Halil Alperen Gözeten - Computer Science - 21902464
Kaan Tek - Computer Science - 21901946
TA: Artun Saday, Instructor: Cem Tekin

EEE-485 Final Project Report
Malicious URL Detection

1. Introduction

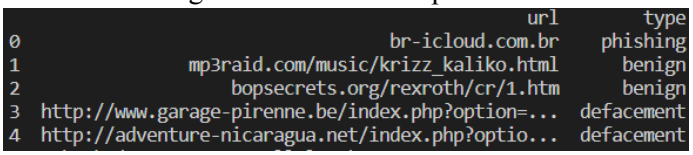
As technology keeps progressing quickly and the use of the internet becomes more and more common in daily life, malicious URLs have become a real threat. These are links that are created artificially in order to promote scams, frauds, and phishing and are even used for cyberattacks. It is important for one to understand whether a URL is malicious or not before clicking the link; otherwise, one might be a victim of people with bad intentions. Hence, we aim to design and implement a model that can effectively predict whether a URL is safe or not with the help of various machine-learning algorithms.

2. Task

The main tasks of this project include data processing, model training, performance evaluation, and comparison. In this process, we first examine the data in our hands and add some new features to it, which will be explained in more detail in the Preprocessing section. Then, we use Logistic Regression, Multinomial Naive Bayes, and Neural Networks as machine-learning and deep-learning algorithms in order to test and train our model. Then, various complexities of these models are tested while measuring the performance and tuning the necessary hyperparameters. In the end, we will compare the performances and the efficiencies of these models according to some metrics, such as accuracy, and conclude which one fits our purpose the best.

3. Dataset

We used a dataset named "*Malicious URLs dataset*" that we found on *kaggle,* which has been formed by collecting URL data from different sources, including ISCX-URL2016 [1]. This dataset consists of 651191 URLs in total, a data frame that contains the names of the URLs together with their labels, out of which 428103 are considered to be benign, while the remaining are malicious. This dataset is great in terms of providing us with a huge amount of data along with their labels. However, it seems to lack information about some of the features that could be useful in training our model, and therefore, we performed some feature engineering on the existing data in order to extract some features, such as the length of the URL, length of the domain name; and the number of some special characters that appear in the URL, such as "-, @, ?, =", or the number of letters and digits, and the counts of the special words like *http, https, www* and so on (explained in more detail in the Preprocessing section). For this dataset, we used 70-10-20 as our training, validation, and test splits, and we trained our final models on training + validation and reported metrics on the test dataset.



	url	type
0	br-icloud.com.br	phishing
1	mp3raid.com/music/krizz_kaliko.html	benign
2	bopsecrets.org/rexroth/cr/1.htm	benign
3	http://www.garage-pirenne.be/index.php?option=...	defacement
4	http://adventure-nicaragua.net/index.php?optio...	defacement

Fig.1: *The First 5 Rows of The Dataset [1]*

4. Programming Environment

During the implementations, the main programming language used is Python, and the version used is 3.9.7. Visual Studio Code is used as the IDE as it helps the organization and maintenance of the project together with the version control tools such as Git and GitHub. Throughout the project, we carried out the developments on a collaborative repository on GitHub.

We plan to benefit from NumPy library for efficient linear algebra operations, Pandas for data preparation and wrangling, Matplotlib for data visualization, Python's tld library to extract useful information from URLs, and tqdm to create progress bars for loops.

5. Data Preprocessing

The initial dataset before preprocessing only consisted of long strings of URLs. Therefore, we had to do feature engineering and data preprocessing in order to come up with features that could be useful for our models [2][3].

First, we used a method to determine whether the URL is an IP address or not, and if it is, which type of URL it is. As a result, we came up with a feature named *'use_of_ip'* that takes binary values [3][4]. Then, we benefited from the get_tld() function of the tld library and used it to extract

full-length domain (fld), top-level domain (tld), domain, and subdomain names if they exist [5][6]. We then found their lengths, which resulted in 4 new features: *'url_length', 'subdomain_length', 'tld_length', 'fld_length',* and *'path_length'*. We thought length of these could be a useful feature as we observed that malicious URLs tend to be longer than the benign ones, and this is especially the case for the domain names. We also added another binary feature called, and *'use_of_shortener'*, which indicate the path of the URL (for example, the remaining part of the URL after '.com' and whether the URL contains a shortening service (for example, Bitly or TinyURL) or not, respectively. We found the list of shortening services from a GitHub repository and searched for the existence of them [7]. For the final version of our model, we also added a method to see if the URL contains some suspicious words, such as: cash, free, lucky, bonus… and added a new feature *'is_susp'* accordingly. Similarly, when we investigated our dataset, we noticed that many of the malicious URLs contained the word 'php', hence we added a new feature *'is_php'*, to see if the URL contains that word or not.

We then added more features to represent the number of occurrences of alphanumeric characters, digits, and punctuations, named *'count_letters', 'count_digits'*, and *'count_puncs'*. Then we counted the number of some special characters in the URL format in order to determine whether a URL could be malicious or not, and therefore, we added the following features as well: *'count.', 'count@', 'count%', 'count?',* and *'count='* [2]. We used simple lambda functions and Python's count() function in order to obtain these numbers. Again for our final model, we added features of counts of some other characters as well: *'count+', 'count#', 'count/', 'count$', 'count!', 'count*', 'count,', 'count_',* and *'count:'*. However, as these features are not directly correlated with the URL structure, incorporating all of them simultaneously resulted in a degradation of the models' performance and thus, we employed techniques for feature selection to mitigate this issue. In addition to these features, we also attempted to incorporate the count of the '%20' character (used to represent spaces in URLs) as an additional feature, however, through experimentation, we discovered that this particular feature adversely affects the performance of all machine learning models [8].

Additionally, we added *'count_dirs'*, and *'first_dir_length'*, which represent the number of the subdirectories in the URL (this is in fact equal to the number of '/' characters) and the length of the first string, respectively. We did this extraction based on our previous observation that malicious URLs tend to be longer in length. We also added another two features named *'url_length_q'* and *'fld_length_q',* to distribute url_length and fld_length into 4 groups named 'Short', 'Medium', 'Long', and 'Very Long' by using percentiles based on the distribution of the data as. We obtained categorical features as a result of this operation, therefore, we later converted these values to integers using the fit_transform() method of OrdinalEncoder.

We also checked whether the URL starts with 'http' or 'https', and their number of occurrences as well as the occurrence of 'www'. Hence, we added: *'https', 'count-https', 'count-http',* and *'count-www'* and all these features take binary values since a URL doesn't contain more than one of these.

Then we added *'letters_ratio', 'digit_ratio'*, and *'punc_ratio'*, which we simply obtained by dividing count_letters, count_digits, and count_puncs by url_length since we thought they could be helpful for the Logistic Regression model, but we can't use these features for the Naive Bayes models since they take non-integer values.

Finally, we added the label feature called *'is_malicious'*. In our initial dataset, the label feature consisted of 4 different values: Benign, Defacement, Phishing, and Malware. We decided to label URLs that are Benign as 0 and all the others that are malicious as 1. Therefore, we converted our problem into a binary classification in which we simply aim to determine whether a URL is benign or not. With respect to these changes, we added the new feature *'is_malicious'*. After doing this, the class distributions became around 65-35 and this skewness of the dataset didn't create any problem for us when training our models and thus, we didn't apply any oversampling or undersampling technique.

We saved this new data frame with the name 'url_processed.csv', and in both of our models, we used %70 of our data as the training set, %10 as the validation set, and %20 as the test set.

6. Detailed Description of the Proposed Methods

In this project, we will examine the effectiveness of different machine learning algorithms on our URL classification problem, which will include Logistic Regression, Multinomial Naive Bayes, and Neural Networks. All these three types of models are known to be efficient in classification

problems and hence, are suitable for our binary classification problem. Now, we will explain each of these methods in more detail.

6.1. Logistic Regression

Logistic Regression, first of all, is very efficient for classification problems and also efficient to train, and it's quite advantageous when the data is linearly separable. Besides that, the coefficients obtained in the Logistic Regression can be interpreted to get insights about the features, which can be helpful for the Final Report when picking the best subset of features for obtaining the most generalizable model. A Logistic Regression model tries to fit the logistic model to the data, which models the probability of an event and estimates the necessary parameters for that. For the Logistic Regression Model, we have the following maximum conditional likelihood estimators:

$$P(Y = 0|\, X,\, w) \;=\; \frac{1}{1 + e^{(w_0 + \sum_i w_i X_i)}} = \frac{1}{1 + e^{w^T x}} \text{ and}$$

$$P(Y = 1|\, X,\, w) \;=\; \frac{e^{(w_0 + \sum_i w_i X_i)}}{1 + e^{(w_0 + \sum_i w_i X_i)}} = \frac{e^{w^T x}}{1 + e^{w^T x}}$$

where $X = [1,\, X_1,\, X_2,\, ...,\, X_p]^T$ is the feature vector and $w = [w_0,\, w_1,\, ...,\, w_p]^T$ is the parameter (weight) vector. Using these, we can form the likelihood function as

$$L(w) = P(D|w) = \prod_{i:\, y_i = 1} \frac{e^{w^T x_i}}{1 + e^{w^T x_i}} \prod_{i:\, y_i = 0} \frac{1}{1 + e^{w^T x_i}}$$

where $D = \{(x_i, y_i)\}$ for $i = 1, 2, .., n$ and n denotes the total number of samples. For making the computations easier, let's find the log-likelihood function:

$$l(w) = ln(P(D|w)) = \sum_{i=1}^{n} [y_i w^T x_i - ln(1 + e^{w^T x_i})]$$

The derivative of this function is:

$$\bigtriangledown_w l(w) = ln(P(D|w)) = \sum_{i=1}^{n} x_i [y_i - \frac{e^{x_j^T w^{(t)}}}{1 + e^{x_j^T w^{(t)}}}]$$

At the MLE estimate of w, the derivative of this function is 0. But this equation has no closed-form solution; therefore, we will use Gradient Update to converge to a local maximum. Hence, we can form the gradient update for the weight vector w can be derived as the following:

$$w_i^{(t+1)} \leftarrow w_i^{(t)} + \alpha \sum_j x_i^j [y^j - \widehat{P}(Y^j = 1|x_j, w^{(t)})] = w_i^{(t)} + \alpha \sum_j x_i^j [y^j - \frac{e^{x_j^T w^{(t)}}}{1 + e^{x_j^T w^{(t)}}}]$$

for t in range $[0,\, \#\, epochs]$

In the above notation, j denotes an index to a data sample and $x_0^j = 1$, y_j is the actual label for the data sample j, $w_0 = b$ is the bias term and $w_i^{(t)}$ denotes the ith element of the weight vector at iteration t and finally α denotes the learning rate for the update.

6.2. Naive Bayes Models with Bag-Of-Words Representation

Naive Bayes classifiers are based on the application of the Bayes Theorem, and they have 'naive' (conditional independence) assumptions for the features. Because of this, the Naive Bayes models handle the noise well. Also, the training using the maximum likelihood for these models is quite cheap, requiring linear time; the classification of a single data point is also quite fast by only picking the highest probability class. These models are useful in the classification for large datasets, as in the case of URL dataset since they're highly scalable with the number of data samples and the number of features. We will use the Multinomial and Bernoulli models together with the bag-of-words representation. The bag-of-words document representation has the assumption that a feature's (word)

existence is conditionally independent of its position given the class of the document, and we will use this assumption for both models.

