Phishing Detection Final Project

By:

Kadir Altunel, Asha Saxena, Bryam Piedra

Contributions:

Data Analysis: All members contributed

Applying Machine Learning Algorithm: All member contributed

Written Report: Kadir Altunel and Bryam Piedra worked on the report

Video Presentation: Asha Saxena worked on the video presentation

BACKGROUND

Phishing is considered to be a form of a cybersecurity attack, where emails or any other type of electrical communication is sent by an attacker, who impersonates to be someone else. This type of attack usually tries to obtain sensitive information, such as usernames, passwords, credit card numbers, or other personal data. There are several clues that might give away a phishing attack, such as misspelled URLs, the use of public email addresses instead of well-known ones, asking for personal information, etc. However, the methods used in this type of cybersecurity attacks have evolved over time and are becoming very sophisticated, to the point where they can be very hard to distinguish.

There are several types of phishing attacks, each one has its own characteristics. Some of the are:

- **Spear phishing attacks**: These attacks usually employ gathered information specific to the victim to more successfully represent the message as being authentic.
- **Whaling attacks**: These are a type of spear phishing attacks that specifically target senior executives within an organization with the objective of stealing large sums of sensitive data.
- **Pharming:** These are a type of phishing attacks that uses domain name system cache poisoning to redirect users from a legitimate website to a fraudulent one.
- **Voice phishing:** This is a form of phishing that occurs over voice-based media, including voice over IP (also called vishing). This type of scam uses speech synthesis software to leave voicemails notifying the victim of suspicious activity in a bank account or credit account.
- **SMS phishing**: This is a mobile device-oriented phishing attack that uses text messaging to convince victims to disclose account credentials or install malware.

Being a victim of these type of phishing attacks can cause many consequences with some of them being as severe as identity theft and financial loss. Therefore, individuals should remain vigilant when dealing with such cybersecurity threats and educate themselves on ways to detect phishing attempts if they ever occur.

In the ongoing fight against the cybersecurity attack presented in this paper, there are datasets that can be leveraged to develop techniques for detecting and analyzing different types of phishing attacks. In this paper, we analyze such dataset which contains a comprehensive collection of features extracted from URLs. This dataset was released by Michelle Velice Patricia. For this dataset, XGBoost, Random Forest, and Neural network was used to perform classification. Also, Multiple Imputation by Chain Equations (MICE) and IsolationForest was used which will be discussed in the *Methods* section of the paper. Finally, a discussion of the results from the methodology and conclusions will be added in the *Results* section of the paper.

DATASET

The Dataset analyzed in this paper contains a comprehensive collection of features extracted from URLs, which provide valuable insights into attributes commonly associated with phishing activities. The dataset contains 112 features, with some of them being quantities of symbols and characters within URLs to domain and directory properties, as well as parameters and server-related information. Also, it includes indicators such as email presence, time responses, domain registration details, and SSL certificate status. The data types for this dataset were: integers and floats.

The figure shown above, contains a few attributes that were used in the analysis of this dataset. These attributes are:

qty_dot_url: Quantity of dots in the URL.

qty_hyphen_url: Quantity of hyphens in the URL.

qty_underline_url: Quantity of underscores in the URL.

qty_slash_url: Quantity of slashes in the URL.

qty_questionmark_url: Quantity of question marks in the URL.

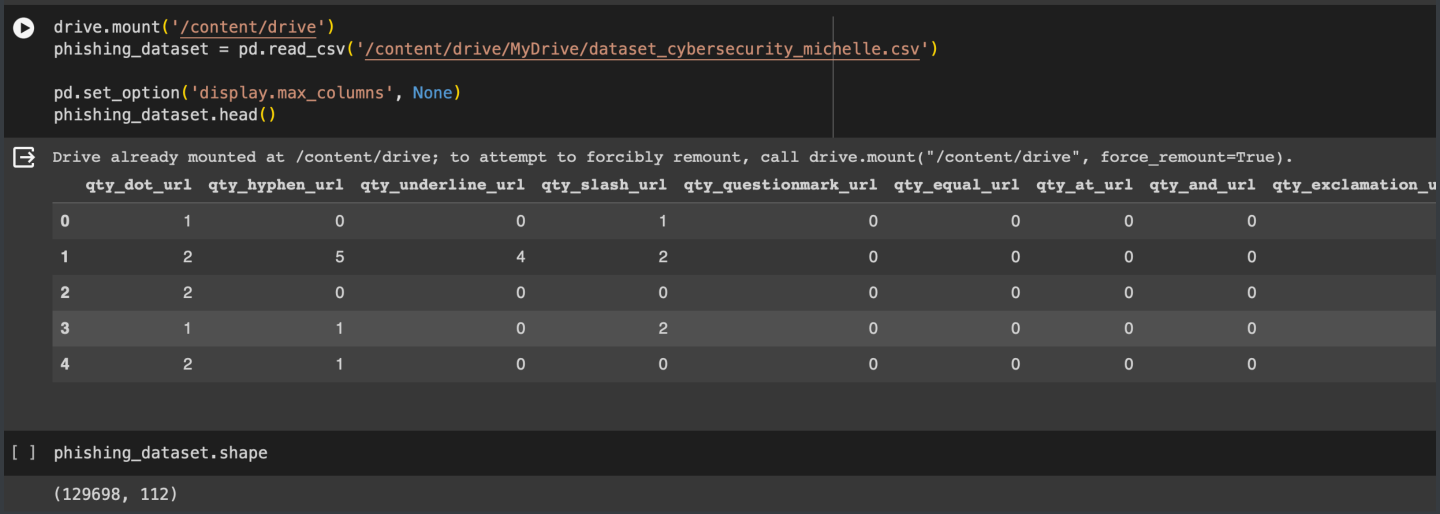
qty_equal_url: Quantity of equal signs in the URL.

qty_at_url: Quantity of "@" symbols in the URL.

qty_and_url: Quantity of "&" symbols in the URL.

