated the development of effective methods to identify and classify malicious URLs. In this project, stating from a URLs dataset we extract string-based features (such as number of dots, number of subdomains, etc.) to detect malicious URLS using different models widely used in machine learning and in subsequent experiments we compare the performances of those models to choose the best one and the best parameters setting. We also propose an original and innovative approach, inspired by vectorization techniques typical of Natural Language Processing and applied to url's dataset. The peculiarity of this work is that it is based only on intrinsic characteristics that can only be deduced from the URL itself without taking into account either characteristics of the associated website or characteristics relating to network traffic, such features are not always available and are time-dependent and, so, they may bring to inconsistent results.

Contents

Chapter 1

Introduction

Every day, millions of people connect to the internet to search, access online services, study, work or simply for entertainment, it is estimated that over 5.7 million searches are made on Google every minute [1] Surfing the web exposes users not only to a great deal of information, but potential risks from malicious sites including:

- Spreading malware: Malicious sites can host and distribute malware such as viruses, worms, Trojans, ransomware and spyware. These malicious programmes can infect the user's device, compromise privacy, cause damage to files and even allow unauthorised access to the system.

- Phishing: Malicious sites can be used to conduct phishing attacks. These sites can be designed to look like legitimate websites, such as online banking services or payment platforms, in order to trick users into entering sensitive information such as user names, passwords, financial information or personal details. This information can then be used for fraud or identity theft.

- Personal Data Theft: Malicious websites may attempt to collect personal information, such as identification data, addresses, credit card numbers, or login details. This data can be sold or used for illegal activities, such as identity theft or hacking into accounts.

- Redirection to Malicious Sites: Some malicious websites may redirect users to even more dangerous sites, where they can be exposed to additional threats such as automatic malware downloads or scam attempts.

- Spread of Harmful or Illegal Content: Malicious websites can host harmful content, such as illegal pornography, violence, defamation, or hate speech.

Accessing such content can be harmful to the user and may also violate existing laws.

- Utilization of Computer Resources: Some malicious websites may exploit the user's device for malicious purposes. For example, they can use the computer's processing power for cryptocurrency mining without the user's consent, causing device slowdowns or overheating.

- Online Deception and Scams: Malicious websites can employ deceptive tactics, such as fake offers or counterfeit promotions, to obtain personal or financial information from users. They may also deceive users into making payments for products or services that are never delivered.

To cope with these risks, there are sites and online services that can help you know whether a domain is potentially malicious or not, services based on huge databases containing malicious domains or domains that have given users problems. It was from these databases that a dataset containing a list of malicious domains was created and published on Kaggle in 2021, available at the following link: Malicious URLs dataset containing more than 641.000 URLs. As already mentioned in the abstract, the aim of this project is to compare different classification models widely known and used in machine learning to detect malicious URLs using only the features we can obtain from the URL without taking into account either the characteristics of the associated website or the network traffic obtained by visiting the domain. We set ourselves this goal because any user with a minimum of surfing experience can recognise a potentially malicious URL from a safe URL at a glance without the use of additional tools, so we wanted to transfer this common experience into machine learning models by selecting the best model.

Chapter 2

Project structure

The case study consists of a jupyter notebook containing all the code written to perform the entire url classification process. To run the notebook the requirements are:

- Apache Spark installed on the device
- Python 3.9 installed on the device
- Python libraries (*pip install -r requirements.txt*)

Once all requirements are met, you can run the notebook by giving the command from the terminal: *jupyter-notebook* which will automatically open the browser with the jupyter environment.

Figure 2.1: jupyter environment

Chapter 3

Data exploration

As a first step of the case study there is the creation of spark session

```
spark = SparkSession.builder.appName('URLs_mining').getOrCreate()
spark
```

Once the spark session has been initialised, I proceed with the data exploration phase.

As a first step, I show the original dataset in this way:

```
data = spark.read.csv("malicious_urls.csv", header=True, inferSchema=True)
data.show()
```

Which outputs:

```
+--------------------+----------+
|                 url|      type|
+--------------------+----------+
|    br-icloud.com.br|  phishing|
|mp3raid.com/music...|    benign|
|bopsecrets.org/re...|    benign|
|http://www.garage...|defacement|
|http://adventure-...|defacement|
|http://buzzfil.ne...|    benign|
|espn.go.com/nba/p...|    benign|
|yourbittorrent.co...|    benign|
|http://www.pashmi...|defacement|
|allmusic.com/albu...|    benign|
|corporationwiki.c...|    benign|
|http://www.ikenmi...|defacement|
|myspace.com/video...|    benign|
|http://www.lebens...|defacement|
|http://www.szabad...|defacement|
|http://larcadelca...|defacement|
|quickfacts.census...|    benign|
|nugget.ca/Article...|    benign|
|uk.linkedin.com/p...|    benign|
|http://www.vnic.c...|defacement|
+--------------------+----------+
only showing top 20 rows
```

Figure 3.1: Malicious URLs dataset (top 20 rows)

And them I proceed to count all the different types of URLs in the dataset using:

```
data.groupBy("type").count().show()
```

which outputs:

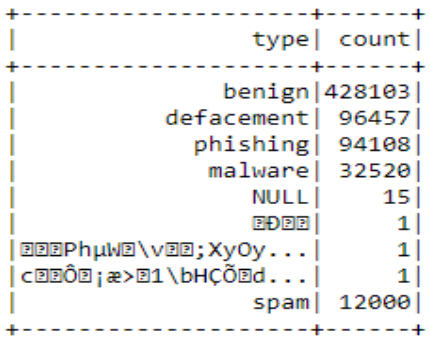


Figure 3.2: Count of URL types

Chapter 4

Data pre-processing

4.1 Preliminary Preprocessing

The data preprocessing phase is crucial to transform the initial dataset into the feature dataset used to train subsequently defined models.
First, I remove the null values of the URLs, on which it is impossible to extract features, and show again the count of the various types of urls present

```
data = data.dropDuplicates(["url"])
data = data.na.drop()

data.groupBy("type").count().show()
```