6.2.1 Multinomial Naive Bayes Model

The Multinomial Naive Bayes Model takes multiple occurrences of a feature (word) into account and calculates probabilities with respect to occurrences of other features (words). Suppose we have a document D_i with n_i features in it, then the probability of D_i belonging to class y_k can be computed as

$$P(D_i|Y = y_k) = P(X_1 = x_1, X_2 = x_2, \ldots, X_{n_i} = x_{n_i}|Y = y_k) = \prod_{j=1}^{n_i} P(X_j = x_j|Y = y_k)$$

In the above equation X_j represents the feature at j^{th} position in the document D_i and x_j represents the actual feature that's in the j^{th} position in the document, and lastly n_i denotes the number of positions in the document. In the above formulation, the length of the feature vector for the document, the length of the $X^{(i)}$, depends on the number of features in the document, which is n_i. But looking from the perspective of the set of features, we can also formulate this as

$$P(D_i|Y = y_k) = \prod_{j=1}^{|V|} P(X_j|Y = y_k)^{t_{w_j,i}}$$

In this notation, $|V|$ is the size of the feature set V, and X_j represents the appearance of the j^{th} feature and $t_{w_j,i}$ indicates how many times feature w_j appears in a document D_i. In classifying the documents, we are interested in the probability distributions over a set of possible classes, and hence, we need to calculate $P(Y = y_k|D_i)$. Using the Bayes Rule, this is equal to:

$$P(Y = y_k|D_i) = \frac{P(Y=y_k)\prod_{j=1}^{|V|} P(X_j|Y=y_k)^{t_{w_j,i}}}{\sum_k P(Y=y_k)\prod_{j=1}^{|V|} P(X_j|Y=y_k)^{t_{w_j,i}}}$$

We realize that the denominator is fixed for each class y_k and thus, we can say

$$P(Y = y_k|D_i) \propto P(Y = y_k)\prod_{j=1}^{|V|} P(X_j|Y = y_k)^{t_{w_j,i}}$$

The class for which this probability is maximized will be the prediction:

$$\widehat{y_i} = argmax_k P(Y = y_k|D_i) = argmax_k P(Y = y_k)\prod_{j=1}^{|V|} P(X_j|Y = y_k)^{t_{w_j,i}}$$

Since each probability in the above multiplication is in the interval $[0, 1]$, we might face underflow issues in programming this method. Since the logarithm is a strictly increasing function in the interval $[0, 1]$, maximizing the above probability is the same as maximizing the logarithm of it. Therefore,

$$\widehat{y_i} = argmax_k P(Y = y_k|D_i) = argmax_k [ln(P(Y = y_k)) + \sum_{j=1}^{|V|} t_{w_j,i} * P(X_j|Y = y_k)]$$

where

$$P(X_j|Y = y_k) = \frac{T_{j,Y=y_k}}{\sum_{j=1}^{|V|} T_{j,Y=y_k}} \text{ and } P(Y = y_k) = \frac{N_k}{N}$$

and here, N indicates the total number of samples in the training set while N_k indicates the total number of samples of with true label y_k. Also, $T_{j,Y=y_k}$ is the number of occurrences of the j^{th} feature

in documents of class y_k in the training set, including the multiple occurrences of that feature. Here, we can also possibly apply smoothing to these estimated parameters, which is by adding imaginary occurrences (α times where α is the smoothing parameter) of every feature to every class, and parameters become:

$$P(X_j|Y = y_k) = \frac{T_{j,Y=y_k} + \alpha}{\alpha * |V| \sum_{j=1}^{|V|} T_{j,Y=y_k}}$$

This way, the model becomes more "fair" by hallucinating each feature in each class, not directly yielding to the elimination of a document because of the absence of a specific feature.

6.2.2 Bernoulli Naive Bayes Model

The Bernoulli Model is interested in the absence or presence of the features and represents this with 0 and 1, where the latter indicates the presence of the feature. Unlike Multinomial, a probability for a parameter belonging to a specific feature is independent of the other features. Again, we will predict as the class for which the class probability is maximized, which is:

$$\widehat{y_i} = argmax_k P(Y = y_k|D_i)$$

$$= argmax_k P(Y = y_k) \prod_{j=1}^{|V|} (t_j P(X_j|Y = y_k) + (1 - t_j)(1 - P(X_j|Y = y_k)))$$

Again taking the logarithm, we have

$$\widehat{y_i} = argmax_k [ln(P(Y = y_k)) + \sum_{j=1}^{|V|} ln(t_j P(X_j|Y = y_k) + (1 - t_j)(1 - P(X_j|Y = y_k)))]$$

To calculate this, the needed parameters are

$$P(X_j|Y = y_k) = \frac{S_{j,Y=y_k}}{N_k} \text{ and } P(Y = y_k) = \frac{N_k}{N}$$

where N indicates the total number of samples in the training set while N_k indicates the total number of samples of with true label y_k. Also, $S_{j,Y=y_k}$ is the number of occurrences of the j^{th} feature in documents of class y_k in the training set, not including the multiple occurrences of that feature.

6.3. Neural Networks

Finally, we have chosen to implement Neural Networks as our preferred method due to their ability to learn non-linear and complex patterns in the data. Compared to previous methods, neural networks offer greater flexibility in fitting the data. A neural network is composed of multiple layers of interconnected neurons, and various parameters, such as the number of neurons in a layer, the connectivity between layers, and the number of hidden layers, can be adjusted or tuned when designing the model. After each layer, non-linear activation functions are applied to the outputs, enhancing the model's capacity to create more generalizable representations. By stacking multiple layers of neurons, the model can effectively capture intricate functions, albeit at the cost of optimization efforts. To update the model with new data, we have opted to test the batch gradient-descent (BGD) approach with batch sizes 32 and 64. The model weights are updated using back-propagation, which involves tracing from the deepest layer to the input layer and applying the chain rule to derive the necessary update equations for the parameters of the model. However, the process can sometimes get stuck in flat regions or oscillate around the minimum, leading to slower convergence. Momentum addresses this issue by introducing a "velocity" term that accumulates the past gradients and influences the current weight update. It adds a fraction of the previous velocity to the current gradient, making the updates more consistent and stable over time and therefore, we also introduced the momentum parameter in the trainings. In our URL classification task, we primarily experimented with the Rectified Linear Unit (ReLU) activation function for the hidden layers. This choice was motivated by its ability to alleviate the vanishing gradients problem and expedite the training process (essential because of the size of the dataset) compared to other activation functions.

Additionally, we intend to use the Binary Cross-Entropy loss function, sigmoid output activation, and a threshold to utilize our model as a classifier. To address potential overfitting issues, we might employ (if necessary) L2 and L1 regularization methods, which can enhance the model's generalization capabilities and prevent overfitting, thus improving its performance.

7. Preliminary Results

We implemented our Logistic Regression and Multinomial Naive Bayes models for this stage and tested and evaluated their final performances.

7.1. Logistic Regression

The new features that will be mentioned in 7.2.2 are added to the design matrix before training and testing the Multinomial Naive Bayes Model. We trained a total of 12 logistic regression models and tested their performances using our validation dataset. The first 4 of these 12 models have their weights initialized with normal distribution, the next 4 have their weights initialized with uniform distribution, and the remaining 4 with simply zeros. For each of the three different types of these initializers, we applied batch gradient descent by training with four different batch sizes, which are 32, 64, 128, and 256. We did this training using our train and validation data sets and fed each of our models with the training set a total of 20 times, so the epoch value picked was equal to 20. An example accuracy that we observed on our validation sets for each initialization method and different batch sizes can be seen in the plot below:

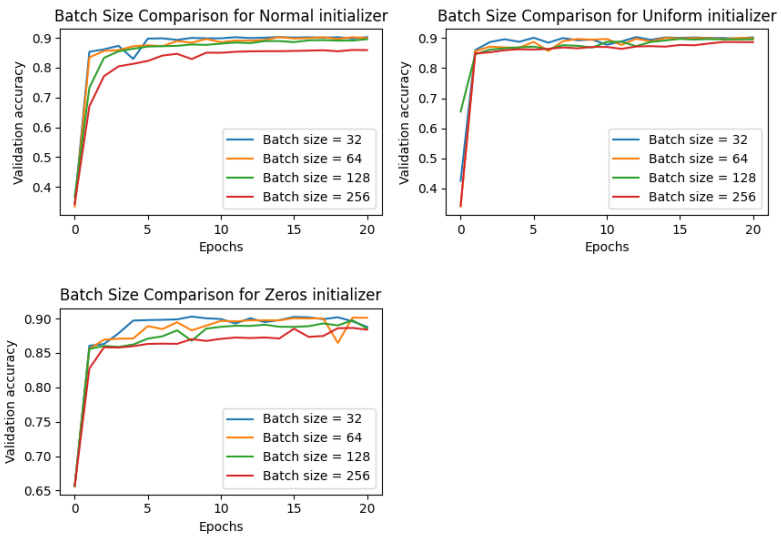


Fig.2: *Validation accuracies for different initializations and batch sizes*

The values for each accuracy points in the plot are:

```
Final Validation accuracy for Normal Initialization With Batch Size: 32 is : 0.899783
Final Validation accuracy for Normal Initialization With Batch Size: 64 is : 0.900475
Final Validation accuracy for Normal Initialization With Batch Size: 96 is : 0.885026
Final Validation accuracy for Normal Initialization With Batch Size: 128 is : 0.891721
Final Validation accuracy for Uniform Initialization With Batch Size: 32 is : 0.905373
Final Validation accuracy for Uniform Initialization With Batch Size: 64 is : 0.890800
Final Validation accuracy for Uniform Initialization With Batch Size: 96 is : 0.898309
Final Validation accuracy for Uniform Initialization With Batch Size: 128 is : 0.888758
Final Validation accuracy for Zeros Initialization With Batch Size: 32 is : 0.902670
Final Validation accuracy for Zeros Initialization With Batch Size: 64 is : 0.902241
Final Validation accuracy for Zeros Initialization With Batch Size: 96 is : 0.897419
Final Validation accuracy for Zeros Initialization With Batch Size: 128 is : 0.888143
```

On the machine equipped with Intel i7-10750h, the trainings took 17, 12 and 8 and 7 seconds on average, by using the tqdm library for 32, 64, 128, and 256 batch sizes, respectively.

The results of the above experiment were changing as each run used different seeds, but in several runs and as in the above example, the model in which Normal distribution was used with a batch size of 64 seemed to be one of the best performers in terms of accuracy, and it usually converged to a local minima faster than the other models. Therefore, we chose it as our final model,

trained it on our train and validation set, and then tested it on our test set. The final accuracy of our model can be seen in the plot below:

Fig.3: *Final Test Accuracy for the Final Logistic Regression Model*

For this test, the obtained confusion matrix was as follows:

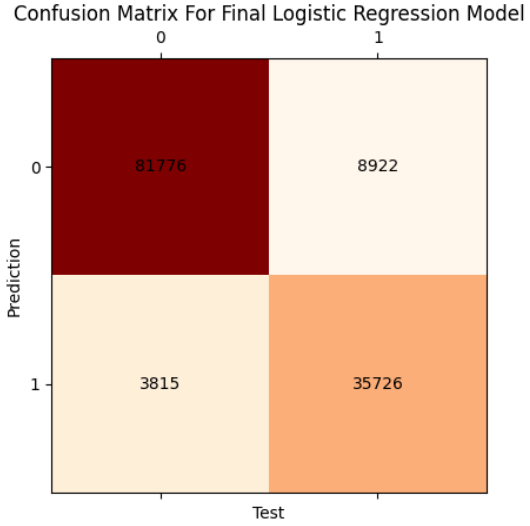


Fig.4: *Confusion Matrix for the Final Logistic Regression Model*

```
Final Test Accuracy: 0.90220
Final Test Precision: 0.9035
Final Test Recall: 0.8002
F1 Score For Final Logistic Regression Model: 0.84871
```

7.2.1. Multinomial Naive Bayes Model

The new features that will be mentioned in 7.2.2 are added to the design matrix before training and testing the Multinomial Naive Bayes Model. As mentioned earlier, we trained our Multinomial Naive Bayes model with 5 different smoothing parameters: 0, 2, 4, 6, and 8, to see the effects of this parameter on the model quality. The accuracy that we obtained on our validation set can be seen in the plot below:

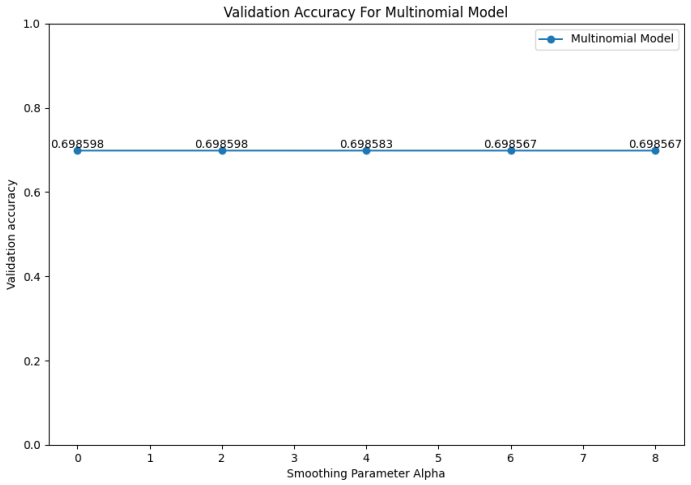


Fig.5: *Validation Accuracy for the Multinomial Models With Different Smoothing Parameters*

Again on the machine equipped with Intel i7-10750h, the trainings took around 0.04 seconds (Python's *time* library gave this result) for a single Multinomial Model since the calculations were quite fast as they're completed in linear runtime complexity and by avoiding for-loops with the help of numpy library.

Interestingly, as can be inferred from the results we obtained, increasing the smoothing parameter did not have much of an effect on our model after some point, and the model with no smoothing included seemed to have slightly better results on several runs. Therefore, we chose it as our final model and trained it from scratch on training + validation data, and tested it on our test set. According to this, the confusion matrix obtained was as follows.