METHOD IMPLEMENTATION AND RESULTS

To start the analysis of the phishing dataset, we begin by implementing data exploration. This foundational process is especially vital when dealing with datasets such as phishing data. Data exploration enables analysts to develop a comprehensive understanding of the dataset, laying the groundwork for extracting valuable insights. Through thorough investigation and an open-minded approach, data exploration empowers analysts to navigate the dataset effectively, thereby enhancing the overall analytical process. Its emphasis on in-depth inquiry is essential for informing decision-making and driving advancements in cybersecurity research, particularly in the field of phishing prevention.



```
drive.mount('/content/drive')
phishing_dataset = pd.read_csv('/content/drive/MyDrive/dataset_cybersecurity_michelle.csv')

pd.set_option('display.max_columns', None)
phishing_dataset.head()
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

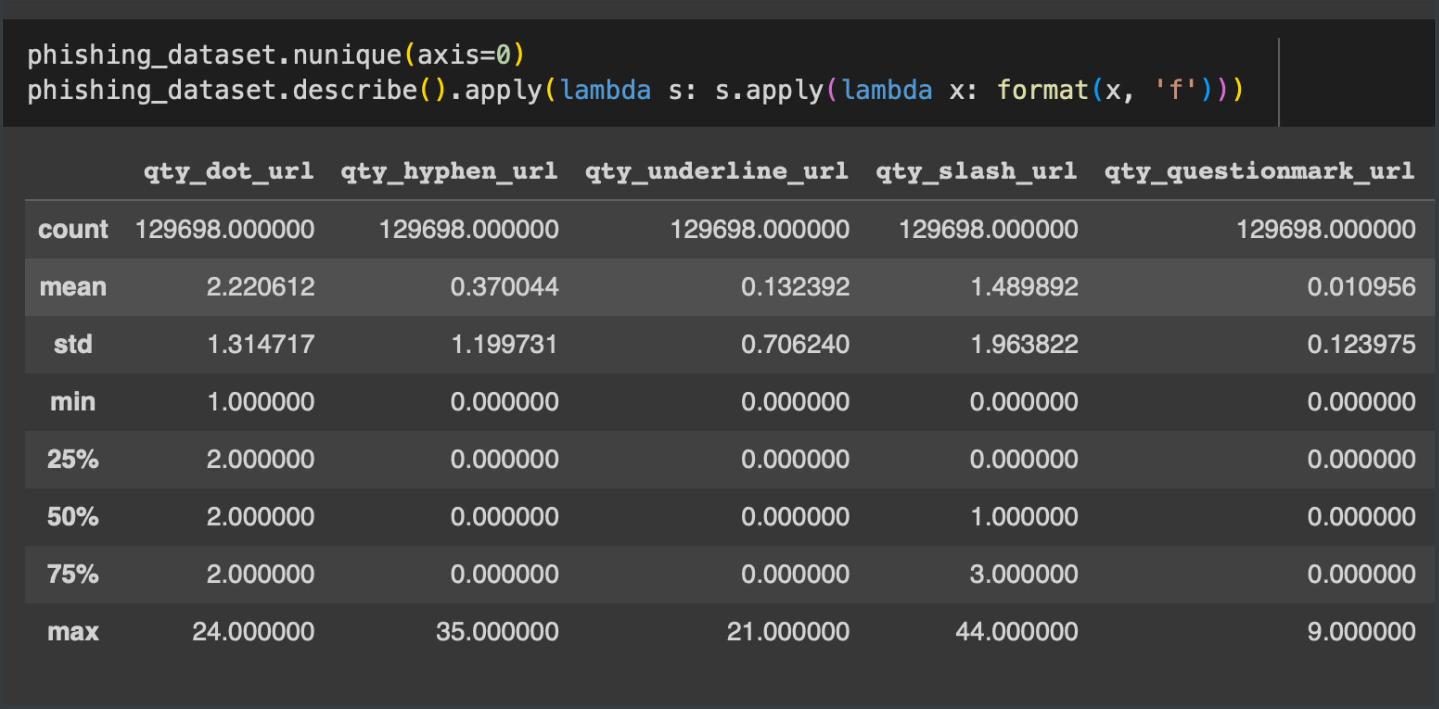
	qty_dot_url	qty_hyphen_url	qty_underline_url	qty_slash_url	qty_questionmark_url	qty_equal_url	qty_at_url	qty_and_url	qty_exclamation_u
0	1	0	0	1	0	0	0	0	
1	2	5	4	2	0	0	0	0	
2	2	0	0	0	0	0	0	0	
3	1	1	0	2	0	0	0	0	
4	2	1	0	0	0	0	0	0	

```
[ ] phishing_dataset.shape

(129698, 112)
```

For the phishing detection project, we integrate a comprehensive cybersecurity dataset titled "dataset_cybersecurity_michelle.csv" into Python working environment. Leveraging the Google Drive API, we mount Google Drive to gain access to this extensive collection of data. Utilizing the pandas library, renowned for its robust data manipulation capabilities, we import the dataset into a DataFrame, which enables us to work with the data in a structured tabular format.

We adjust the display settings of pandas to ensure full visibility of the dataset's numerous attributes. This preliminary glimpse into the data is achieved through the invocation of the `.head()` function, which displays the first five entries. These entries reveal a variety of features, such as counts of dots, hyphens, underlines, and slashes within URLs—potentially indicative markers in the identification of phishing attempts.

The scale of the dataset is substantial, with its dimensions being revealed as 129,698 rows and 112 columns. This indicates a rich, multidimensional dataset that is poised to be an invaluable asset for the subsequent data analysis and machine learning applications, focusing on the nuances of cybersecurity threats.