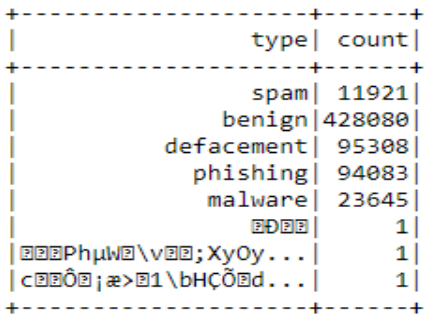
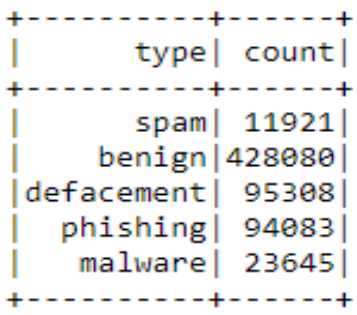


Figure 4.1: Count of URL types after removing NULL values

Then I proceed to filter URLs whose type is written in ASCII

```
data = data.filter(col("type").rlike(r'^[ -~]+$'))
data.groupBy("type").count().show()
```

and I get this:



type	count
spam	11921
benign	428080
defacement	95308
phishing	94083
malware	23645

Figure 4.2: Count of URL types after filtering

I visualise the various URL types in a bar graph using the matplotlib library and converting the spark dataframe to a pandas dataframe

```
pd_data = data.toPandas()

count = pd_data.type.value_counts()
bar_labels = [count.index[i].capitalize()+' (n. ' + str(count[i]) + ')'
    for i in range(len(count.index))]
bars = plt.bar(count.index, height=count, label=bar_labels,
    color=['#8fce00', '#0092ff', '#ff8200', '#ff1100', '#ffcd34'])

for rect in bars:
    height = rect.get_height()
    plt.text(rect.get_x() + rect.get_width() / 2.0, height, f'{height:.0f}
        ({height/len(pd_data)*100:.2f}%)', ha='center', va='bottom')

plt.xlabel('LABELS', labelpad=20)
plt.ylabel('COUNT', labelpad=20)
plt.title('EXPLORING DATASET')
plt.show()
```

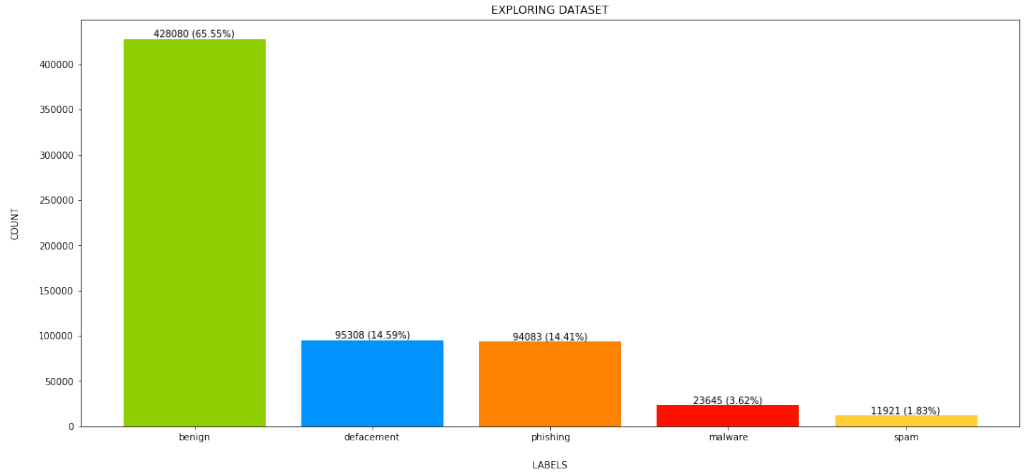


Figure 4.3: Types of URLs

For the purposes of the binary classification task, I create a new column called *url_type* in which I have the following mapping:

- Benign URLs: 0
- Defacement, phsishing, malware, spam URLs: 1

using the following code:

```
data = data.withColumn("url_type", when(data["type"].contains("benign"),
    0).otherwise(1))
data.show()
```

The features extracted from the URLs will be textual features, so you only need to select URLs that contain ASCII characters, and I do it this way:

```
data = data.filter(col("url").rlike(r'^[ -~]+$'))
data.show()
```

After this filtering I visualize for each of two classes (1 and 0) in *url_types* which are:

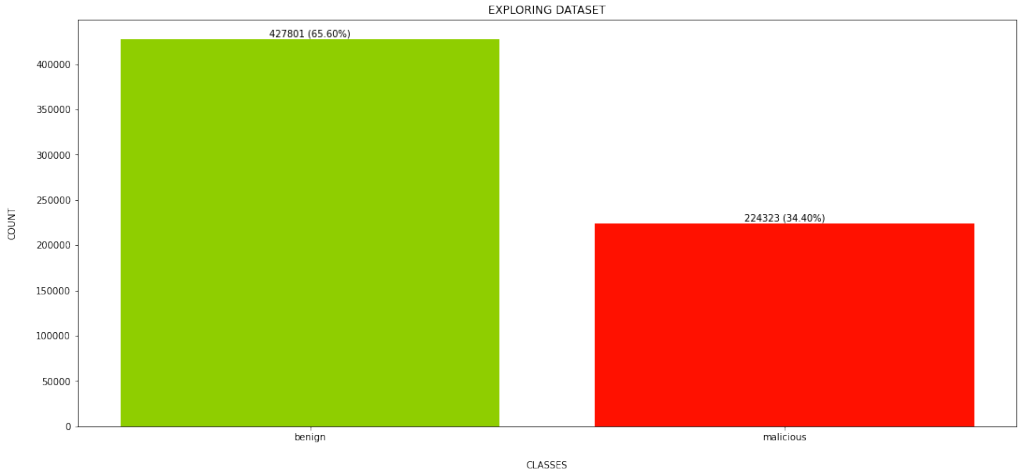


Figure 4.4: Types of URLs in url_types

It is evident that the two classes are unbalanced and I therefore sample one hundred thousand elements for each of the two classes in order to obtain a balanced dataset.

```
sample_size = 100000
data = data.sampleBy("url_type", fractions={0 :
    sample_size/data.filter(col("url_type") == 0).count(), 1
    :sample_size/data.filter(col("url_type") == 1).count()}, seed=42)
data.groupBy("url_type").count().show()
```