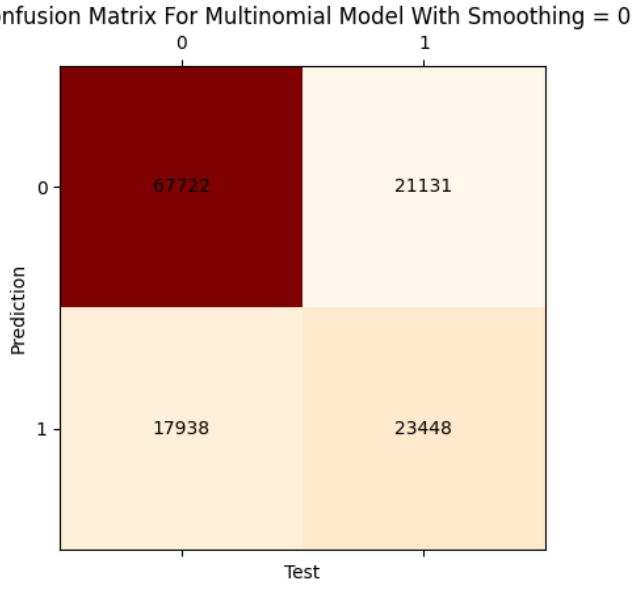


Fig.6: *Confusion Matrix for the Final Multinomiol Model*

```
--------------- Multinomial Model ---------------
The number of correct predictions: 91170, wrong predictions: 39069,
accuracy: 0.7000207311174073
The number of true positives: 23448, true negatives: 67722
The number of false positives: 17938, false negatives: 21131
f1 score for multinomial model with smoothing = 0:  0.5455243413016926
```

7.2.2. Bernoulli Naive Bayes Model

In the second phase of the report, we added some new features related to the count of specific characters such as '+', '#', '/', '$', '!', '*', ',', '_', ':' and so on, which were not directly related to the

structure of the URL. When we added all these features, for the first time, we observed a decrease in the quality of the Bernoulli Model and therefore decided to pick a best-performing subset of these features. In order to test the usefulness of these features, we decided to apply the forward selection method to extract the best set of features on the Bernoulli Model. The Bernoulli Model was picked since these new features mostly take binary values, 0 and 1, making them suitable with Bernoulli Model, and also, there was no parameter to tune in the Bernoulli Model. Besides that, the Bernoulli Model is quite accurate and computationally efficient; the same experiment would take much longer runtimes with Logistic Regression and Neural Network Models. With feature selection, the accuracies have been calculated on the validation dataset, and in each iteration, the character that improves the accuracy the most has been picked.

```
--------------- Bernoulli Model ---------------
Base Acc: 0.8632810700410019

--------------- Iteration 1 ---------------
New highest acc: 0.8642024601114882
New highest acc: 0.8691779664921144
Selected Char: /

--------------- Iteration 2 ---------------
New highest acc: 0.8699765045532026
Selected Char: +

--------------- Iteration 3 ---------------
New highest acc: 0.8699918610543774
New highest acc: 0.8701454260661251
Selected Char: ,

--------------- Iteration 4 ---------------
New highest acc: 0.8701607825672999
Selected Char: !

--------------- Iteration 5 ---------------
Selected Set Of Features:['/', '+', ',', '!']
```

The above selected features have been then added to the train, test, and validation datasets for all machine learning algorithms tested in the project. There is no parameter that we can tune for our Bernoulli Naive Bayes; therefore, we directly trained it on our train and validation and evaluated it on our test set. Below is the confusion matrix and the results of some other metrics:

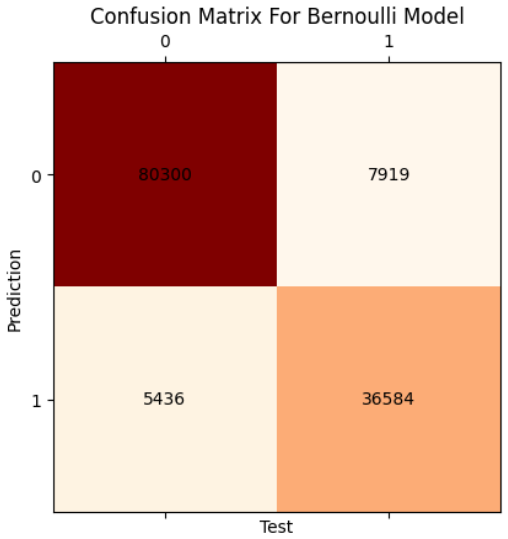


Fig.7: *Confusion Matrix for the Final Bernoulli Model*

```
--------------- Bernoulli Model ---------------
The number of correct predictions: 90257, wrong predictions: 39982,
accuracy: 0.8974577507505432
```

```
The number of true positives: 36584, true negatives: 80300
The number of false positives: 5436, false negatives: 7919
f1 score for bernoulli model:  0.8456479779942905
```

On the machine equipped with Intel i7-10750h, the training of the Bernoulli model took around 0.228995 seconds (Python's *time* library gave this result); still the calculations were quite fast but slower than the Multinomial case.

7.3 Neural Network Model

The neural networks that we tested in the project were fully connected ANN (Artificial Neural Network) models, and we tuned these models to obtain their best performance on our dataset. As our loss function in training, the Binary Cross Entropy loss function has been used as it's appropriate to the task of the project, which is binary classification. Besides, the sigmoid activation function has been used in the output layer in order to map the output of the model to a probability between 0 and 1, and batch gradient descent has been used to update the model. Together with this, a threshold of 0.5 has been chosen for the final classification. Also, in the training process, 500 epochs have been used together with a stopping criterion, which exits training if more than five epochs (patience=5) pass without an increase in the accuracy of less than 1e-7.

When adjusting the necessary hyperparameters to tune the model, there were four important ones, which are the number of neurons in the layers, batch size, momentum, and the learning rate. We also had the option to choose between ReLU and Sigmoid activation functions, but since our dataset is large, the performance of the training was also an important criterion for us. As a result of this, we chose ReLU as our activation function since it's fast to compute ReLU and its derivative. Also, usually, the Sigmoid function saturates and kills gradients, leading to a higher number of training epochs for the model to converge, which makes it an unfavorable option. The number of neurons that we tested were [64,1], [32, 32, 1], [64, 64, 1], where the numbers in the list indicate the number of neurons in the layers of our network. For the batch size, the values in the set {32, 64} have been tested, and for momentum and learning rate, the values from the sets {0.85, 0.95} and {0.01, 0.005, 0.001, 0.0005} have been tested, respectively. The best network has been picked based on the validation accuracy by training each configuration up to 200 epochs (maximum). Also, for the final best neural network model, both accuracy and F-1 score metrics have been reported in order to make a fair comparison with other selected machine learning algorithms. Based on the validation accuracy, the following histogram has been formed showing the distribution of the results for 48 different configurations of the network:

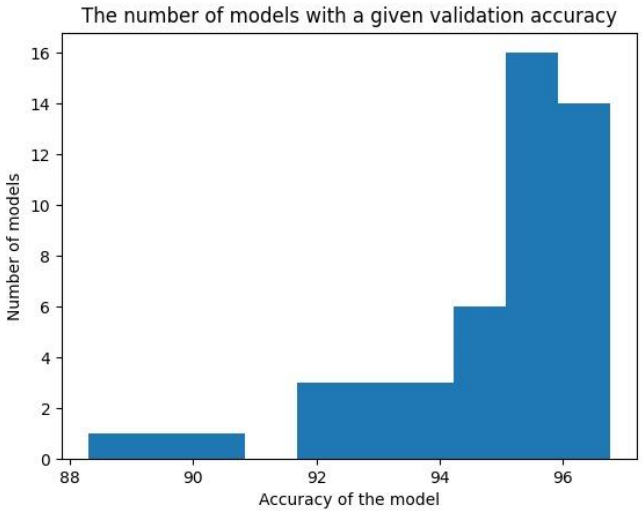


Fig.8: *Histogram Showing The Distribution Of The Model Validation Accuracies For Different Configurations Of Hyperparameters*

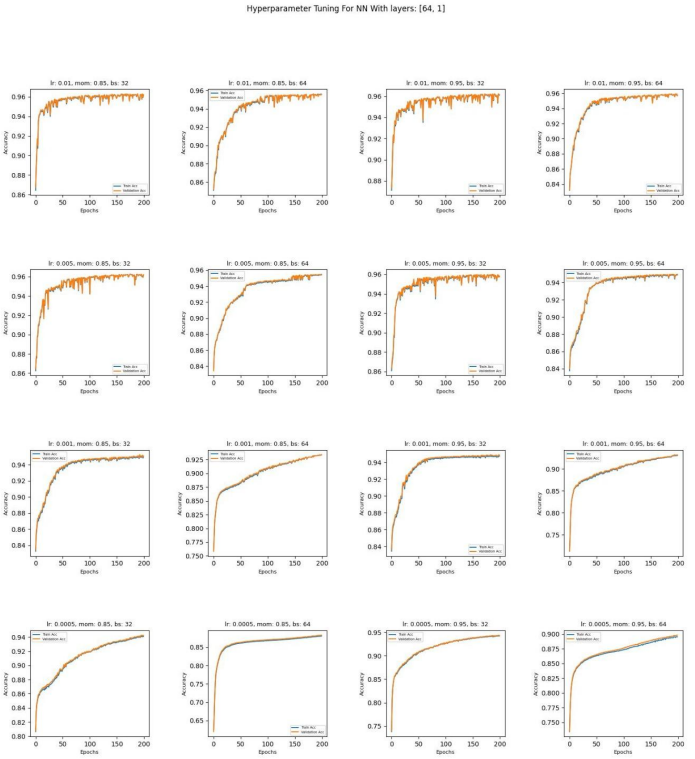


Fig.9: *Plot Showing The Train and Validation Accuracies For Different Parameter Settings For The Model Size [64,1]*

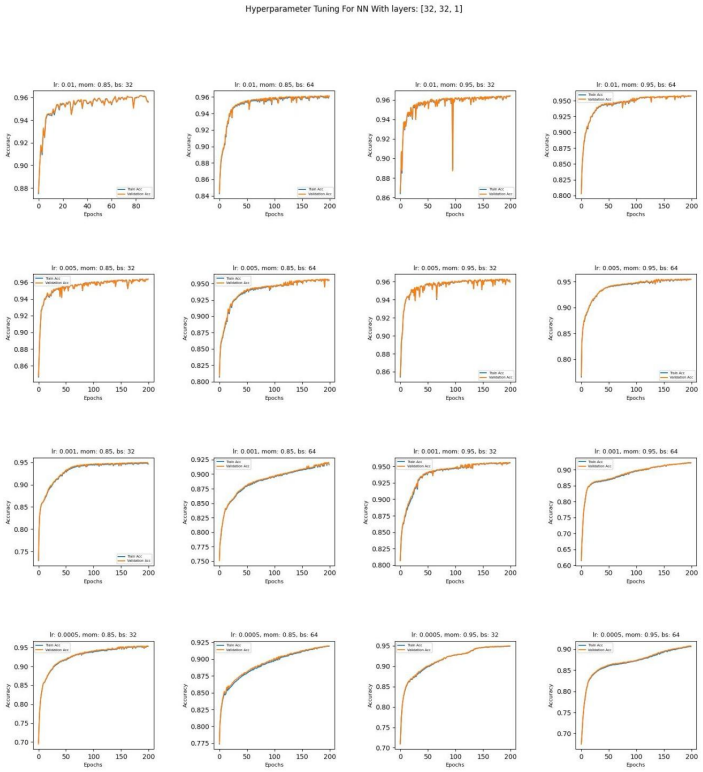


Fig.10: *Plot Showing The Train and Validation Accuracies For Different Parameter Settings For The Model Size [32, 32,1]*

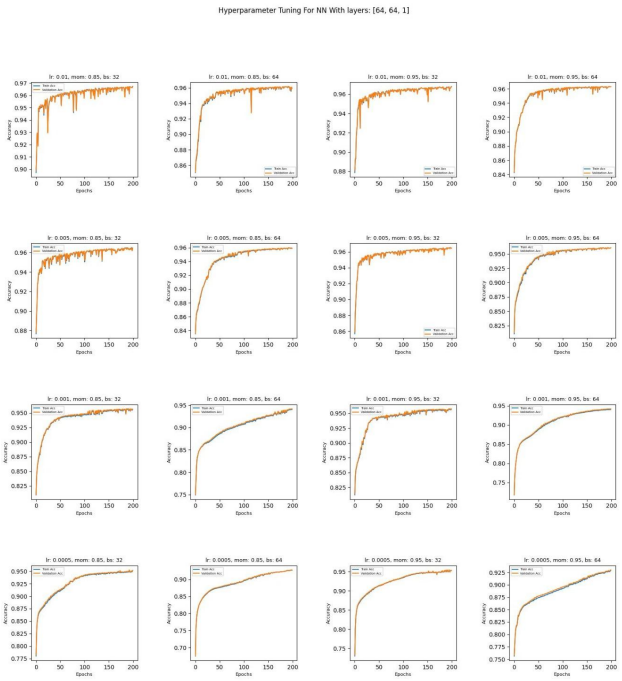


Fig.11: *Plot Showing The Train and Validation Accuracies For Different Parameter Settings For The Model Size [64, 64, 1]*

On the machine equipped with Intel i7-10750h, the training took around 6 and 3.5 seconds for each epoch when batch sizes are 32 and 64, respectively. The plots above illustrate the accuracies corresponding to various hyperparameter configurations for each model size. It can be inferred that the neural network's performance exhibits greater variability with different hyperparameter settings compared to the other machine learning models we investigated. Based on the validation accuracy, the best-performing network has been observed under the following configuration, which has been then trained again on the train + validation dataset and tested on the test dataset:

```
Best Neural Network Settings: Size=[64, 64, 1], Learning Rate=0.01, Batch Size:32, Momentum:0.95
```