```
phishing_dataset.nunique(axis=0)
phishing_dataset.describe().apply(lambda s: s.apply(lambda x: format(x, 'f')))
```

	qty_dot_url	qty_hyphen_url	qty_underline_url	qty_slash_url	qty_questionmark_url
count	129698.000000	129698.000000	129698.000000	129698.000000	129698.000000
mean	2.220612	0.370044	0.132392	1.489892	0.010956
std	1.314717	1.199731	0.706240	1.963822	0.123975
min	1.000000	0.000000	0.000000	0.000000	0.000000
25%	2.000000	0.000000	0.000000	0.000000	0.000000
50%	2.000000	0.000000	0.000000	1.000000	0.000000
75%	2.000000	0.000000	0.000000	3.000000	0.000000
max	24.000000	35.000000	21.000000	44.000000	9.000000

Next, we perform a statistical examination of the dataset to derive insights into the characteristics of URLs that may be indicative of phishing activity. To achieve this, we calculate the count of unique values across the dataset using `nunique(axis=0)` to understand the diversity in the URL features. Subsequently, we use the `describe()` function to obtain a comprehensive summary of the data, which includes count, mean, standard deviation, minimum and maximum values, as well as the 25th, 50th (median), and 75th percentiles for each feature related to URL characteristics.

For instance, the 'qty_dot_url' feature, which represents the count of dots in a URL, shows an average (mean) of approximately 2.22 with a standard deviation of 1.31, indicating variability in how many dots URLs typically contain. The maximum number reported for this feature is 24, which can be an outlier or indicative of a complex or potentially malicious URL.

This statistical summary offers critical insights into the dataset and assists in identifying patterns or anomalies that could be further investigated for phishing detection. It sets the groundwork for the application of machine learning algorithms to classify URLs as phishing attempts or benign by quantitatively assessing URL patterns.

```
find_missing = (phishing_dataset==-1).sum()

with pd.option_context('display.max_rows', None, 'display.max_columns', None):
  print(find_missing.sort_values(ascending=False))
```

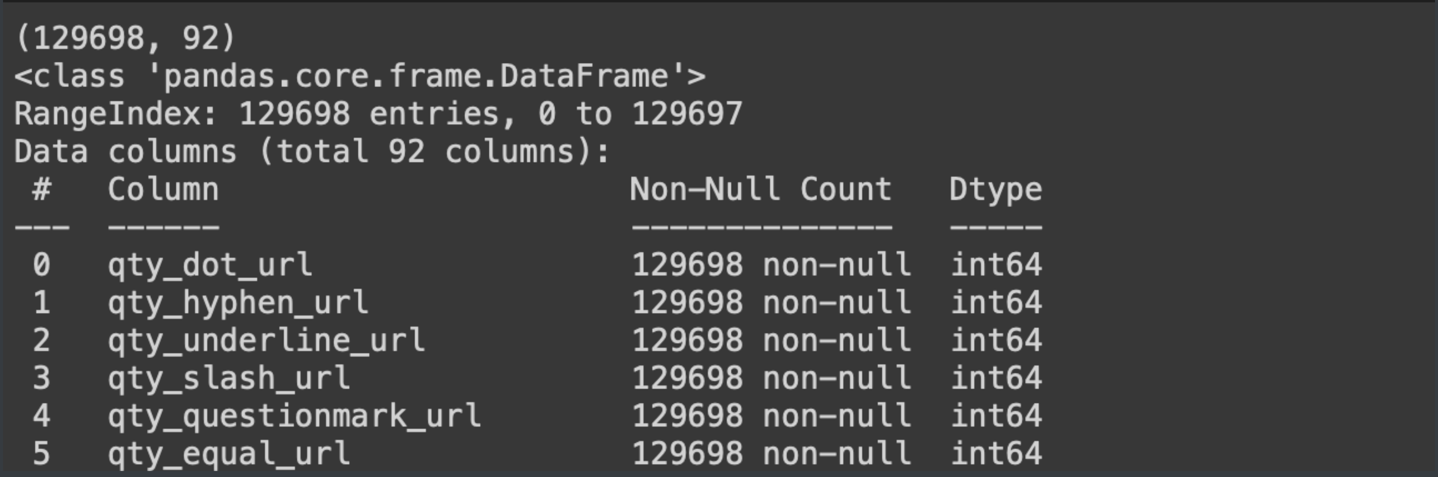
```
qty_plus_params               117020
qty_asterisk_params           117020
qty_hashtag_params            117020
qty_dollar_params             117020
qty_percent_params            117020
params_length                 117020
tld_present_params            117020
qty_params                    117020
qty_questionmark_params       117020
qty_equal_params              117020
qty_at_params                 117020
qty_and_params                117020
qty_exclamation_params        117020
qty_space_params              117020
qty_tilde_params              117020
qty_comma_params              117020
```

Next, we detail a meticulous data cleaning process conducted on the cybersecurity dataset. This process commences with the identification of missing values across different features in the dataset. To accurately pinpoint these missing values, which are denoted by the placeholder '-1' within the dataset, we calculate the sum of occurrences of this placeholder across all features using the command `(phishing_dataset==-1).sum()` .

Subsequently, we deploy the pandas library's context manager to override the default display settings, allowing me to view the full breadth of the dataset without any truncation. By invoking `sort_values(ascending=False)` on the series of missing values and printing the results, we obtain a descending order list of features by the count of missing values, providing a clear picture of which features required attention due to the prevalence of missing data.

The output reveals that several parameters, including 'qty_plus_params', 'qty_asterisk_params', 'qty_hashtag_params', and others, each had 117,020 instances marked as missing. This information is critical as it influences the subsequent steps in data preprocessing, where such missing values must be addressed through imputation or elimination to refine the dataset for the predictive modeling phase of the project on phishing URL detection.