And I show the balanced dataset

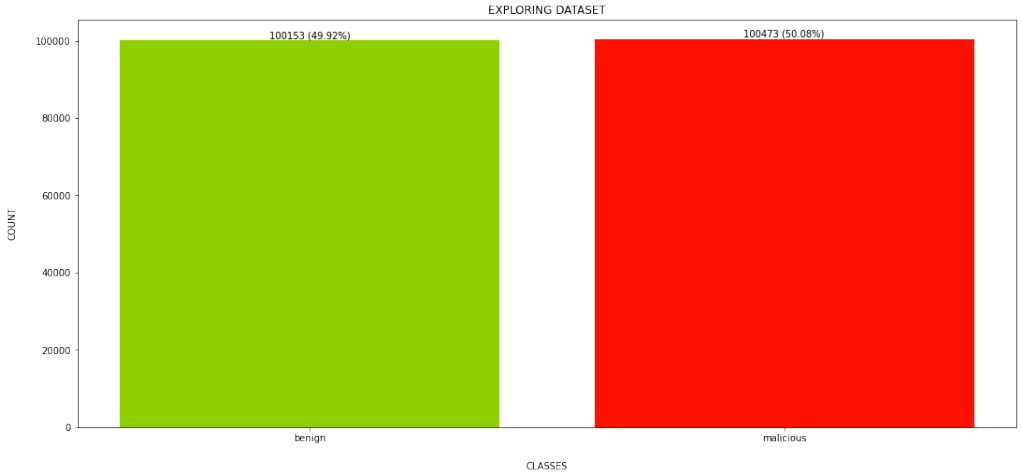


Figure 4.5: Balanced dataset

4.2 Features engineering

From the URLs that have undergone the filtering process, I proceed to extract the following textual features from the URL strings:

- "url_length" : lenght of URL
- "digit_count": number of digit characters in the URL
- "ampersand_count": number of ampersand characters in the URL
- "dot_count": number of dot characters in the URL
- "percent_count": number of '%' characters in the URL
- "at_count": number of '@' characters in the URL
- "tilde_count": number of '~' characters in the URL
- "hash_count": number of '' characters in the URL
- "has_https": checks if the URL starts with 'https'
- "has_http": checks if the URL starts with 'http'
- "starts_with_digit": checks if the URL starts with a digit

in the follwing way:

```
data = data.withColumn("url_length", length("url"))
data = data.withColumn("digit_count", length(regexp_replace(col("url"),
    "[^0-9]", "")))
data = data.withColumn("ampersand_count",
    length(regexp_replace(col("url"), "[^&]", "")))
data = data.withColumn("underscore_count",
    length(regexp_replace(col("url"), "[^_]", "")))
data = data.withColumn("dot_count", length(regexp_replace(col("url"),
    "[^.]", "")))
data = data.withColumn("percent_count", length(regexp_replace(col("url"),
    "[^%]", "")))
data = data.withColumn("at_count", length(regexp_replace(col("url"),
    "[^@]", "")))
data = data.withColumn("tilde_count", length(regexp_replace(col("url"),
    "[^~]", "")))
```

```
data = data.withColumn("hash_count", length(regexp_replace(col("url"),
    "[^#]", "")))
data = data.withColumn("has_https", when(data["url"].startswith("https"),
    1).otherwise(0))
data = data.withColumn("has_http", when(data["url"].startswith("http"),
    1).otherwise(0))
data = data.withColumn("starts_with_digit",
    when(data["url"].rlike("^[0-9]"), 1).otherwise(0))
```

Having done this, I display the distribution of the values of the various features:

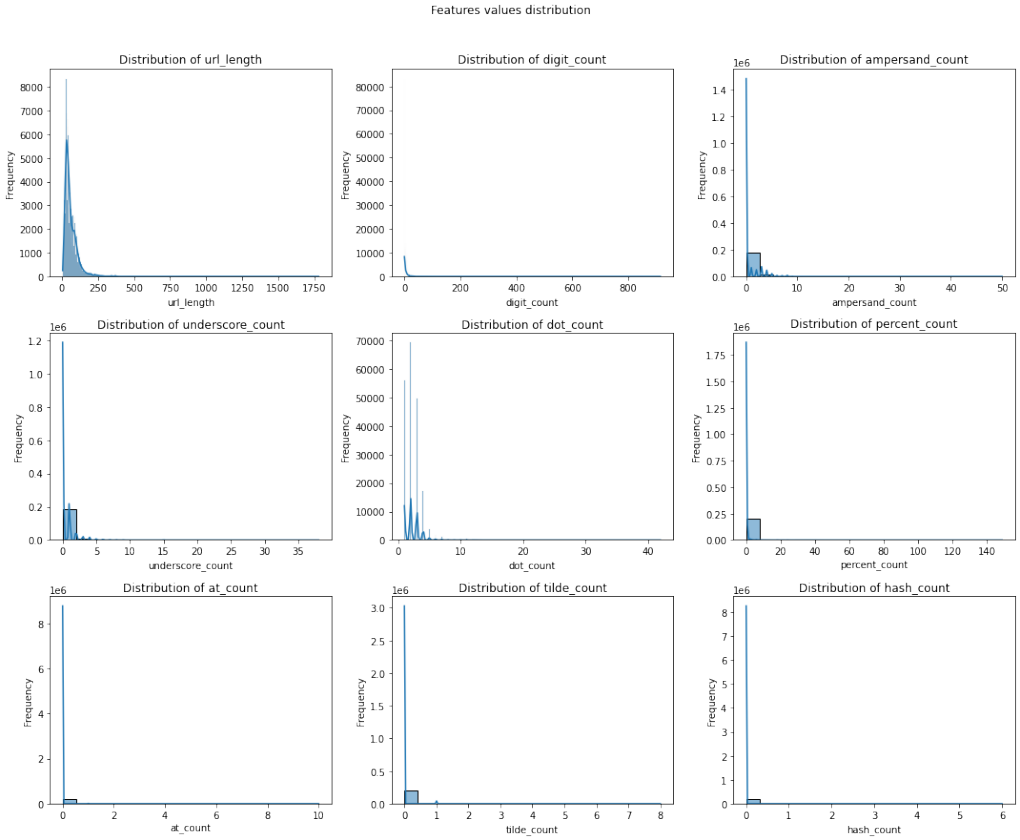


Figure 4.6: Features values distribution

4.3 Feature selection

In the feature selection phase I select the features with which models, which I will define in the next chapter, will be trained. For the purposes of the experiment, i.e. to compare the performance of basic classification models using different feature selection techniques, I decided to use the following techniques for training the models:

- Base features (all text features extracted from URLs)
- Principal component analysis
- Variance Threshold Selector
- Variable assocation: selection of variables highly collelated

As a first step, before applying the listed techniques, I delete the type column and put the features I am going to assemble into a vector in a list:

```
data = data.drop("type")
data.columns

feature_columns = [
    "url_length",
    "digit_count",
    "ampersand_count",
    "underscore_count",
    "dot_count",
    "percent_count",
    "at_count",
    "tilde_count",
    "hash_count",
    "has_https",
    "has_http",
    "starts_with_digit"
]
```

I use *VectorAssembler* to put together in a vector all the listed features for each row and it is useful to deal with ML tasks. Feature vectors are saved in a column called *features_ vector*.

```
assembler = VectorAssembler(inputCols=feature_columns,
    outputCol="features_vector")
data = assembler.transform(data)
```

4.3.1 Variable associations

I visualise in a confusion matrix the associations between the various features extracted from the URLs and then I plot it

```
correlation_matrix = Correlation.corr(data,
    "features_vector").collect()[0][0]

correlation_df = pd.DataFrame(correlation_matrix.toArray(),
    columns=feature_columns, index=feature_columns)

correlation_df.index, correlation_df.columns = feature_columns,
    feature_columns

sns.heatmap(correlation_df, annot=True, cmap="coolwarm", fmt=".2f",
    linewidths=.5)
plt.title("Correlation Matrix")
plt.show()
```

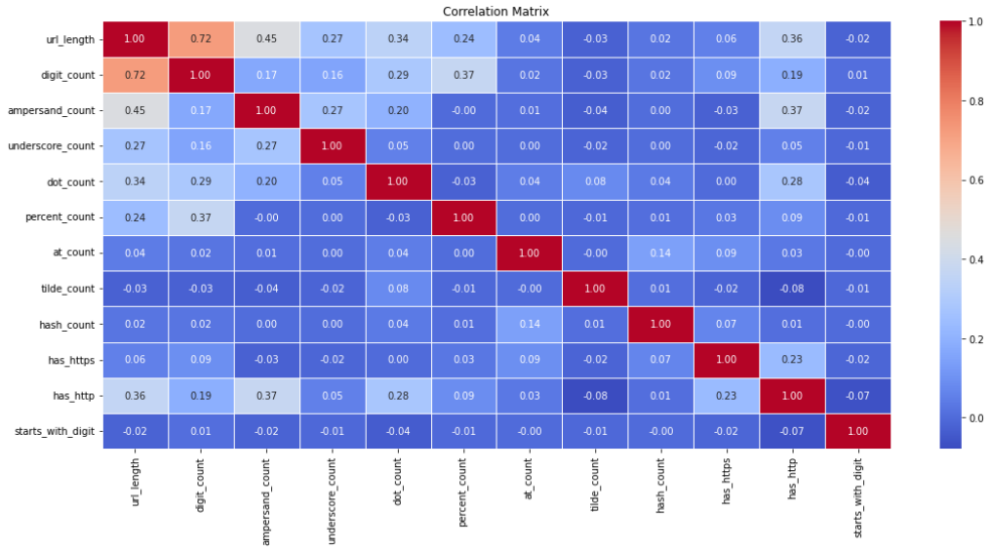


Figure 4.7: Correlation matrix

and I proceed to compute all the pairs of correlated features:

```
correlation_pairs = set()

for col1 in correlation_df.columns:
    for col2 in correlation_df.columns:
        if col1 != col2:
            correlation_value = correlation_df.loc[col1, col2]
            pair = (col1, col2) if col1 < col2 else (col2, col1)
            correlation_pairs.add((pair, abs(correlation_value)))

correlation_pairs_list = list(correlation_pairs)
correlation_pairs_list.sort(key=lambda x: x[1], reverse=True)

correlation_pairs_list
```

Then I select the couples of features which correlation is **above 0.35%** and I put those couples in a list of features:

```
threshold_correlation = 0.35

threshold_features_set_list = [
    feature_pair[0] for feature_pair in correlation_pairs_list
    if feature_pair[1] > threshold_correlation
]

threshold_features = []

for set_features in threshold_features_set_list:
    for feature in set_features:
        if feature not in threshold_features:
            threshold_features.append(feature)

threshold_features
```

And then assembling in a vector the list of selected features, which are:

- digit_count
- url_length
- ampersand_count

- has_http
- percent_count

```
assembler = VectorAssembler(inputCols=threshold_features,
    outputCol="threshold_features_vector")
data = assembler.transform(data)
```