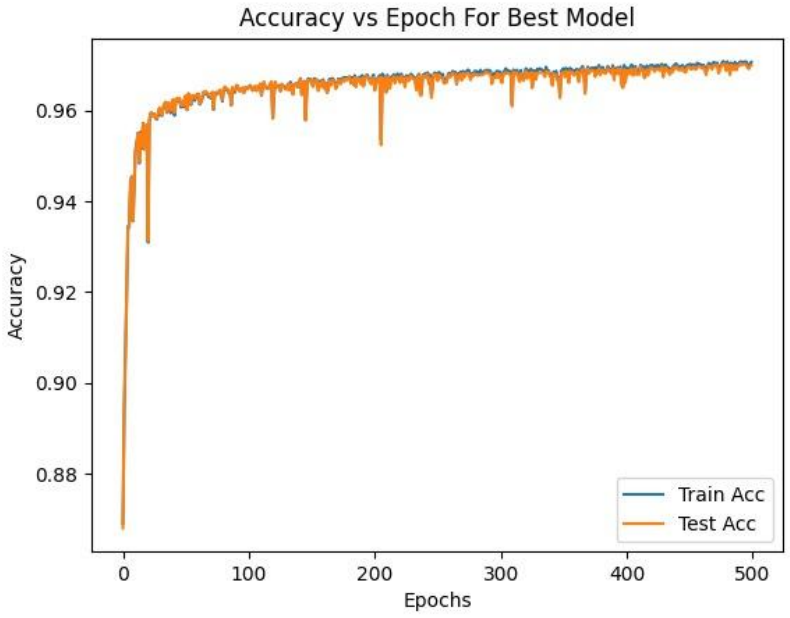


Fig.12: *Plot Showing The Train and Validation Accuracies For Different Parameter Settings For The Model Size [64, 64, 1]*

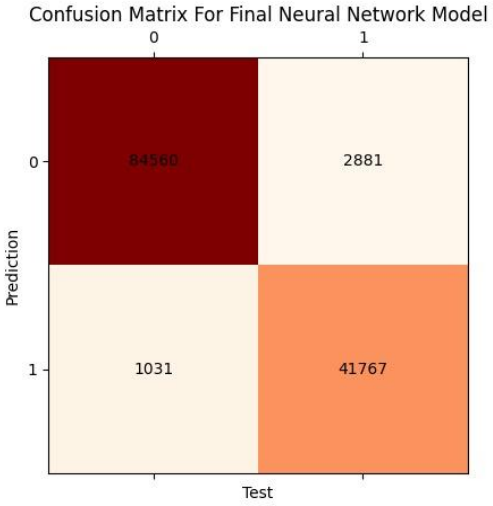


Fig.13: *Plot Showing The Train and Validation Accuracies For Different Parameter Settings For The Model Size [64, 64, 1]*

```
Final Test Accuracy: 0.969963
F1 Score For Final Neural Network Model:  0.9552638199574595
```

From the above plots, we observe that our best model does not overfit to the data after 500 epochs of training and also the train and test accuracies tend to go together though test accuracies are slightly lower. There are a few reasons behind this interesting fact that our model behaves similarly on test and train datasets, which is also the case for different models when tuning the parameters. With a large dataset like the URL dataset (around 600,000 data points), the model has access to a diverse range of examples. This ample data allows the model to learn better representations and generalize well to unseen data. Also, if the train and test datasets are sampled from the same distribution and are representative of the overall dataset, it is likely that the model learns patterns and generalizes well to both sets. Another thing that contributes is the dataset being balanced for two classes which enables the model to learn from a representative number of examples from each class, resulting in better generalization. Also, the complexity of the models tested was proper when the number of data points is considered. Consequently, the train and test accuracies tend to be similar as the model captures the underlying patterns effectively.

The results obtained from the experimentation phase demonstrated that the Neural Network model yielded the highest accuracy of 0.97 and an F-1 score of 0.955, making it the most effective model for URL classification. This outcome highlights the power of Neural Networks in capturing intricate patterns and relationships within the data, enabling them to make accurate predictions.

7.4 Discussion Of The Obtained Results

As expected, we obtained the highest accuracy and highest F-1 scores from the Neural Network, followed by the Logistic Regression Model. Though the Bernoulli model was really close to Logistic Regression in terms of classification accuracy, the Multinomial Model was clearly worse than the Bernoulli Model. The reason behind this might be some of the features that play an important role in explaining the variance, which take values of only 0 and 1, such as *use_of_ip*,*count_http*, *count_https*, and *count_www,* and others. Since Bernoulli only focuses on the absence or presence of the features, these kinds of binary features that we have might have benefited the score of the Bernoulli Model, but at the same time, since the probabilities calculated are relative in the Multinomial Model, the effects of these features might have vanished. This usage of binary features is also not a problem in the Logistic Regression Model, as the model actually learns the ‘importance’ of each feature by estimating the corresponding coefficient for it. In the final report, the classification accuracies for both Bernoulli and Logistic Regression Models have been improved with the extraction of new features. Finally, because of the learning flexibility of neural networks for complex functions, we obtained a higher classification score from the neural networks, yet they required a great effort for

optimization in order to achieve this. Interestingly, our model also did not overfit the data since the large and diverse dataset provides the model with enough examples to learn from, and the models picked were not overly complex relative to the complexity of the problem, mitigating overfitting.

8. Encountered Challenges

We faced a number of challenges during the development of this project. The first and most obvious one was when we were in the early pre-processing stages. Since the original dataset we had did not have any features itself, we had to do plenty of research about URLs in order to come up with features that could actually be useful for our model and in order to achieve this. Once we were done with preprocessing, we started implementing our Logistic Regression model, which we had difficulties implementing from scratch without using predefined methods of some libraries such as scikit-learn. And for the future, we might have to do a more detailed analysis of the features that we use to train our Logistic Regression model, as some of these features may not be as relevant as they are for the Naive Bayes model and do not contribute much to the performance of our Logistic Regression model while increasing it complexity. For our Naive Bayes models, their implementations were comparably more straightforward than Logistic Regression, and we did not face major difficulties.

In our final version, when implementing the Neural Network method, we encountered more challenges as it is known to be more complex, and it's somewhat new for us to implement it from scratch. Besides that, the optimization of the network was compelling as there were many settings for us to tune, and also the training times were another hurdle. Hence, we needed to make some decisions and tune some of the parameters by hand, as the time wouldn't permit experimenting with each possible configuration. Another challenge was to come up with some additional features that could be useful for the training of our models. In order to find relevant features that could be used to determine whether a URL is malicious or not, we did some exhaustive research online and performed some experiments ourselves to see if the newly added features improved the performance of our models or not. As a result of this, we recognized that some of the features harmed the performance of the models, so we had to tackle this problem by selecting a subset of them with a forward selection process.

9. Gantt Chart

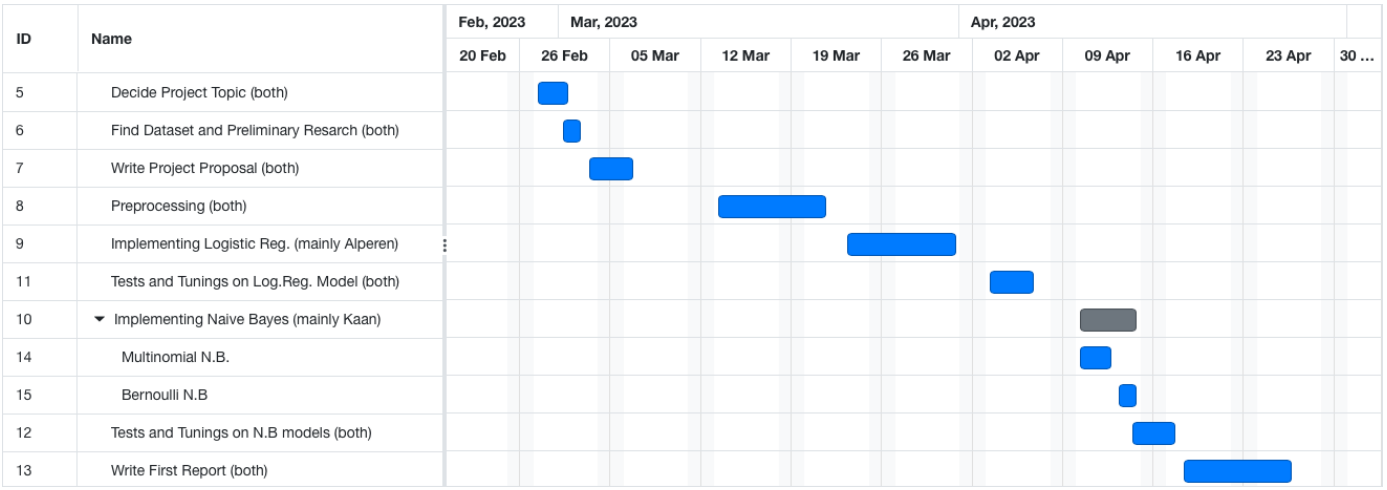


Fig.14: Gantt Chart Showing The Progress Plan Of The URL Classification Project Until The First Report

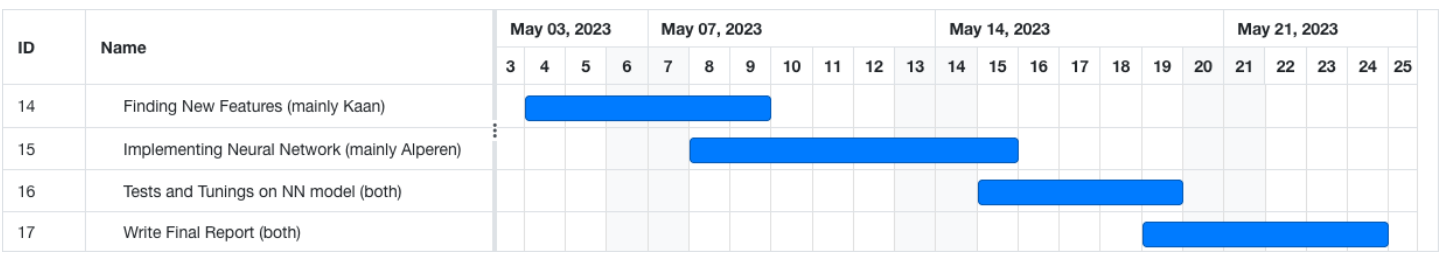


Fig.15: Gantt Chart Showing The Progress Plan Of The URL Classification Project Between First and Final Reports

10.Conclusion

The objective of this project was to classify URLs as either malicious or benign by extracting relevant features and utilizing various machine-learning models. This endeavor aimed to enhance the detection of URL-based threats. Extensive research and literature review were conducted to extract meaningful features from URL strings. The training and testing processes yielded significant results, achieving high accuracy and F-1 scores exceeding 0.9 in correctly classifying URLs. These outcomes underscore the efficacy of Neural Networks in capturing intricate patterns and relationships within the data. Additionally, the probabilistic models and Logistic Regression model demonstrated notable success. Nonetheless, further improvements are required to optimize these models for the URL dataset. Furthermore, employing comprehensive feature selection methods such as forward selection or backward elimination for each machine learning algorithm can help identify the most suitable feature sets. In cases where training times are considerable, such as with Neural Networks, employing statistical feature selection methods may prove advantageous. Advancements can be pursued through the exploration of ensemble methods that harness the strengths of different models. Additionally, incorporating more advanced feature extraction techniques, guided by ongoing research targeting emerging malicious URL techniques, holds promise for further advancements in this field.

11. References

[1] Siddhartha, “Malicious urls dataset,” *Kaggle*, 23-Jul-2021. [Online]. Available: https://www.kaggle.com/datasets/sid321axn/malicious-urls-dataset. [Accessed: 26-Apr-2023].

[2] “URL,” *Wikipedia*, 26-Apr-2023. [Online]. Available: https://en.wikipedia.org/wiki/URL#Syntax. [Accessed: 26-Apr-2023].

[3] sid321axn, “Malicious URL detection using ML & Feat Engg,” *Kaggle*, 23-Jul-2021. [Online]. Available: https://www.kaggle.com/code/sid321axn/malicious-url-detection-using-ml-feat-engg/notebook. [Accessed: 26-Apr-2023].

[4] *IPv4 and IPv6 address formats*. [Online]. Available: https://www.ibm.com/docs/en/ts3500-tape-library?topic=functionality-ipv4-ipv6-address-formats. [Accessed: 26-Apr-2023].

[5] “TLD,” *PyPI*. [Online]. Available: https://pypi.org/project/tld/. [Accessed: 26-Apr-2023].

[6] R. Ramakrishnan, “URL feature engineering and Classification,” *Medium*, 25-May-2021. [Online]. Available: https://medium.com/nerd-for-tech/url-feature-engineering-and-classification-66c0512fb34d. [Accessed: 26-Apr-2023].