```
columns_to_drop = find_missing[find_missing > 59703].index

phishing_dataset_cleaned = phishing_dataset.drop(columns=columns_to_drop)

print(phishing_dataset_cleaned.shape)

print(phishing_dataset_cleaned.info(verbose=True, show_counts=True))
```



```
(129698, 92)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 129698 entries, 0 to 129697
Data columns (total 92 columns):
 #   Column                        Non-Null Count   Dtype
---  ------                        --------------   -----
 0   qty_dot_url                   129698 non-null  int64
 1   qty_hyphen_url                129698 non-null  int64
 2   qty_underline_url             129698 non-null  int64
 3   qty_slash_url                 129698 non-null  int64
 4   qty_questionmark_url          129698 non-null  int64
 5   qty_equal_url                 129698 non-null  int64
```

Following the identification of missing values within the dataset, we proceed with data cleaning by discarding features that exhibited a significant number of missing entries. To maintain the integrity of the analysis, we determine that any feature with more than half of its values missing —specifically, more than 59,703 missing entries out of the total of 129,698—should be considered too sparse for reliable analysis and hence removed from the dataset.

The selection of columns to be dropped is executed using a filtering condition on the previously calculated series of missing values, identifying those columns which surpassed the threshold for missing data. The identified columns are then eliminated from the dataset, resulting in a reduced set of features that we deem more suitable for building robust predictive models.

After this pruning process, the dataset is pared down to 92 features from the original 112, as evidenced by the output of `phishing_dataset_cleaned.shape`.

```
[ ]  mice_kernel = ImputationKernel(data = phishing_dataset_cleaned,
                                    save_all_iterations = True,
                                    random_state = 2023)

     mice_kernel.mice(10, verbose=True)
     phishing_dataset_imputed = mice_kernel.complete_data()
     print(phishing_dataset_imputed.head())
```

```
[ ]  check_missing = phishing_dataset_imputed.isnull().sum()

     with pd.option_context('display.max_rows', None, 'display.max_columns', None):
       print(check_missing.sort_values(ascending=False))
```

```
phishing                    0
qty_dot_url                 0
qty_hyphen_url              0
qty_underline_url           0
email_in_url                0
time_response               0
domain_spf                  0
asn_ip                      0
time_domain_activation      0
```

In the data preprocessing stage of the project, we address missing values in the cleaned dataset by employing a statistical imputation technique known as Multiple Imputation by Chained Equations (MICE) which can be accessed in detail in the Appendix: Methods section of the report. We instantiate an ImputationKernel from the respective Python library, configuring it with the cleaned dataset, setting `save_all_iterations` to True for thorough analysis, and defining a `random_state` for reproducibility.

Then, we initiate the MICE process, requesting ten iterations to refine the imputation with a verbose output to monitor the progress. Upon completion of this process, we call the `complete_data()` method on the kernel object to retrieve the imputed dataset.

To ensure the integrity of the imputation, we perform a verification step, checking for null values using the `isnull().sum()` method. The output confirms that the dataset no longer contains any missing values across all features, which is validated by sorting and displaying the sum of null values for each feature, resulting in zero for all.

This rigorous imputation process ensures that our dataset is robust, with all missing values judiciously estimated, thereby enhancing the reliability of the subsequent machine learning models that we apply to the data in our project.

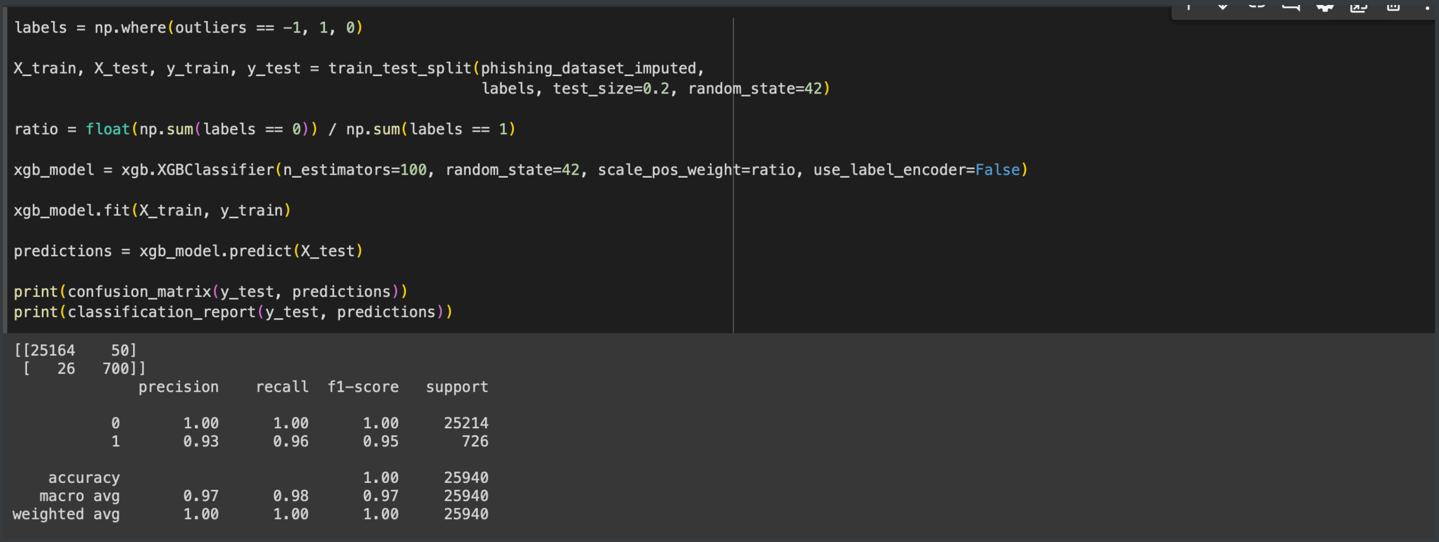
```
iso_forest = IsolationForest(n_estimators=500, contamination='auto', random_state=42)


iso_forest.fit(phishing_dataset_imputed)


outliers = iso_forest.predict(phishing_dataset_imputed)

print(outliers)

outlier_count = (outliers == -1).sum()
inlier_count = (outliers == 1).sum()

outlier_count, inlier_count
```

```
[ 1 -1  1 ...  1  1  1]
(3655, 126043)
```

After deploying MICE for our phishing dataset, we continue the project by implementing an anomaly detection algorithm using the Isolation Forest method. We configure an Isolation Forest model with 500 estimators, which refers to the number of base estimators in the ensemble, and set the contamination parameter to 'auto' to allow the algorithm to automatically determine the threshold for outlier detection. The random_state parameter is fixed at 42 to ensure reproducibility of results.

We fit the model to our imputed phishing dataset, which has already undergone preprocessing to deal with missing values. Once the model is trained, we use it to predict the outliers within our dataset. The prediction yields an array of labels where '-1' signifies an outlier and '1' denotes an inlier.

By calculating the sum of occurrences of '-1' and '1' in our prediction array, we are able to quantify the number of outliers and inliers detected by the model. In our case, the model identifies 3,655 outliers and 126,043 inliers, providing us with a clear perspective on the distribution of anomalies within our data. This step is crucial as it helps us isolate potential anomalies that could be indicative of phishing activities, thereby enhancing the robustness of our cybersecurity analysis.