4.3.2 Variance Threshold Selector

The VarianceThresholdSelector is a feature selection technique in machine learning used to eliminate features with low variance. Variance measures the spread of values within a feature, the selector calculates the variance for each feature and removes those with values below a specified threshold. This technique is useful for discarding features with limited variability, thereby simplifying the model and reducing overfitting.

```
selector = VarianceThresholdSelector(featuresCol="features_vector",
    varianceThreshold=1.0, outputCol="vts_features")
data = selector.fit(data).transform(data)
```

4.3.3 Principal Component Analysis - PCA

Principal Component Analysis (PCA) is a dimensionality reduction technique used in data analysis and machine learning. Its goal is to project the original data into a lower-dimensional space while preserving as much variance as possible. PCA identifies the principal axes along which the data varies the most and uses them as new variables, known as principal components.

Chapter 5

Experimentation

5.1 Choosing best features

For each model we want to choose best features among those from:

- Original features set
- Principal Component Analysis
- Autoencoder (encoded features)

The PCA for all models is made up of 2 to 8 components for all models considered and we compute metrics such as:

- Execution time
- Precision
- Recall
- F1-score
- Accuracy

We have added the execution time dependency because we are working on a fairly limited sample of the original dataset, therefore, evaluating the times together with the quality of the evaluation is an essential factor.

5.1.1 Support Vector Machine

By applying the SVM model to the various features sets, we obtain the following results:

Features	Precision	Recall	F1	Accuracy	Exec. Time
PCA_2	0.721	0.594	0.651	0.614	4.27
PCA_4	0.472	0.799	0.593	0.676	4.10
PCA_6	0.795	0.775	0.785	0.782	3.66
PCA_8	0.806	0.776	0.791	0.787	3.43
AutoEncoder_8	0.810	0.769	0.789	0.783	3.41
ALL_FEATURES_15	0.802	0.786	0.794	0.792	4.08

Table 5.1: Evaluation Metrics by Feature Sets | SVM

5.1.2 AdaBoost

By applying the AdaBoost model to the various features sets, we obtain the following results:

Features	Precision	Recall	F1	Accuracy	Exec. Time
PCA_2	0.708	0.759	0.733	0.742	0.23
PCA_4	0.731	0.791	0.759	0.768	0.34
PCA_6	0.779	0.783	0.781	0.782	0.48
PCA_8	0.746	0.808	0.776	0.784	0.60
AutoEncoder_8	0.761	0.793	0.777	0.781	0.56
ALL_FEATURES_15	0.810	0.847	0.828	0.832	0.28

Table 5.2: Evaluation Metrics by Feature Sets | AdaBoost

5.1.3 Random Forest

By applying the Random Forest model to the various features sets, we obtain the following results:

Features	Precision	Recall	F1	Accuracy	Exec. Time
PCA_2	0.801	0.821	0.811	0.813	1.06
PCA_4	0.849	0.860	0.854	0.855	1.75
PCA_6	0.853	0.867	0.860	0.861	1.80
PCA_8	0.856	0.867	0.861	0.862	1.79
AutoEncoder_8	0.852	0.870	0.861	0.862	1.69
ALL_FEATURES_15	0.884	0.900	0.892	0.893	0.79

Table 5.3: Evaluation Metrics by Feature Sets | Random Forest

5.1.4 Neural network

By applying the Neural Network model to the various features sets, we obtain the following results:

Features	Precision	Recall	F1	Accuracy	Exec. Time
PCA_2	0.721	0.594	0.651	0.614	169.63
PCA_4	0.709	0.720	0.714	0.717	178.49
PCA_6	0.795	0.772	0.783	0.780	189.09
PCA_8	0.817	0.763	0.789	0.782	176.72
AutoEncoder_8	0.845	0.806	0.825	0.821	177.50
ALL_FEATURES_15	0.825	0.787	0.806	0.801	176.18

Table 5.4: Evaluation Metrics by Feature Sets | Neural Network

5.1.5 Results comparison

We compare metrics calculated for all models (SVM, AdaBoost, Random Forest, Neural Network) trained on various features considered:

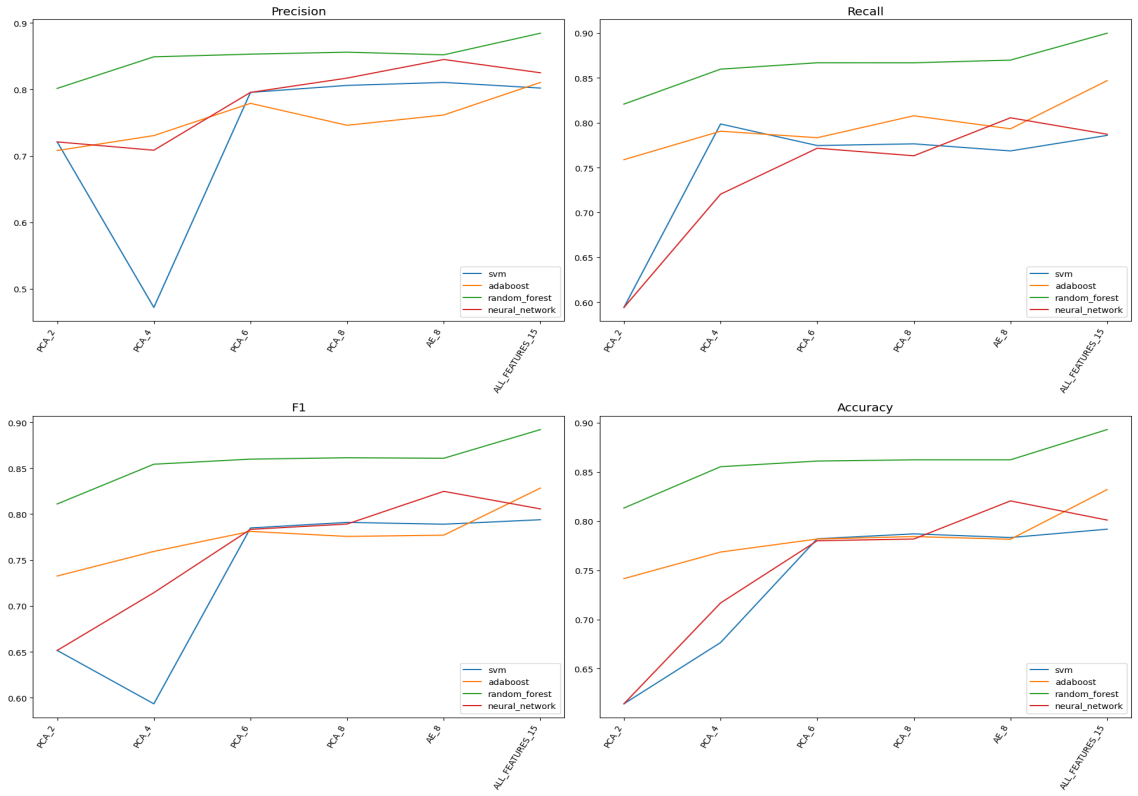


Figure 5.1: Metrics comparison

Finally we plot time varying features considered:

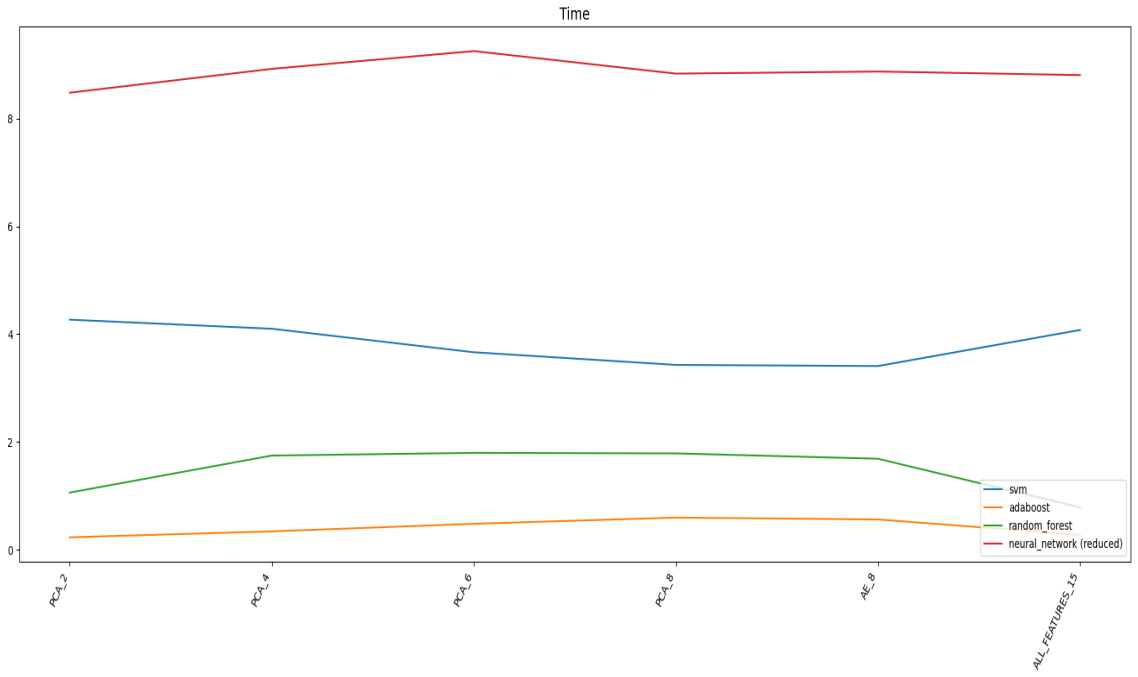


Figure 5.2: Time comparison

We have scaled the times of the Neural Network to make the graph more meaningful, anyways the consistency over the different features sets is maintained.

5.2 Hyperparameters tuning

After having seen the best features for the models, we want to see which parameters are best for each model in order to obtain more accuracy from the models.

5.2.1 Support Vector Machine

In SVM hyperparameters tuning we precede in two steps:

1. Choosing the best kernel
2. Choosing best parameters

SVM best kernel
Kernels considered for the SVM classifier are:

- Linear
- Poly
- RBF
- SIGMOID

we get following results:

Kernel	Precision	Recall	F1	Accuracy	Exec. Time
LINEAR	0.706	0.824	0.760	0.777	2.79
POLY	0.911	0.645	0.756	0.705	3.21
RBF	0.802	0.786	0.794	0.792	4.14
SIGMOID	0.621	0.628	0.624	0.626	4.32

Table 5.5: Evaluation Metrics for Different Kernels

SVM best hyperparameters
We do tuning of two hyperparameters: C representing the regularization parameter and *gamma* representing the kernel coefficient. Tuning is done using Grid search: an exhaustive search technique used to find the optimal combination of parameters

for a predictive model. It performs a systematic search over a predefined grid of values for each parameter, evaluating the model for each combination and selecting the one with the best performance.
We consider those values for hyper-parameters:

Hyperparameter	Range Values
C	[0.1, 1, 10, 100, 1000, 5000]
gamma	[10, 1, 0.1, 0.01, 0.001, 0.0001]

Table 5.6: Hyperparameter Range Values | SVM

Grid search returned, as best parameters, those values:

- C: 5000
- gamma: 10

We train a new SVM classifier using *RBF* kernel and we compute metrics:

Metric	Base Model	Best Model	Gain (%)
Precision	0.802	0.848	5.74
Recall	0.786	0.874	11.24
F1	0.794	0.861	8.45
Accuracy	0.792	0.863	9.00
Execution Time	4.229	46.584	1001.53