[7] 738, “738/awesome-URL-shortener: A curated list of awesome URL shortener,” *GitHub*. [Online]. Available: https://github.com/738/awesome-url-shortener. [Accessed: 26-Apr-2023].

[8] R. E. Ikwu, “Extracting feature vectors from URL strings for malicious URL detection,” *Medium*. [Online]. Available: https://towardsdatascience.com/extracting-feature-vectors-from-url-strings-for-malicious-url-detection-cbafc24737a [Accessed: 23-May-2023].

12. Appendix

utils.py

utils.py > ...

```python
import re
import pandas as pd

from tld import get_tld
from typing import Tuple, Union
from urllib.parse import urlparse

def contains_ip_address(url: str) -> bool:
    contains_ip = re.search(
        '(([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.'
        '([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\/)|'  # IPv4
        '(([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.'
        '([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\/)|'  # IPv4 with port
        '((0x[0-9a-fA-F]{1,2})\\.(0x[0-9a-fA-F]{1,2})\\.(0x[0-9a-fA-F]{1,2})\\.(0x[0-9a-fA-F]{1,2})\\/)' # IPv4 in hexadecimal
        '(?:[a-fA-F0-9]{1,4}:){7}[a-fA-F0-9]{1,4}|'
        '([0-9]+(?:\.[0-9]+){3}:[0-9]+)|'
        '((?:(?:\d|[01]?\d\d|2[0-4]\d|25[0-5])\.){3}(?:25[0-5]|2[0-4]\d|[01]?\d\d|\d)(?:\/\d{1,2})?)', url)  # Ipv6
    if contains_ip:
        return 1
    return 0

""" returns full length domain (fld), top level domain (tld), domain, and
subdomain names for a given URL. These can be useful as features """
def extract_tld(url: str, fix_protos: bool = False) -> Tuple[str, str, str, str]:
    res = get_tld(url, as_object = True, fail_silently=True, fix_protocol=fix_protos)

    fld = res.fld
    tld = res.tld
    domain = res.domain
    subdomain = res.subdomain

    return subdomain, domain, tld, fld

""" takes an instance and checks if the url is an IP address. If yes,
set four instances to None. If not, call extract_tld to process URL"""
def process_url(data: pd.Series) -> Tuple[str, str, str, str]:
    try:
        if data['use_of_ip'] == 0:
            if str(data['url']).startswith('http:'):
                return extract_tld(data['url'])
            return extract_tld(data['url'], fix_protos=True)
```

utils.py > ...

```python
            return None, None, None, None
    except:
        return None, None, None, None

""" returns the path of the url
for example: www.bilkent.com/srs/cgpa -> srs/cgpa """
def get_path(url: str) -> Union[str, None]:

    try:
        res = get_tld(url, as_object = True, fail_silently=True, fix_protocol=True)
        if res.parsed_url.query:
            joined = res.parsed_url.path + res.parsed_url.query
            return joined
        else:
            return res.parsed_url.path
    except:
        return None

""" returns letter count """
def letter_count(url: str) -> int:

    letter_count = 0
    for c in url:
        if c.isalpha():
            letter_count += 1
    return letter_count

""" return digits count """
def digit_count(url: str) -> int:

    d_count = 0
    for d in url:
        if d.isnumeric():
            d_count += 1
    return d_count

""" the number of '/' characters gives the number of subdirectories """
def sub_directory_count(url: Union[str, None]):
    if url:
        dir_count = url.count('/')
        return dir_count
```

utils.py > ...

```python
        return dir_count
    return 0

""" return length of the first directory """
def len_first_directory(url: Union[str, None]):
    if url:
        if len(url.split('/')) > 1:
            first_dir_len = len(url.split('/')[1])
            return first_dir_len
    return 0

""" check whether the URL has a shortening service or not
Shortening Service: a third-party website that converts that long URL to a short,
case-sensitive alphanumeric code."""
def check_shortening_service(url: str) -> int:

    check = re.search('bit\.ly|goo\.gl|shorte\.st|go2l\.ink|x\.co|ow\.ly|t\.co|tinyurl|tr\.im|is\.gd|cli\.gs|'
                      'yfrog\.com|migre\.me|ff\.im|tiny\.cc|url4\.eu|twit\.ac|su\.pr|twurl\.nl|snipurl\.com|'
                      'short\.to|BudURL\.com|ping\.fm|post\.ly|Just\.as|bkite\.com|snipr\.com|fic\.kr|loopt\.us|'
                      'doiop\.com|short\.ie|kl\.am|wp\.me|rubyurl\.com|om\.ly|to\.ly|bit\.do|t\.co|lnkd\.in|'
                      'db\.tt|qr\.ae|adf\.ly|goo\.gl|bitly\.com|cur\.lv|tinyurl\.com|ow\.ly|bit\.ly|ity\.im|'
                      'q\.gs|is\.gd|po\.st|bc\.vc|twitthis\.com|u\.to|j\.mp|buzurl\.com|cutt\.us|u\.bb|yourls\.org|'
                      'x\.co|prettylinkpro\.com|scrnch\.me|filoops\.info|vzturl\.com|qr\.net|1url\.com|tweez\.me|v\.gd|'
                      'tr\.im|link\.zip\.net', url)
    return 1 if check else 0

""" check whether the URL starts with 'http' or 'https' """
def check_http(url):
    htp = urlparse(url).scheme
    check = str(htp)

    return 1 if check == 'https' else 0

def suspicious_words(url):
    match = re.search('client|admin|server|cash|PayPal|login|signin|bank|account|update|free|lucky|service|bonus|ebayisapi|sex|webscr', url)

    return 1 if match else 0
```

preprocessing.py

preprocessing.py > ...

```python
import pandas as pd
from utils import contains_ip_address, process_url, get_path, check_shortening_service, sub_directory_count, letter_count, digit_count, len_first_directory, check_http,
from sklearn.preprocessing import OrdinalEncoder

df = pd.read_csv("data/malicious_phish.csv")

print(df.head())

# create a column named is_ip by using the ip check function
df['use_of_ip'] = df['url'].apply(lambda i: contains_ip_address(i))

# get the number of URLs containing IP address
print(df['use_of_ip'].value_counts())

# an example url starting with the ip address
print(df[df['use_of_ip'] == 1].iloc[0]['url'])

print(df['url'][1])

# add new features
df[['subdomain', 'domain', 'tld', 'fld']] = df.apply(lambda x: process_url(x), axis=1, result_type="expand")

# General Features
df['url_path'] = df['url'].apply(lambda x: get_path(x))
df['use_of_shortener'] = df['url'].apply(lambda x: check_shortening_service(x))

# URL component length
df['url_length'] = df['url'].apply(lambda x: len(str(x)))
df['subdomain_length'] = df['subdomain'].apply(lambda x: len(str(x)))
df['tld_length'] = df['tld'].apply(lambda x: len(str(x)))
df['fld_length'] = df['fld'].apply(lambda x: len(str(x)))
df['path_length'] = df['url_path'].apply(lambda x: len(str(x)))

# count the number of alphanumeric characters, digits, and punctuations
df['count_letters']= df['url'].apply(lambda i: letter_count(i))
df['count_digits']= df['url'].apply(lambda i: digit_count(i))
df['count_puncs'] = (df['url_length'] - (df['count_letters'] + df['count_digits']))

df['is_php'] = df['url'].apply(lambda a: a.count('.php'))
df['count_%20'] = df['url'].apply(lambda a: a.count('%20')) # usually effects the model in a bad way

```

preprocessing.py > ...

```python
# check special character counts for the url
for c in ".@-%?=":
    df['count' + c] = df['url'].apply(lambda a: a.count(c))
    # print(df['count'+c][:5])

# extra features that are not directly related
# with the structure of a URL
featureList = ['+', '#', '/', '$', '!', '*', ',', '_', ':']
for c in featureList:
    df['count' + c] = df['url'].apply(lambda a: a.count(c))

df['is_susp'] = df['url'].apply(lambda i: suspicious_words(i))

# Binary Label by converting benign to 0 and all other classes to 1
df['is_malicious'] = df['type'].apply(lambda x: 0 if x == 'benign' else 1)

# Inspect the malicious URLs
print(df[df['is_malicious'] == 1].head())

# the number of subdirectories (number of /) and the length can be helpful
df['count_dirs'] = df['url_path'].apply(lambda x: sub_directory_count(x))
df['first_dir_length'] = df['url_path'].apply(lambda x: len_first_directory(x))

# Binned Features
groups = ['Short', 'Medium', 'Long', 'Very Long']
# URL Lengths in 4 bins
df['url_length_q'] = pd.qcut(df['url_length'], q=4, labels=groups)
# FLD Lengths in 4 bins
df['fld_length_q'] = pd.qcut(df['fld_length'], q=4, labels=groups)

# Counts for https, http, www
df['https'] = df['url'].apply(lambda i: check_http(i))
df['count-https'] = df['url'].apply(lambda a: a.count('https'))
df['count-http'] = df['url'].apply(lambda a: a.count('http'))
df['count-www'] = df['url'].apply(lambda a: a.count('www'))

# Percentage Features
df['letters_ratio'] = df['count_letters'] / df['url_length']
df['digit_ratio'] = df['count_digits'] / df['url_length']
df['punc_ratio'] = df['count_puncs'] / df['url_length']

```

preprocessing.py > ...

```python
enc = OrdinalEncoder()
df[["url_length_q","fld_length_q"]] = enc.fit_transform(df[["url_length_q","fld_length_q"]])

print(df.head())

# write back the file
df.to_csv('data/url_processed.csv', index=False)
```

naive_bayes.py

naive_bayes.py > ...

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time

df = pd.read_csv("data/url_processed.csv")

# shuffle the data frame
df = df.sample(frac = 1)

# pick only the numeric features
X = df[['use_of_ip', 'url_length', 'subdomain_length', 'tld_length', 'fld_length', 'path_length',
        'count_letters', 'count_digits', 'count_puncs', 'count.', 'count@', 'count-',
        'count%', 'count?', 'count=', 'count_dirs', 'use_of_shortener', 'first_dir_length',
        'url_length_q', 'fld_length_q', 'https', 'count-https', 'count-http', 'count-www', 'is_susp', 'is_php']]

print(X.head())

y = df['is_malicious']

print(X.isna().sum())

# fill the nan values with 0
X = X.fillna(0)

# check again the number of nan values
print(X.isna().sum())

X.replace([np.inf, -np.inf], 0, inplace=True)

X['first_dir_length'] = X['first_dir_length'].astype('int64')
X['url_length_q'] = X['url_length_q'].astype('int64')
X['fld_length_q'] = X['fld_length_q'].astype('int64')

# divide the data into train, test and validation datasets
n = len(y)

train_size = int(n * 0.7)
valid_size = int(n * 0.1)
test_size = int(n * 0.2)

```

naive_bayes.py > ...

```
X_train = X.iloc[:train_size]
y_train = y.iloc[:train_size]

X_valid = X.iloc[train_size: train_size + valid_size]
y_valid = y.iloc[train_size: train_size + valid_size]

X_test = X.iloc[train_size + valid_size:]
y_test = y.iloc[train_size + valid_size:]

print(X_train.shape, X_valid.shape, X_test.shape)

print(X_train.head())

# convert df into numpy
x_train = X_train.to_numpy()
x_valid = X_valid.to_numpy()
x_test = X_test.to_numpy()
y_train = y_train.to_numpy()
y_test = y_test.to_numpy()
y_valid = y_valid.to_numpy()

# calculate the spam and normal class probabilities
spam_count = np.count_nonzero(y_train)
normal_count = y_train.shape[0] - spam_count
p_spam = spam_count / y_train.shape[0]
p_normal = 1 - p_spam

# print the spam percentage
print("Spam percentage: " + str((p_spam)*100))

# add the number of words for each email
word_counts = np.sum(x_train, axis = 1)

# Get the total word counts for categories
spamWordCount = word_counts.T @ y_train
normalWordCount = word_counts.T @ (1 - y_train)

start_time = time.time()
# Get the frequencies for each word seperately
spamFrequencies = y_train.T @ x_train
normalFrequencies = (1 - y_train.T) @ x_train
```

naive_bayes.py > ...