```python
labels = np.where(outliers == -1, 1, 0)

X_train, X_test, y_train, y_test = train_test_split(phishing_dataset_imputed,
                                                    labels, test_size=0.2, random_state=42)

ratio = float(np.sum(labels == 0)) / np.sum(labels == 1)

xgb_model = xgb.XGBClassifier(n_estimators=100, random_state=42, scale_pos_weight=ratio, use_label_encoder=False)

xgb_model.fit(X_train, y_train)

predictions = xgb_model.predict(X_test)

print(confusion_matrix(y_test, predictions))
print(classification_report(y_test, predictions))
```

```
[[25164    50]
 [   26   700]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     25214
           1       0.93      0.96      0.95       726

    accuracy                           1.00     25940
   macro avg       0.97      0.98      0.97     25940
weighted avg       1.00      1.00      1.00     25940
```

In the current phase of our research, we apply the eXtreme Gradient Boosting (XGBoost) algorithm to further refine our phishing detection model. The XGBoost model is favored for its performance and speed. Labels are assigned to our dataset where outliers are marked with '1' and inliers with '0'. We then split our dataset into training and testing sets, allocating 20% of the data for testing while maintaining a random state for consistent train-test splits across experiments.

To address class imbalance, we calculate the ratio of inliers to outliers and set the `scale_pos_weight` parameter in our XGBoost classifier to this ratio. This technique helps in normalizing the influence of each class on the learning process. We configure the classifier with 100 trees and ensure reproducibility by fixing the `random_state`. The `use_label_encoder=False` parameter is set to avoid using the deprecated label encoder in XGBoost.

Post-training, we deploy the model to make predictions on our test set. The performance is quantitatively assessed using a confusion matrix and a classification report. The confusion matrix indicates a substantial number of true positives (700) and true negatives (25,164), suggesting a strong ability to correctly identify both phishing and non-phishing instances. The classification report provides detailed metrics such as precision, recall, and F1-score. A precision of 0.93 for the phishing class and an F1-score of 0.95 demonstrate the model's high accuracy and balanced performance. Moreover, the weighted averages of precision, recall, and F1-score all stand at 1.00, reflecting the model's overall accuracy and its robustness in the context of phishing detection.

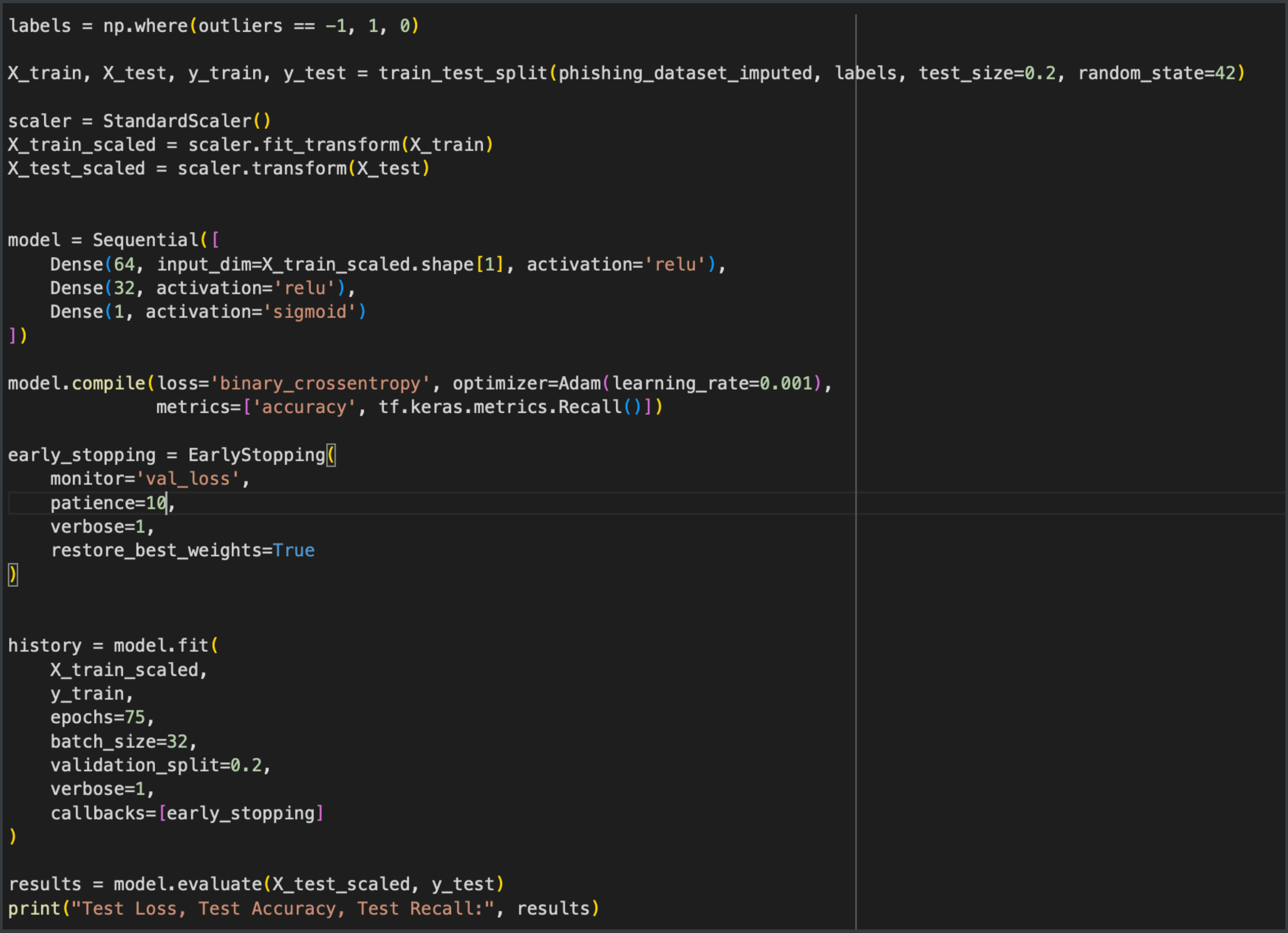
```
labels = np.where(outliers == -1, 1, 0)