Table 5.7: Model Performance Comparison | SVM

5.2.2 AdaBoost

For the AdaBoost model we tune two hyperparameters: the learning rate and the number of estimators choosing among following values using Grid Search:

Hyperparameter	Range Values
n_estimators	[10, 50, 100, 500, 1000]
learning_rate	[0.0001, 0.001, 0.01, 0.1, 1.0]

Table 5.8: Hyperparameter Range Values | AdaBoost

We get as best parameters:

- learning_rate: 1.0

- n_estimators: 500

We train a new AdaBoost classifier setting those hyperparameters and getting following results:

Metric	Base Model	Best Model	Gain (%)
Precision	0.810	0.810	0.00
Recall	0.847	0.850	0.42
F1	0.828	0.830	0.20
Accuracy	0.832	0.834	0.24
Execution Time	0.287	2.872	902.19

Table 5.9: Model Performance Comparison | AdaBoost

5.2.3 Random Forest

In Random Forest hyperparameters tuning we precede in two steps:

1. Choosing the best criterion
2. Choosing best parameters

Random Forest best criterion
Criterion considered for the Random Forest classifier are:

- Gini
- Entropy
- Log_Loss

we get following results:

Criterion	Precision	Recall	F1	Accuracy	Exec. Time
GINI	0.884	0.900	0.892	0.893	0.82
ENTROPY	0.888	0.900	0.894	0.894	0.89
LOG_LOSS	0.888	0.900	0.894	0.894	0.88

Table 5.10: Evaluation Metrics for Different Criterions

The values obtained by varying the criterion are quite the same, so we decide to apply the default one, which is '*gini*', for the future computations.

Random Forest best hyperparameters

We do tuning of five hyperparameters:

1. Number of estimators = number of trees in the forest
2. Max depth = max number of levels in each decision tree
3. Min samples split = min number of data points placed in a node before the node is split
4. Min samples leaf = min number of data points allowed in a leaf node
5. Bootstrap = method for sampling data points (with or without replacement)

We consider those values sets for hyperparameters tuning:

Hyperparameter	**Range Values**
n_estimators	[100, 500, 1000, 3000]
max_depth	[10, 100, 200, None]
min_samples_split	[2, 10, 20]
min_samples_leaf	[1, 5]
bootstrap	[True, False]

Table 5.11: Hyperparameter Range Values | Random Forest

Best parameter for Random forest are:

- n_estimators: 3000
- max_depth: 100
- min_samples_split: 10
- min_samples_leaf: 1
- bootstrap: False

We train a new Random forest classifier with the best hyperparameters and getting following results:

Metric	Base Model	Best Model	Gain (%)
Precision	0.884	0.888	0.45
Recall	0.900	0.904	0.45
F1	0.892	0.896	0.45
Accuracy	0.893	0.897	0.45
Execution Time	0.789	31.783	3929.51

Table 5.12: Model Performance Comparison | Random Forest

5.2.4 Neural network

This time we try different configurations of parameters and then choosing the set that gives the best accuracy, hyperparameters and values to choose are the following:

Hyperparameter	Range Values
batch	[100, None]
lr	[0.1, 0.01]
momentum	[0.0, 0.9]
weight_decay	[0.0, 0.01]

Table 5.13: Hyperparameter Range Values | Neural Network

The configuration with those parameters gives an accuracy of 0.80:

- batch: 100
- lr: 0.1
- momentum: 0.9
- weight_decay: 0.0

We train again the new neural network using those parameters and we get:

Metric	Base Model	Best Model	Gain (%)
Precision	0.721	0.842	16.71
Recall	0.594	0.793	33.55
F1	0.651	0.817	25.38
Accuracy	0.614	0.811	32.07
Execution Time	3.580	248.808	6850.78

Table 5.14: Model Performance Comparison | Neural Network

5.3 Alternative approach - URL2VEC

We propose, also, an alternative approach to classifying malicious URLs. This deep-learning approach is inspired by word2vec-based word embedding used in Natural Language Processing and transfers the same principles to a character embedding that can be achieved from a corpus of URLs. So this alternative approach, that we can call **URL2Vec**, instead of extracting the lexical features as done so far, it constructs a structure-wise URL representations in vector forms and uses a neural network to classify the URLs starting from this vector representation.

The following figure shows the general workflow of our model.

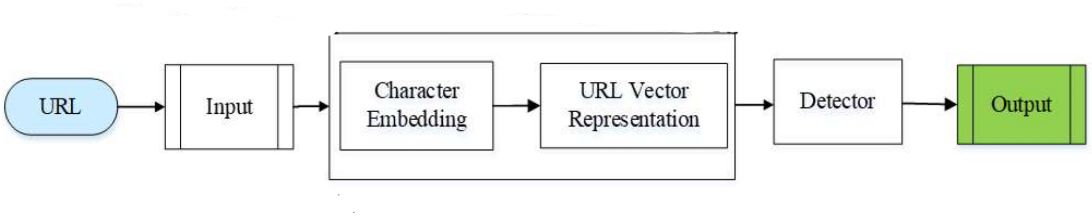


Figure 5.3: Workflow of our model

URL2Vec is a method that aims to capture the sequential dependencies and contextual information of URLs by encoding the characters that form them into numerical representations. This process first involves the building of a dictionary of chars (including specials) from a corpus of URLs taken from our original dataset, then transforms URLs into embedding representation and we choose a fixed size for the vector representation. Finally the resulting vector representations enable a machine learning or deep learning model to classify URLs based on their numerical features.

In our implementation, after a careful examination of our data, we fix to 67 the length of the features vector, we truncate the longer URLs and padding with zero (reserved value) the shorter ones. This is done because 67 is the mean value of URLs lengths and cover two thirds of the total examples without truncation and loosing of information.