```
normalFrequencies = (1 - y_train.T) @ x_train

# Calculate the parameters by dividing with the total word count
spamProbabilities = spamFrequencies / spamWordCount
normalProbabilities = normalFrequencies / normalWordCount

# Take the log of the probabilities
with np.errstate(divide='ignore'):
    logSpamProbabilities = np.log(spamProbabilities)
    logNormalProbabilities = np.log(normalProbabilities)

logSpamProbabilities[np.isneginf(logSpamProbabilities)]= -1e+12
logNormalProbabilities[np.isneginf(logNormalProbabilities)]= -1e+12
print("--- %s seconds ---" % (time.time() - start_time))

multinomial_acc_history = []
def fit_smoothened_multinomial(alpha, spamFrequencies, normalFrequencies, x_train, x_test, y_test):
    regularizedSpamFrequencies = spamFrequencies + alpha
    regularizedNormalFrequencies = normalFrequencies + alpha

    # regularize the word counts for spam and normal categories, as well
    regularizedSpamWordCount = spamWordCount + alpha * x_train.shape[1]
    regularizedNormalWordCount = normalWordCount + alpha * x_train.shape[1]

    # divide by the total word count to find the probabilities
    regularizedSpamProbabilities = regularizedSpamFrequencies / regularizedSpamWordCount
    regularizedNormalProbabilities = regularizedNormalFrequencies / regularizedNormalWordCount

    # take the logarithm of the probabilities
    regularizedLogSpamProbabilities = np.log(regularizedSpamProbabilities)
    regularizedLogNormalProbabilities = np.log(regularizedNormalProbabilities)

    num_correct = tp = tn = fp = fn = 0
    for i in range(x_test.shape[0]):
        row = x_test[i]
        probSpam = np.log(p_spam)
        probNormal = np.log(p_normal)

        probSpam += (regularizedLogSpamProbabilities @ row)
        probNormal += (regularizedLogNormalProbabilities @ row)
```

naive_bayes.py > ...

```
            predicted = 0
            if probSpam > probNormal:
                predicted = 1

            if predicted == y_test[i]:
                num_correct += 1
                if predicted == 1:
                    tp += 1
                else:
                    tn += 1
            else:
                if predicted == 1:
                    fp += 1
                else:
                    fn += 1

        print("--------------- Multinomial Model With Additive Smoothing alpha = %d ---------------" % (alpha))
        print("The number of correct predictions: " + str(num_correct) + ", wrong predictions: " + str(x_test.shape[0] - num_correct) + ", accuracy: " + str(num_correct / x
        print("The number of true positives: " + str(tp) + ", true negatives: " + str(tn))
        print("The number of false positives: " + str(fp) + ", false negatives: " + str(fn))
        multinomial_acc_history.append(num_correct / x_test.shape[0]) # add the accuracy to the history

    # apply smoothing and compare on the validation set to pick the best smoothing
    for i in range(0, 10, 2):
        fit_smoothened_multinomial(i, spamFrequencies, normalFrequencies, x_train, x_valid, y_valid)

    # plotting the validation accuracy for different smoothing parameter values
    x = list(range(0, 10, 2))
    plt.figure(figsize=(18, 12))
    plt.title('Smoothing Parameter Comparison')
    plt.xlabel('Smoothing Parameter Alpha')
    plt.ylabel('Validation accuracy')
    plt.plot(x, multinomial_acc_history,  '-o', label='Multinomial Model')
    plt.yticks([x / 5 for x in range(0, 6)])
    for (parameter, acc) in zip(x, multinomial_acc_history):
        plt.text(parameter, acc, "{:.6f}".format(acc), va='bottom', ha='center')
    plt.legend()
    plt.title('Validation Accuracy For Multinomial Model')
    plt.show()

    # The best behaving multinomial model is one with smoothing = 0, report its accuracy again
```

naive_bayes.py > ...

```
    # by training on the train + validation set
    x_train = np.concatenate((x_train, x_valid), axis=0)
    y_train = np.concatenate((y_train, y_valid), axis=0)

    # calculate the spam and normal class probabilities
    spam_count = np.count_nonzero(y_train)
    normal_count = y_train.shape[0] - spam_count
    p_spam = spam_count / y_train.shape[0]
    p_normal = 1 - p_spam

    # print the spam percentage
    print("Spam percentage: " + str((p_spam)*100))

    # add the number of words for each email
    word_counts = np.sum(x_train, axis = 1)

    # Get the total word counts for categories
    spamWordCount = word_counts.T @ y_train
    normalWordCount = word_counts.T @ (1 - y_train)

    # Get the frequencies for each word seperately
    spamFrequencies = y_train.T @ x_train
    normalFrequencies = (1 - y_train.T) @ x_train

    # Calculate the parameters by dividing with the total word count
    spamProbabilities = spamFrequencies / spamWordCount
    normalProbabilities = normalFrequencies / normalWordCount

    # Take the log of the probabilities
    with np.errstate(divide='ignore'):
        logSpamProbabilities = np.log(spamProbabilities)
        logNormalProbabilities = np.log(normalProbabilities)

    logSpamProbabilities[np.isneginf(logSpamProbabilities)]= -1e+12
    logNormalProbabilities[np.isneginf(logNormalProbabilities)]= -1e+12

    num_correct = tp = tn = fp = fn = 0
    y_pred = []
    for i in range(x_test.shape[0]):
        row = x_test[i]
        probSpam = np.log(p_spam)
```

naive_bayes.py > ...

```
        probNormal = np.log(p_normal)

        probSpam += (logSpamProbabilities @ row)
        probNormal += (logNormalProbabilities @ row)

        predicted = 0
        if probSpam > probNormal:
            predicted = 1

        y_pred.append(predicted)
        if predicted == y_test[i]:
            num_correct += 1
            if predicted == 1:
                tp += 1
            else:
                tn += 1
        else:
            if predicted == 1:
                fp += 1
            else:
                fn += 1
    print("--------------- Multinomial Model ---------------")
    print("The number of correct predictions: " + str(num_correct) + ", wrong predictions: " + str(x_test.shape[0] - num_correct) + ", accuracy: " + str(num_correct / x_tes
    print("The number of true positives: " + str(tp) + ", true negatives: " + str(tn))
    print("The number of false positives: " + str(fp) + ", false negatives: " + str(fn))

    # plot confusion matrix
    confusion = pd.crosstab(y_test, y_pred)
    fig, ax = plt.subplots()
    ax.matshow(confusion,cmap='OrRd')
    ax.set(xlabel='Test', ylabel='Prediction')

    for i in range(2):
      for j in range(2):
        c = confusion[j][i]
        ax.text(i, j, str(c), va='center', ha='center')
    plt.title('Confusion Matrix For Multinomial Model With Smoothing = 0')
    plt.show()

    # print f1 score = 2 * precision * recall / (precision +  recall)
    precision = tp / (fp + tp)
```

naive_bayes.py > ...

```
    precision = tp / (fp + tp)
    recall = tp / (tp + fn)
    f1_score = (2 * precision * recall) / (precision + recall)
    print("f1 score for multinomial model with smoothing = 0: ", f1_score)

    """
    Bernoulli Model is trained on train + validation and its accuracy is reported
    on the test dataset since it doesn't have any parameters to tune
    """
    start_time = time.time()
    def fit_bernoulli(x_train, x_test, y_train, y_test):
        # create a copy of the train and test datasets
        bernoulli_x_train = np.copy(x_train)
        bernoulli_x_test = np.copy(x_test)

        # make nonzero positions 1 for the bernoulli model
        bernoulli_x_train[bernoulli_x_train != 0] = 1
        bernoulli_x_test[bernoulli_x_test != 0] = 1

        # get the frequencies of the words for spam and normal categories
        bernoulliSpamFrequencies = y_train.T @ bernoulli_x_train
        bernoulliNormalFrequencies = (1 - y_train.T) @ bernoulli_x_train

        bernoulliSpamProbabilities = bernoulliSpamFrequencies / spam_count
        bernoulliNormalProbabilities = bernoulliNormalFrequencies / normal_count

        #print("--- %s seconds ---" % (time.time() - start_time))
        num_correct = tp = tn = fp = fn = 0
        y_pred_berno = []
        for i in range(bernoulli_x_test.shape[0]):
            row = bernoulli_x_test[i]
            probSpam = np.log(p_spam)
            probNormal = np.log(p_normal)

            # calculate the spam probabilities for each word separately
            spamWordExists = (bernoulliSpamProbabilities * row)
            spamWordDoesntExist = ((1 - bernoulliSpamProbabilities) * (1 - row))
            spamWord = spamWordDoesntExist + spamWordExists
            with np.errstate(divide='ignore'):
                spamWord = np.log(spamWord)
            spamWord[np.isneginf(spamWord)] = -1e+12 # replace 0 with extremely small values
```

naive_bayes.py > ...

```
            spamWord[np.isneginf(spamWord)] = -1e+12 # replace 0 with extremely small values
            probSpam += np.sum(spamWord, axis=0)

            # calculate the normal probabilities for each word separately
            normalWordExists = (bernoulliNormalProbabilities * row)
            normalWordDoesntExist = ((1 - bernoulliNormalProbabilities) * (1 - row))
            normalWord = normalWordDoesntExist + normalWordExists
            with np.errstate(divide='ignore'):
                normalWord = np.log(normalWord)
            normalWord[np.isneginf(normalWord)] = -1e+12 # replace 0 with extremely small values
            probNormal += np.sum(normalWord, axis=0)

            predicted = 0
            if probSpam > probNormal:
                predicted = 1

            y_pred_berno.append(predicted)
            if predicted == y_test[i]:
                num_correct += 1
                if predicted == 1:
                    tp += 1
                else:
                    tn += 1
            else:
                if predicted == 1:
                    fp += 1
                else:
                    fn += 1

    return y_pred_berno, tp, tn, fp, fn

x_train = X.iloc[:train_size].to_numpy()
x_valid = X.iloc[train_size: train_size + valid_size].to_numpy()
y_train = y.iloc[:train_size].to_numpy()
y_valid = y.iloc[train_size: train_size + valid_size].to_numpy()

# get a base accuracy score
y_pred_berno, tp, tn, fp, fn = fit_bernoulli(x_train, x_valid, y_train, y_valid)

# list of possible features
featureList = ['+', '#', '/', '$', '!', '*', ',', '_', ':']
```

naive_bayes.py > ...

```
selectedFeatures = []
X_current = X.copy()
noIncrease = False
iterationNo = 1
maxAcc = (tp + tn) / (tp + tn + fp + fn) # initial accuracy
print('\n--------------- Bernoulli Model ---------------\nBase Acc: ' + str(maxAcc))

# Apply forward selection to get the best features
while not noIncrease:
    noIncrease = True
    maxChar = None

    print(f'\n--------------- Iteration {iterationNo} ---------------')
    iterationNo += 1
    for c in featureList:
        if c not in selectedFeatures:
            X_new = X_current.copy()
            X_new['count'+c] = df['count'+c]
            x_train_new = X_new.iloc[:train_size]
            x_valid_new = X_new.iloc[train_size: train_size + valid_size]
            y_pred_berno, tp, tn, fp, fn = fit_bernoulli(x_train_new, x_valid_new, y_train, y_valid)
            acc = (tp + tn) / (tp + tn + fp + fn)

            if acc > maxAcc:
                maxAcc = acc
                noIncrease = False
                maxChar = c
                print('New highest acc: ' + str(maxAcc))

    if maxChar is not None:
        selectedFeatures.append(maxChar)
        X_current['count'+maxChar] = df['count'+maxChar]
        print('Selected Char: ' + str(maxChar))

print('Selected Set Of Features:' + str(selectedFeatures))
print(X_current.head())

# The best behaving multinomial model is one with smoothing = 0, report its accuracy again
# by training on the train + validation set
x_train = X_current.iloc[:train_size + valid_size].to_numpy()
```

naive_bayes.py > ...

```python
x_train = X_current.iloc[:train_size + valid_size].to_numpy()
x_test = X_current.iloc[train_size + valid_size:].to_numpy()
y_train = y.iloc[:train_size + valid_size].to_numpy()

y_pred_berno, tp, tn, fp, fn = fit_bernoulli(x_train, x_test, y_train, y_test)
print("--------------- Bernoulli Model ---------------")
print("The number of correct predictions: " + str(num_correct) + ", wrong predictions: " + str(x_test.shape[0] - num_correct) + ", accuracy: " + str((tp + tn) / x_test.
print("The number of true positives: " + str(tp) + ", true negatives: " + str(tn))
print("The number of false positives: " + str(fp) + ", false negatives: " + str(fn))

# plot confusion matrix
confusion = pd.crosstab(y_test, y_pred_berno)
fig, ax = plt.subplots()
ax.matshow(confusion,cmap='OrRd')
ax.set(xlabel='Test', ylabel='Prediction')

for i in range(2):
  for j in range(2):
    c = confusion[j][i]
    ax.text(i, j, str(c), va='center', ha='center')
plt.title('Confusion Matrix For Bernoulli Model')
plt.show()

# print f1 score = 2 * precision * recall / (precision +  recall)
precision = tp / (fp + tp)
recall = tp / (tp + fn)
f1_score = (2 * precision * recall) / (precision + recall)
print("f1 score for bernoulli model: ", f1_score)
```

log_regression.py

log_regression.py > ...

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from typing import Literal, Tuple
from tqdm import tqdm

df = pd.read_csv("data/url_processed.csv")
np.random.seed(2023)

# shuffle the data frame
df = df.sample(frac = 1)

# our feature matrix
X = df[['use_of_ip', 'url_length', 'subdomain_length', 'tld_length', 'fld_length', 'path_length',
        'count_letters', 'count_digits', 'count_puncs', 'count.', 'count@', 'count-',
        'count%', 'count?', 'count+', 'count/', 'count,', 'count!', 'letters_ratio', 'digit_ratio', 'punc_ratio', 'count_dirs',
        'use_of_shortener', 'first_dir_length',
        'url_length_q', 'fld_length_q', 'https', 'count-https', 'count-http', 'count-www', 'is_susp', 'is_php']]

print(X.head())

# labels
y = df['is_malicious']

print(X.isna().sum())

# fill the nan values with 0
X = X.fillna(0)

# check again the number of nan values
print(X.isna().sum())

# divide the data into train, test and validation datasets
length = len(y)

# %70 train, %10 validation, %20 test
train_set_size = int(length * 0.7)
valid_set_size = int(length * 0.1)
test_set_size = int(length * 0.2)