X_train, X_test, y_train, y_test = train_test_split(phishing_dataset_imputed,
                                                    labels, test_size=0.2, random_state=42)

rf_model = RandomForestClassifier(n_estimators=100, random_state=42, class_weight='balanced')
rf_model.fit(X_train, y_train)


predictions = rf_model.predict(X_test)

print(confusion_matrix(y_test, predictions))
print(classification_report(y_test, predictions))
```

```
[[25198    16]
 [   79   647]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     25214
           1       0.98      0.89      0.93       726

    accuracy                           1.00     25940
   macro avg       0.99      0.95      0.96     25940
weighted avg       1.00      1.00      1.00     25940
```

Next, we continute evaluating our model by applying a Random Forest Classifier, a machine learning model known for its high accuracy and ability to operate over large datasets with a multitude of features.

Our Random Forest Classifier is configured with 100 estimators—individual decision trees that contribute to the overall decision-making process. We apply a 'balanced' class weighting to account for any discrepancies in the representation of classes within the dataset. This helps to mitigate the bias toward the majority class that would otherwise occur in an imbalanced dataset.

The confusion matrix indicates a substantial number of true positives (647) and true negatives (25,198), again, suggesting a strong ability to correctly identify both phishing and non-phishing instances. A precision of 0.98 for the phishing class and an F1-score of 0.93 shows that our Random Forest model demonstrates exceptional performance, indicating its potential efficacy in identifying and mitigating phishing threats within cybersecurity frameworks.



```
labels = np.where(outliers == -1, 1, 0)

X_train, X_test, y_train, y_test = train_test_split(phishing_dataset_imputed, labels, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)


model = Sequential([
    Dense(64, input_dim=X_train_scaled.shape[1], activation='relu'),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])

model.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.001),
              metrics=['accuracy', tf.keras.metrics.Recall()])

early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=10,
    verbose=1,
    restore_best_weights=True
)


history = model.fit(
    X_train_scaled,
    y_train,
    epochs=75,
    batch_size=32,
    validation_split=0.2,
    verbose=1,
    callbacks=[early_stopping]
)

results = model.evaluate(X_test_scaled, y_test)
print("Test Loss, Test Accuracy, Test Recall:", results)
```

For the final algorithm for our phishing detection, we use neural networks.

Prior to training, we standardize our features using the `StandardScaler` to ensure that our neural network receives data that is on a similar scale, which is essential for the effective training of neural networks.

We construct a neural network using the Keras library with the following architecture:

- An input layer with 64 neurons and the 'ReLU' activation function.
- A hidden layer with 32 neurons, also using the 'ReLU' activation function.
- An output layer with a single neuron with the 'sigmoid' activation function, suitable for binary classification.

Our model is compiled with the binary cross-entropy loss function and the Adam optimizer with a learning rate of 0.001. We track the accuracy and recall during training as performance metrics.

To prevent overfitting, we employ an early stopping mechanism with a `patience` of 10 epochs, which terminates the training process if the validation loss does not improve for 10 consecutive epochs. The best weights observed during training are restored to the model at the end of training.

The model is trained on the scaled training data for up to 75 epochs with a batch size of 32 and a validation split of 20%. The `EarlyStopping` callback is used to monitor the model's performance on the validation set.