Then we trained the same fully connected neural network, already described in the previous section, and here we plot results:

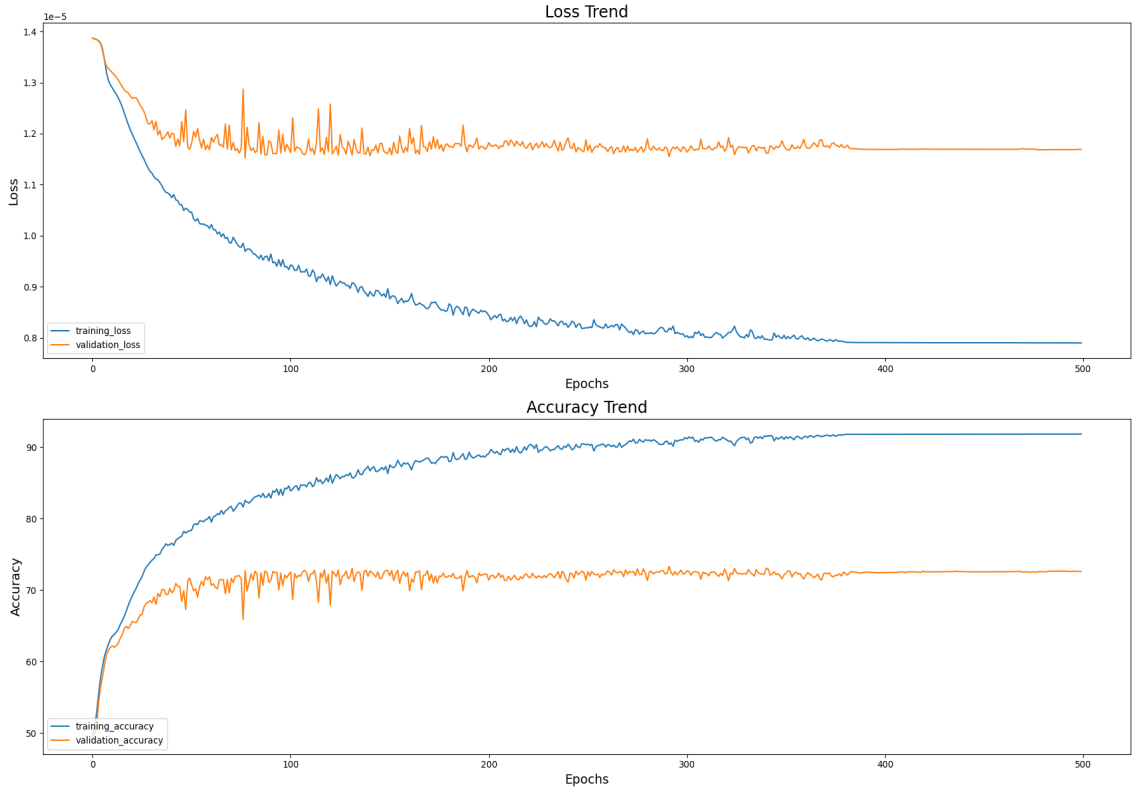


Figure 5.4: Accuracy and loss neural network with URLs vectors

We got an accuracy of 72.52% and an execution time of 173.71 seconds.

Class	Precision	Recall	F1-Score	Support
safe_URL	0.72	0.74	0.73	2024
unsafe_URL	0.73	0.71	0.72	1976

Table 5.15: Classification Metrics | AdaBoost

Accuracy	Macro avg	Weighted avg	Support
0.73	0.73	0.73	4000

Table 5.16: Classification Metrics/2

Chapter 6

Conclusions

Our project overall tests five different machine learning or deep learning approaches to classify malicious URLs. From experiments executed it turns out that the best model to deal with this task is the **Random Forest** classifier on lexical features, which outperforms all the other models implemented with **89,7%** of accuracy. Although the accuracy of this model is very good, the execution time is slightly longer, for which would be preferable the **AdaBoost** classifier, that although not achieves the same performance, has an faster execution time.

6.1 Experiments results

With respect to the single experiments developed we can conclude the following considerations.

- **1) Chose best features**: regarding the choice of the best set of features, we can conclude that the set of initial extracted 15 lexical features leads to better results and to execution times that do not differ too much from the application of the models on a reduced set. However the 8 features set extracted from the Autoencoder have shown very promising results, indeed leading to higher performances respect to the whole set in the case of Neural Network and SVM.

- **2) Hyperparameters tuning**: the careful selection of the best parameters has brought advantages for all models even if in some cases only by a few tenths of a percentage. Also, with execution times in mind, training and testing of optimized models has increased exponentially. Therefore, we can conclude that only the SVM model (+9% in accuracy) and Neural Network

(+32% in accuracy) have achieved a performance advantage such as to justify longer execution times.

- **3) Alternative approach (URL2VEC)**: this approach, of great interest and still little explored by the scientific community, has not led to better results than traditional approaches. In part it is due to the classification model applied, the fully connected neural network, which is a type of network that makes no assumptions on the type of input, making it very versatile but with poorer performance compared to models that make assumption on sequentiality of input data, such as the recurrent ones or as the transformers models.

6.2 Future developments

Possible future developments for this project could be:

- Implementation of a powerful neural network for URL2Vec approach, such as any model that keep track of dependencies on sequential data (LSTM, GRU, Transformers, etc.).
- Applying the models implemented not only on lexical features but also on host-based features.
- Building a meta model that classifies only the malicious URLs previously detected according to the type of possible attacks (defacement, phishing, malware, spam, etc.).
- Once chosen the best model and the best settings applying them to the whole dataset and not on a small sample as done in this project due to limited hardware resource.
- Implementation of a web interface that works as utility for users who want to verify the quality of an URL received.

Bibliography

[1] Il Sole 24 ore. Quanti dati sono generati ogni minuto? https://www.infodata.ilsole24ore.com/2022/01/02/quanti-dati-generati-minuto/?refresh$_ce = 1. Accessed : 18 - 06 - 2023.$