```

log_regression.py > ...

```python
X_train = X.iloc[:train_set_size]
y_train = y.iloc[:train_set_size]

X_valid = X.iloc[train_set_size: train_set_size + valid_set_size]
y_valid = y.iloc[train_set_size: train_set_size + valid_set_size]

X_test = X.iloc[train_set_size + valid_set_size:]
y_test = y.iloc[train_set_size + valid_set_size:]

print(X_train.shape, X_valid.shape, X_test.shape)

print(X_train.head())

## Implement the Logistic Regression model

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def calc_accuracy(y_true, y_pred):
    y_true = np.asarray(y_true)
    y_pred = np.asarray(y_pred)
    return np.mean(y_true == y_pred)

# logistic regression class
class LogisticRegression:

    # param b : y interception
    # param W : weights
    # initializer : initial distribution of weights
    def __init__(self, initializer: Literal['normal', 'uniform', 'zeros']):

        self._b = None
        self._W = None
        self.initializer = initializer

    @property
    def b(self) -> np.ndarray:
        return self._b

    @property
    def W(self) -> np.ndarray:
```

log_regression.py > ...

```python
    @property
    def W(self) -> np.ndarray:
        return self._W

    def __repr__(self) -> str:
        return f'LogisticRegression(initializer={self.initializer})'

    def __str__(self) -> str:
        # calls __repr__
        return repr(self)

    # initializes the parameters according to given features
    def init_params(self, in_features: int,
                    initializer: Literal['normal', 'uniform', 'zeros'] = None):
        if initializer is None:
            initializer = self.initializer
        if initializer == 'zeros':
            self._b = 0
            self._W = np.zeros(in_features)
        else:
            random = np.random.default_rng()
            if initializer == 'uniform':
                self._b = random.uniform(-0.01, 0.01, size=1)
                self._W = random.uniform(-0.01, 0.01, size=in_features)
            elif initializer == 'normal':
                self._b = random.normal(0, 1, size=1)
                self._W = random.normal(0, 1, size=in_features)


    # calculates the probability for the output of the sigmoid function
    def __call__(self, X: np.ndarray) -> np.ndarray:
        return sigmoid(self.b + X @ self.W)

    # fits the model
    def fit(self, X: np.ndarray, y: np.ndarray, X_valid: np.ndarray, y_valid: np.ndarray, epochs: int = 1,
            batch_size: int = None, learning_rate: float = 0.01, shuffle: bool = True) -> np.ndarray:

        # X : design matrix
        # y : label vector
        # X_valid : validation set design matrix
        # y_valid : validation set label vector
        X = np.asarray(X)
```

log_regression.py > ...

```
        X = np.asarray(X)
        y = np.asarray(y)
        X_valid = np.asarray(X_valid)
        y_valid = np.asarray(y_valid)

        if self.b is None or self.W is None:
            self.init_params(X.shape[-1])

        if batch_size is None:
            batch_size = len(y)
        n_batches = len(y) // batch_size

        acc_log = calc_accuracy(y_valid, self.predict(X_valid))
        history = [acc_log]
        for _ in tqdm(range(epochs)):
            idx = np.random.permutation(len(y)) if shuffle else np.arange(len(y))
            for batch in range(n_batches):
                batch_idx = idx[batch * batch_size: (batch + 1) * batch_size]
                X_batch = X[batch_idx]
                y_batch = y[batch_idx]
                grad_b, grad_W = self._calc_gradients(X_batch, y_batch)
                self._b = self._b - learning_rate * grad_b
                self._W = self._W - learning_rate * grad_W
            acc_log = calc_accuracy(y_valid, self.predict(X_valid))
            history.append(acc_log)

        return np.asarray(history)

    # predict the output with respect to some threshold
    def predict(self, X: np.ndarray, threshold: float = 0.5) -> np.ndarray:
        return np.asarray(self(X) > threshold, dtype=np.int32)

    # calculate gradients and update later
    def _calc_gradients(self, X: np.ndarray, y: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
        y_pred = self(X)
        grad_b = np.mean(y_pred - y)
        grad_W = X.T @ (y_pred - y) / len(y)
        return grad_b, grad_W

### Train the logistic regression model with changing batch sizes
```

log_regression.py > ...

```
acc_batch_normal = []
acc_batch_uniform = []
acc_batch_zeros = []
def logistic_reg_trainer(initializer, batch_size):
    model = LogisticRegression(initializer)
    acc = model.fit(X_train, y_train,
                    X_valid, y_valid,
                    epochs=20,
                    batch_size=batch_size,
                    learning_rate=1e-3)
    if initializer == 'normal' : acc_batch_normal.append(acc)
    if initializer == 'uniform' : acc_batch_uniform.append(acc)
    if initializer == 'zeros' : acc_batch_zeros.append(acc)

fig, axs = plt.subplots(2, 2, figsize=(18, 12))
axs[1, 1].remove()

# weights initialized NORMAL dist with different batches
logistic_reg_trainer('normal', 32)
logistic_reg_trainer('normal', 64)
logistic_reg_trainer('normal', 128)
logistic_reg_trainer('normal', 256)
# plot normal
x = list(range(20 + 1))
#plt.figure(figsize=(18, 12))
axs[0,0].set_title('Batch Size Comparison for Normal initializer')
axs[0,0].set_xlabel('Epochs')
axs[0,0].set_ylabel('Validation accuracy')
axs[0,0].plot(x, acc_batch_normal[0], label='Batch size = 32')
axs[0,0].plot(x, acc_batch_normal[1], label='Batch size = 64')
axs[0,0].plot(x, acc_batch_normal[2], label='Batch size = 128')
axs[0,0].plot(x, acc_batch_normal[3], label='Batch size = 256')
axs[0,0].legend()

# weights initialized UNIFORM dist with different batches
logistic_reg_trainer('uniform', 32)
logistic_reg_trainer('uniform', 64)
logistic_reg_trainer('uniform', 128)
logistic_reg_trainer('uniform', 256)
# plot uniform
#plt.figure(figsize=(18, 12))
```

log_regression.py > ...

```
axs[0,1].set_title('Batch Size Comparison for Uniform initializer')
axs[0,1].set_xlabel('Epochs')
axs[0,1].set_ylabel('Validation accuracy')
axs[0,1].plot(x, acc_batch_uniform[0], label='Batch size = 32')
axs[0,1].plot(x, acc_batch_uniform[1], label='Batch size = 64')
axs[0,1].plot(x, acc_batch_uniform[2], label='Batch size = 128')
axs[0,1].plot(x, acc_batch_uniform[3], label='Batch size = 256')
axs[0,1].legend()

# weights initialized ZEROS with different batches
logistic_reg_trainer('zeros', 32)
logistic_reg_trainer('zeros', 64)
logistic_reg_trainer('zeros', 128)
logistic_reg_trainer('zeros', 256)
# plot zeros
#plt.figure(figsize=(18, 12))
axs[1, 0].set_title('Batch Size Comparison for Zeros initializer')
axs[1, 0].set_xlabel('Epochs')
axs[1, 0].set_ylabel('Validation accuracy')
axs[1, 0].plot(x, acc_batch_zeros[0], label='Batch size = 32')
axs[1, 0].plot(x, acc_batch_zeros[1], label='Batch size = 64')
axs[1, 0].plot(x, acc_batch_zeros[2], label='Batch size = 128')
axs[1, 0].plot(x, acc_batch_zeros[3], label='Batch size = 256')
axs[1, 0].legend()

plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()

for i in range(1, 5):
    print('Final Validation accuracy for Normal Initialization With Batch Size: %d is : %f' % (32 * i, acc_batch_normal[i - 1][-1]))
for i in range(1, 5):
    print('Final Validation accuracy for Uniform Initialization With Batch Size: %d is : %f' % (32 * i, acc_batch_uniform[i - 1][-1]))
for i in range(1, 5):
    print('Final Validation accuracy for Zeros Initialization With Batch Size: %d is : %f' % (32 * i, acc_batch_zeros[i - 1][-1]))

model = LogisticRegression("normal")

X_train_new = np.concatenate((X_train, X_valid), axis=0)
y_train_new = np.concatenate((y_train, y_valid), axis=0)
final_test_acc = model.fit(X_train_new, y_train_new,
                           X_test, y_test,
```

log_regression.py > ...

```
                           epochs=20,
                           batch_size=64,
                           learning_rate=1e-3)

# Final Test Accuracy For The Best Logistic Regression Model
x = list(range(20 + 1))
plt.figure(figsize=(18, 12))
plt.xlabel('Epochs')
plt.ylabel('Test Accuracy')
plt.xticks([x for x in range(0, 22, 2)])
plt.plot(x, final_test_acc, label='Multinomial Model')
plt.legend()
plt.title('Test Accuracy for the Logistic Regression Model')
plt.show()

y_preds = model.predict(X_test)
confusion = pd.crosstab(y_test, y_preds)

# Report the final test accuracy
print('Final Test Accuracy: %f' % (final_test_acc[-1]))

# print f1 score = 2 * precision * recall / (precision +  recall)
precision = confusion[1][1] / (confusion[0][1] + confusion[1][1])
recall = confusion[1][1] / (confusion[1][1] + confusion[1][0])
f1_score = (2 * precision * recall) / (precision + recall)
print("F1 Score For Final Logistic Regression Model: ", f1_score)

# plot confusion matrix
fig, ax = plt.subplots()
ax.matshow(confusion,cmap='OrRd')
ax.set(xlabel='Test', ylabel='Prediction')

for i in range(2):
  for j in range(2):
    c = confusion[j][i]
    ax.text(i, j, str(c), va='center', ha='center')
plt.title('Confusion Matrix For Final Logistic Regression Model')
plt.show()
```

metrics.py

metrics.py > ...

```python
import numpy as np

# mean squared error
def mse(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    y_true = np.asarray(y_true)
    y_pred = np.asarray(y_pred)
    y_true = y_true.reshape((len(y_true), 1))
    return np.sum(np.square(y_true - y_pred)) / len(y_true)

# mean absolute error
def mae(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    y_true = np.asarray(y_true)
    y_pred = np.asarray(y_pred)
    y_true = y_true.reshape((len(y_true), 1))
    return np.sum(np.abs(y_true - y_pred)) / len(y_true)


def r2(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    r = np.corrcoef(y_true.squeeze(), y_pred.squeeze())[0, 1]
    return r ** 2
```

nn.py

nn.py > calc_accuracy

```python
import numpy as np
from activation_functions import ActivationFunction
from typing import Any, Type, Tuple, Iterable
from collections import defaultdict, namedtuple
from tqdm import tqdm
from metrics import mse, mae, r2

def relu(z):
    return np.maximum(0, z)

def relu_backward(z):
    return np.where(z > 0, 1, 0)

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_backward(z):
    return (1 - sigmoid(z)) * sigmoid(z)

def calc_accuracy(y_true, y_pred):
    y_true = np.asarray(y_true).squeeze()
    y_pred = np.asarray(y_pred).squeeze()
    return np.mean(y_true == y_pred)

FullyConnectedLayerWeights = namedtuple('FullyConnectedLayerWeights', ['b', 'W'])
FullyConnectedLayerGradients = FullyConnectedLayerWeights

DEFAULT_METRICS = {
    'MSE': mse,
    'MAE': mae,
    'R2': r2,
}

class NeuralNetwork:
    def __init__(self, n_neurons):
        self.input_units = None
        # number of neurons in the layer
        self.n_neurons = n_neurons
        self.perceptron = None

    def init_network(self):
```

nn.py > calc_accuracy

```python
    def init_network(self):
        rng = np.random.default_rng()

        # init weights
        self.perceptron = []
        for i, n in enumerate(self.n_neurons):
            n_prev = self.n_neurons[i - 1] if i != 0 else self.input_units
            bound = np.sqrt(6 / (n + n_prev))
            b = rng.uniform(-bound, bound, size=(n, 1))
            W = rng.uniform(-bound, bound, size=(n, n_prev))

            layer = FullyConnectedLayerWeights(b, W)
            self.perceptron.append(layer)

    def __call__(self, X):
        Z = X.T
        for layer in self.perceptron[:-1]:
            V = layer.W @ Z + layer.b
            Z = relu(V)
        V = self.perceptron[-1].W @ Z + self.perceptron[-1].b
        Z = sigmoid(V)
        return Z.T

    # predict the output with respect to some threshold
    def predict(self, X: np.ndarray, threshold: float = 0.5) -> np.ndarray:
        return np.asarray(self(X) > threshold, dtype=np.int32)

    # forward pass through the layers by caching the outputs
    def forward(self, X, y):
        Z = X
        V_cache, Z_cache = [], []
        for layer in self.perceptron[:-1]:
            V = layer.W @ Z + layer.b
            Z = relu(V)

            V_cache.append(V)
            Z_cache.append(Z)

        V = self.perceptron[-1].W @ Z + self.perceptron[-1].b
        Z = sigmoid(V)

        V_cache.append(V)
```