```
Test Loss, Test Accuracy, Test Recall: [0.017022788524627686, 0.9932536482810974, 0.9173553586006165]
```

Results from our neural network indicate a high level of accuracy and a relatively high recall. The test loss being low suggests that the model's predictions are quite close to the actual labels. High recall indicates that the model is capable of identifying most of the positive instances correctly, which, in the context of phishing detection, means catching a high number of actual phishing attempts.

CONCLUSION

All three algorithms show exceptional performance in our phishing dataset.

- **Accuracy**: The Neural Network has the highest accuracy, followed closely by Random Forest, and then XGBoost.
- **Precision**: Random Forest has the highest precision, followed by XGBoost. The precision for the Neural Network is not provided, but the high overall accuracy suggests it is likely competitive.
- **Recall**: XGBoost has the highest recall, followed by the Neural Network, and then Random Forest.
- **F1-Score**: XGBoost has the highest F1-score, indicating the best balance between precision and recall. Random Forest has a slightly lower F1-score. The F1-score for the Neural Network is not provided but would likely be high based on the other metrics.

In our scenario, where phishing detection is crucial, often the most important metric is recall because the cost of missing a phishing attempt can be very high. As we see from the results XGBoost has the highest recall, making it the most suitable candidate for the phishing detection.

However, the Neural Network's balanced high accuracy and recall also make it a strong candidate, potentially offering a more nuanced balance of performance metrics, which could be more desirable in certain applications.

APPENDIX: SOURCES

Video: https://drive.google.com/file/d/1ATyfqgloONoW4qo2ZhX6FZr777x4qhWq/view?usp=sharing

Dataset: https://www.kaggle.com/datasets/michellevp/dataset-phishing-domain-detection-cybersecurity

APPENDIX: REFERENCES

https://deeplearningmath.org/drafts/chap4.pdf

https://arxiv.org/pdf/1603.02754.pdf

https://www.techtarget.com/searchsecurity/definition/phishing#:~:text=Phishing%20is%20a%20type%20of,messages%2C%20to%20reveal%20sensitive%20information

APPENDIX: METHODS

DEALING WITH MISSING DATA

Missing data is a common issue in real-world datasets for various reasons such as undefined values and errors during data collection. Training a model with datasets containing many missing values can significantly degrade the quality of a machine learning model, particularly deterministic models which are the most prevalent in practical applications today. Therefore, addressing missing data before deploying machine learning models is essential.

While there are advanced generative models capable of handling incomplete data, their complexity limits widespread application. Missing data typically falls into one of three categories:

- Missing Completely at Random (MCAR)
- Missing at Random (MAR)
- Not Missing at Random (NMAR)

These distinctions are crucial for understanding how to effectively manage and impute missing data in various analytical contexts.

MCAR

Data is considered Missing Completely at Random (MCAR) when the probability of missing data is independent of any model variables. MCAR is an assumption and is difficult to verify with observed data alone, as it is impossible to determine if missingness depends on unobserved values.

Although MCAR assumes that data with missing values do not systematically differ from complete records, this is often not the case in practice. When data are genuinely MCAR, analyses performed on complete records alone can provide unbiased estimates, but this discards valuable incomplete data, reducing the precision of our results. The aim of more advanced analyses is to incorporate these incomplete but useful observations, enhancing the robustness of our statistical conclusions.

MAR

Observations are classified as Missing at Random (MAR) when the probability of data being missing may depend on the variable's inherent value, but this dependency is no longer present when considering the observed data.

When data are MAR, relying solely on complete records for analysis will generally result in biased outcomes. For instance, the sample mean calculated from partially observed variables will likely be skewed. This occurs because, under MAR, complete records do not represent a random subset of the overall target population. Nonetheless, it is possible to achieve unbiased estimates and valid conclusions using advanced statistical techniques like multiple imputation.

It’s important to recognize that MAR is not an inherent trait of the data but rather an assumption applied during analysis. While we can assess the plausibility of this assumption, it cannot be conclusively proven.

NMAR

When data are neither Missing Completely at Random (MCAR) nor Missing at Random (MAR), they are classified as Missing Not at Random (MNAR), also known as Not Missing at Random (NMAR).

Like MAR, analyzing only complete records when data are MNAR typically leads to biased estimates, including the calculation of sample summary statistics. Handling MNAR data is more complex and usually requires incorporating external information to accurately understand and address the mechanism behind the missingness.

Based on the previously mentioned types of missing data, the missing values in the phishing dataset, identified by data points of -1, are classified as Missing at Random (MAR). This classification suggests that while there is a relationship between the missing data and other variables in the dataset, the specific missing entries are random.

In addressing Missing at Random (MAR) data, leveraging multivariate imputation is pivotal. This method fills in missing data by utilizing the correlations and interactions among the available variables in the dataset. Among the most revered techniques in this domain is Multiple Imputation by Chained Equations (MICE), celebrated for its comprehensive and adaptable approach.

Foundations of MICE

Accounting for the Missing Data Process: The model $M(Y_{obs}, Y_{mis}|X)$ incorporates understanding of the mechanisms that lead to data being missing, where Y_{obs} represents observed values, Y_{mis} represents missing values, and X represents auxiliary variables that help explain the missingness.

Preserving Relationships Within Data: It is crucial to maintain the statistical relationships $R(Y_{obs}, Y_{mis})$ that exist within the data to preserve its structure and integrity.

Capturing Uncertainty: By introducing randomness in the imputation process through multiple imputations, MICE effectively encapsulates the uncertainty in the relationships, enhancing the reliability of statistical inferences. These principles ensure that the imputation results align statistically across a range of conditions, conforming to the principles.

Challenges in Multivariate Imputation and Mitigation Strategies:

Incomplete Predictors: When the predictors Y_{-j} used in imputation models are themselves missing, MICE applies an iterative procedure where each variable Y_j is imputed using all other available variables in a chained equation.

Circular Dependence: To mitigate issues where Y_1 depends on Y_2 and vice versa, MICE updates each variable sequentially, using the most recent imputations of other variables in each step.

High Dimensionality vs. Small Sample Size: Problems like collinearity, which often arise when the number of variables p is large relative to the number of observations n might require pre-imputation strategies such as principal component analysis or regularization.

Diverse Data Types: MICE can use different predictive models $M(Y_j|Y_{-j})$ for different types of variables (e.g., logistic regression for binary data, linear regression for continuous data).

Complex Relationships and Censoring: For complex or censored relationships, more sophisticated models or even machine learning techniques might be integrated within the MICE framework.

Logical Inconsistencies: To prevent illogical imputations (e.g., pregnant fathers), constraints or conditional rules based on domain knowledge should be applied.

Unknown Analysis Models: If the future analytical model Q is undetermined, using generalized imputation approaches like predictive mean matching ensures fewer assumptions about the data distribution.

Through these methodologies, MICE facilitates robust and meaningful analyses of datasets completed via imputation, making it a valuable tool in statistical data handling.

The pseudocode for the MICE algorithm can be seen below:

```
Algorithm: Multivariate Imputation by Chained Equations
Define $Y$ as an $n \times p$ data matrix where rows represents samples and columns represent values
Data: Incomplete dataset $Y = (Y^{obs}, Y^{mis})$
Result: Incomplete dataset $Y^T = (Y^{obs}, Y^{mis,T})$
Define $Y_j$ as the $j^{th}$ of column of $Y$ where $Y_j = Y_j^{obs},\ Y_j^{mis}$
for $\leftarrow 1$ to $p$ do
```