nn.py > calc_accuracy

```python
        Z_cache.append(Z)

        # compute metrics
        J = {metric: fn(y.T, Z.T) for metric, fn in DEFAULT_METRICS.items()}
        return J, V_cache, Z_cache

    def backward(self, X, y, V_cache, Z_cache):

        # gradients for the model
        gradients = []

        delta = (1 / Z_cache[-1].shape[1]) * np.subtract(Z_cache[-1], y)
        db = np.mean(delta, axis=1, keepdims=True)
        dW = delta @ Z_cache[-2].T / Z_cache[-2].shape[-1]
        gradients.append(FullyConnectedLayerGradients(db, dW))

        # calculate gradients for the hidden layer
        for i in reversed(range(1, len(self.perceptron) - 1)):
            delta = (self.perceptron[i + 1].W.T @ delta) * relu_backward(V_cache[i])
            db = np.mean(delta, axis=1, keepdims=True)
            dW = delta @ Z_cache[i - 1].T / Z_cache[i - 1].shape[-1]
            gradients.append(FullyConnectedLayerGradients(db, dW))

        # treat the first hidden layer differently by using X (design matrix)
        delta = (self.perceptron[1].W.T @ delta) * relu_backward(V_cache[0])
        db = np.mean(delta, axis=1, keepdims=True)
        dW = (delta @ X.T) / X.shape[-1]
        gradients.append(FullyConnectedLayerGradients(db, dW))

        # get the correct order
        gradients.reverse()
        return gradients

    # forward and backward pass combined
    def step(self, X, y):
        J, V_cache, Z_cache = self.forward(X, y)
        gradients = self.backward(X, y, V_cache, Z_cache)
        return J, gradients

    def fit(self,
            X, # training features
```

nn.py > calc_accuracy

```
            X, # training features
            y, # training labels
            X_valid, # validation features
            y_valid, # validation labels
            alpha=0.1, # learning rate
            momentum=0.85, # momentum
            epochs=50, # number of training iterations
            batch_size=32, # batch size for training
            patience=5, # patience
            min_delta=1e-7, # patience criteria
            shuffle=True,
            cold_start=False
            ):

        X = np.asarray(X)
        y = np.asarray(y)
        X_valid = np.asarray(X_valid)
        y_valid = np.asarray(y_valid)

        if cold_start or self.perceptron is None:
            self.input_units = X.shape[-1]
            self.init_network()

        n_batches = len(X) // batch_size
        history = defaultdict(list)

        delta_weights_prev = None
        n_no_improvement = 0
        for epoch in (progress_bar := tqdm(range(epochs))):
            # get the shuffled data
            train_indices = np.random.permutation(len(X)) if shuffle else np.arange(len(X))
            batch_avg_losses = []
            for batch in range(n_batches):
                # get the batch data from the shuffle
                batch_indices = train_indices[batch * batch_size: (batch + 1) * batch_size]
                X_batch = X[batch_indices].T
                y_batch = y[batch_indices].T

                # take one step: forward and backward combined
                J, gradients = self.step(X_batch, y_batch)

                batch_avg_losses.append(J)
```

nn.py > calc_accuracy

```
                # update weights
                delta_weights = self.add_velocity(delta_weights_prev, gradients, momentum)
                self.update_params(delta_weights, alpha)

                # updates saved for momentum
                delta_weights_prev = delta_weights

            # evaluate the model on validation dataset
            valid_avg_losses = {metric: np.mean(fn(y_valid, self(X_valid)))
                                for metric, fn in DEFAULT_METRICS.items()}
            train_avg_losses = {metric: np.mean([loss[metric] for loss in batch_avg_losses])
                                for metric in DEFAULT_METRICS.keys()}

            valid_acc_log = calc_accuracy(y_valid, self.predict(X_valid))
            train_acc_log = calc_accuracy(y, self.predict(X))

            history['train_acc'].append(train_acc_log)
            history['valid_acc'].append(valid_acc_log)

            for metric in DEFAULT_METRICS.keys():
                history[f'train_{metric}'].append(train_avg_losses[metric])
                history[f'valid_{metric}'].append(valid_avg_losses[metric])

            # for output display
            progress_bar.set_description_str(f'n_neurons={"-".join([str(i) for i in self.n_neurons])}, '
                                             f'alpha={alpha}, momentum={momentum}, batch_size={batch_size}')
            progress_bar.set_postfix_str(f'train_acc={train_acc_log:.7f}, '
                                         f'valid_acc={valid_acc_log:.7f}')

            # exit training if there is no significant improvement in the model
            if epoch > 2 and history['valid_MSE'][-2] - history['valid_MSE'][-1] < min_delta:
                n_no_improvement += 1
                if n_no_improvement > patience:
                    break
            else:
                n_no_improvement = 0
        return history

    # calculate the updates by introducing momentum
    @staticmethod
```

nn.py > calc_accuracy

```
    @staticmethod
    def add_velocity(delta_weights_prev, gradients, momentum):
        delta_weights = [
            FullyConnectedLayerWeights(*[momentum * delta_w_prev + (1 - momentum) * grads
                                         for delta_w_prev, grads in zip(delta_weights_prev_, gradients_)])
            for delta_weights_prev_, gradients_ in zip(delta_weights_prev, gradients)
        ] if delta_weights_prev else gradients
        return delta_weights

    # apply updates on to the weights
    def update_params(self, delta_weights, alpha):
        self.perceptron = [
            FullyConnectedLayerWeights(*[w - alpha * delta for w, delta in zip(layer, delta_weights_)])
            for layer, delta_weights_ in zip(self.perceptron, delta_weights)
        ]
```

nn_test.py

nn_test.py > ...

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from nn import NeuralNetwork
from itertools import product
from nn import calc_accuracy
from os import path

df = pd.read_csv("data/url_processed.csv")
np.random.seed(2023)

# shuffle the data frame
df = df.sample(frac = 1)

# our feature matrix
X = df[['use_of_ip', 'url_length', 'subdomain_length', 'tld_length', 'fld_length', 'path_length',
        'count_letters', 'count_digits', 'count_puncs', 'count.', 'count@', 'count-',
        'count%', 'count?', 'count=', 'count+', 'count/', 'count,', 'count!',
        'letters_ratio', 'digit_ratio', 'punc_ratio', 'count_dirs',
        'use_of_shortener', 'first_dir_length',
        'url_length_q', 'fld_length_q', 'https', 'count-https', 'count-http', 'count-www', 'is_susp', 'is_php']]

print(X.head())

# labels
y = df['is_malicious']

print(X.isna().sum())

# fill the nan values with 0
X = X.fillna(0)

# check again the number of nan values
print(X.isna().sum())

# divide the data into train, test and validation datasets
length = len(y)

# %70 train, %10 validation, %20 test
train_set_size = int(length * 0.7)
valid_set_size = int(length * 0.1)
```

nn_test.py > ...

```python
valid_set_size = int(length * 0.1)
test_set_size = int(length * 0.2)

X_train = X.iloc[:train_set_size]
y_train = y.iloc[:train_set_size]

X_valid = X.iloc[train_set_size: train_set_size + valid_set_size]
y_valid = y.iloc[train_set_size: train_set_size + valid_set_size]

X_test = X.iloc[train_set_size + valid_set_size:]
y_test = y.iloc[train_set_size + valid_set_size:]

nn_layers_list = [
    [64, 1],
    [32, 32, 1],
    [64, 64, 1],
]
nn_alphas = [1e-2, 5e-3, 1e-3, 5e-4]
nn_momentums = [0.85, 0.95]
nn_batch_sizes = [32, 64]

nn_hyperparams = list(product(nn_alphas, nn_momentums, nn_batch_sizes))

def plot_from_history(history, title):
    h = pd.DataFrame.from_dict(history)[['train_acc', 'valid_acc']]
    h = h.set_axis(['Train Acc', 'Test Acc'], axis=1)
    h.plot()
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title(title)
    plt.show()

def subplot_mse(hist_list, nn_hyperparams, layers):

    fig, axs = plt.subplots(4, 4, figsize=(18, 18))
    for i in range(4):
        for j in range(4):
            axs[i,j].set_title('lr: ' + str(nn_hyperparams[4*i+j][0]) + ', mom: ' + str(nn_hyperparams[4*i+j][1]) + ', bs: ' + str(nn_hyperparams[4*i+j][2]), fontdict={
            axs[i,j].set_xlabel('Epochs', fontdict={'fontsize': 8})
            axs[i,j].set_ylabel('Accuracy', fontdict={'fontsize': 8})
            axs[i,j].plot(hist_list[4 * i + j]['train_acc'], label='Train Acc')
```

nn_test.py > ...

```
                axs[i,j].plot(hist_list[4 * i + j]['train_acc'], label='Train Acc')
                axs[i,j].plot(hist_list[4 * i + j]['valid_acc'], label='Validation Acc')
                axs[i,j].legend(fontsize=5)
        plt.suptitle('Hyperparameter Tuning For NN With layers: ' + str(layers))
        plt.subplots_adjust(wspace=0.5, hspace=0.7)
        plt.savefig(path.join('plots', f'{layers}-validation-plot.jpg'))
        plt.show()

    accList = []
    layer_hist_list = []
    for layers in nn_layers_list:
        hist_list = []
        for alpha, momentum, batch_size in nn_hyperparams:
            nn = NeuralNetwork(n_neurons=layers)
            history = nn.fit(X_train, y_train, X_valid, y_valid, alpha=alpha, batch_size=batch_size, momentum=momentum, epochs=200, patience=5)
            hist_list.append(history)
            accList.append(history['valid_acc'][-1]) # push the last accuracy obtained
        layer_hist_list.append(hist_list)

    for i, layers in enumerate(nn_layers_list):
        subplot_mse(layer_hist_list[i], nn_hyperparams, layers)

    # Get the best parameters by using the validation accuracy
    # as the metric
    accList = np.asarray(accList)
    nn_best_index = np.argmax(accList)
    nn_best_layer_index = nn_best_index // len(nn_hyperparams)
    nn_best_layer = nn_layers_list[nn_best_layer_index]
    nn_best_params = nn_hyperparams[nn_best_index % len(nn_hyperparams)]
    nn_best_alpha = nn_best_params[0]
    nn_best_momentum = nn_best_params[1]
    nn_best_batchsize = nn_best_params[2]

    # Plot the distribution of the accuracies
    plt.figure()
    plt.title('The number of models with a given validation accuracy')
    plt.xlabel('Accuracy of the model')
    plt.ylabel('Number of models')
    plt.hist(accList * 100)
    plt.show()

    print(f'\nBest Neural Network Settings: Size={nn_best_layer}, Learning Rate={nn_best_alpha}, Batch Size:{nn_best_batchsize}, Momentum:{nn_best_momentum}')
```

nn_test.py > ...

```
    print(f'\nBest Neural Network Settings: Size={nn_best_layer}, Learning Rate={nn_best_alpha}, Batch Size:{nn_best_batchsize}, Momentum:{nn_best_momentum}')

    # train the best model on train + validation dataset, report metrics
    # on the test dataset
    X_train = X.iloc[:train_set_size + valid_set_size]
    y_train = y.iloc[:train_set_size + valid_set_size]
    best_model = NeuralNetwork(n_neurons=nn_best_layer)
    history = best_model.fit(X_train, y_train, X_test, y_test, alpha=nn_best_alpha, batch_size=nn_best_batchsize, momentum=nn_best_momentum,epochs=500, patience=5)

    # plot the test and train accuries of the best model
    plot_from_history(history, 'Accuracy vs Epoch For Best Model')

    y_test = y_test.to_numpy()
    y_preds = best_model.predict(X_test)
    confusion = pd.crosstab(y_test.reshape((-1,)), y_preds.reshape((-1, )))

    # Report the final test accuracy
    print('Final Test Accuracy: %f' % (calc_accuracy(y_test.reshape((-1,)), y_preds.reshape((-1, )))))

    # print f1 score = 2 * precision * recall / (precision +  recall)
    precision = confusion[1][1] / (confusion[0][1] + confusion[1][1])
    recall = confusion[1][1] / (confusion[1][1] + confusion[1][0])
    f1_score = (2 * precision * recall) / (precision + recall)
    print("F1 Score For Final Neural Network Model: ", f1_score)

    # plot confusion matrix
    fig, ax = plt.subplots()
    ax.matshow(confusion,cmap='OrRd')
    ax.set(xlabel='Test', ylabel='Prediction')

    for i in range(2):
      for j in range(2):
        c = confusion[j][i]
        ax.text(i, j, str(c), va='center', ha='center')
    plt.title('Confusion Matrix For Final Neural Network Model')
    plt.show()
```