```
    imputation model for the incomplete variable $Y_j \leftarrow P(Y_j|Y_{-j},\ \theta_j)$
    starting imputations $Y_j^{mis\ 0}$ $\leftarrow$ draws from $Y_j^{obs}$
Define $Y_{-j}^t = (Y_1^t,\ Y_2^t,\ \ldots,\ Y_{j-1}^t,\ Y_{j+1}^{t-1},\ \ldots,\ Y_{p-1}^{t-1},\ Y_p^{t-1})$ where $Y_j^t$ is the $j^{th}$ feature at iteration $t$
for $t \leftarrow 1$ to $T$ do
    for $j \leftarrow 1$ to $p$ do
        $\theta_j^t \leftarrow$ draw from posterior $P(\theta_j|Y_j^{obs},\ Y_{-j}^t)$
        $Y_j^{mis,\ t} \leftarrow$ draws from posterior predictive $P(Y_j^{mis}|Y_{-j}^t,\ \theta_J^t)$
return $Y^t$
```

DEALING WITH OUTLIERS/ANOMALIES

Anomalies are unexpected and nonconforming patterns that emerge from the fabric of observed datasets. The discovery, comprehension, and projection of such anomalies constitute a fundamental aspect of contemporary data mining practices. Effective anomaly detection serves as a conduit for distilling vital intelligence from vast data repositories. This intelligence, in turn, is instrumental for a multitude of practical uses, such as thwarting nefarious cyber activities, pinpointing and rectifying defects in intricate systems, and deepening the understanding of the dynamics within natural, societal, and technological constructs.

The challenge of anomaly detection lies in the identification of these unusual patterns within datasets. While the term 'anomaly' is widely recognized, various domains refer to these irregularities by other nomenclatures such as outliers, discordant observations, exceptions, aberrations, surprises, peculiarities, or contaminants. Nonetheless, the concepts of 'anomalies' and 'outliers' are frequently utilized synonymously. The spectrum of applications for anomaly detection is broad and impactful, encompassing areas such as financial fraud prevention in credit card transactions, insurance, and healthcare, and of course in our case, phishing detection in cyber-security. The significance of detecting anomalies is magnified by the fact that within many sectors, these atypical data points can equate to vital insights that prompt significant and oftentimes crucial actions.

The Isolation Forest stands out as a particularly compelling algorithm within the domain of anomaly detection. This algorithm operates on an unsupervised learning principle, utilizing ensemble methods akin to those in a random forest framework. Isolation Forest collectively employs multiple decision trees to determine the anomaly score of a data point by averaging individual assessments.

Distinct from conventional anomaly detection algorithms, which typically establish a baseline for normality before identifying deviations, the Isolation Forest algorithm strategically seeks to segregate anomalies at the outset. By directly targeting the isolation of outliers, the algorithm efficiently pinpoints data points that diverge from the expected pattern without the prerequisite of defining normalcy.

The Isolation Forest, or simply iForest which innovatively constructs a collection of isolation trees (iTrees) pertinent to a dataset. Anomalies are identified based on the premise that they are discernible by shorter path lengths within these iTrees. The iForest technique hinges on just two parameters: the count of trees in the ensemble and the size of the dataset sample. Empirical results suggest that the iForest exhibits rapid convergence in detection efficacy with a minimal ensemble size. Furthermore, the algorithm attains superior detection capability and efficiency with a relatively modest sub-sample size.

The fundamental distinction of iForest from existing profiling-based approaches lies in its isolation-centric methodology. In contrast to traditional model-based, distance-based, and density-based anomaly detection frameworks, iForest is characterized by:

- The ability to construct partial models through the unique isolation properties of iTrees, leveraging sub-sampling to a degree unattainable by current techniques. This facilitates the omission of large sections of an iTree which are unnecessary for isolating normal data points, hence, there is no need for their construction. Smaller sample sizes are found to be optimal in enhancing iTree quality due to the diminution of swamping and masking effects.
- The independence from distance or density calculations when identifying anomalies, thereby eschewing the substantial computational costs associated with these metrics in other methodologies.

- A linear computational time complexity coupled with a low overhead in both constant factors and memory requirements. The most competitive existing algorithms only approximate linear time complexity while incurring significant memory demands.
- Scalability, allowing for efficient adaptation to datasets of considerable magnitude and high dimensionality, inclusive of numerous irrelevant attributes.

APPLYING ALGORITHMS

The performance of machine learning models hinges heavily on the chosen algorithm. Improper selection can lead to suboptimal results, wasted computational resources, and a failure to realize the full potential of available data. Here are the key considerations on how to choose the correct algorithm on a given machine learning model:

Key Considerations

- **Problem Definition:** Supervised tasks like classification and regression require distinct algorithmic approaches compared to unsupervised learning techniques like clustering or dimensionality reduction. Clearly defining the problem is the first step towards narrowing down algorithm candidates.
- Data Characteristics:
    - **Size and Complexity:** Large, complex datasets may benefit from computationally intensive models like deep neural networks, while smaller datasets could be better suited to linear models or decision trees.
    - **Noise and Outliers:** The robustness of different algorithms to imperfections in data must be factored in.
    - **Linear vs. Non-linear Patterns:** Algorithms capable of handling non-linear relationships (e.g., SVMs with non-linear kernels, neural networks) are crucial when data patterns are complex.
- Trade-offs:
    - **Accuracy vs. Computational Cost:** Evaluate the trade-off between potential accuracy gains and the associated increases in model complexity and training time.

    - **Interpretability vs. Performance:** Assess whether understanding model reasoning is essential, potentially favoring simpler, more interpretable models, or if achieving peak performance outweighs explainability.
- Practical Considerations:
    - **Feature Engineering:** Consider the algorithm's reliance on manual feature selection/transformation vs. the ability to learn patterns from raw data.
    - **Implementation Availability:** Prioritize algorithms with robust implementations in widely used libraries (e.g., scikit-learn, TensorFlow) to streamline development.
    - **Adaptability:** For applications with dynamic data streams, choose algorithms that enable incremental learning.

XGBoost

XGBoost or eXtreme Gradient Boosting stands as a preeminent algorithm within the realm of ensemble learning methods, distinguished by its strategy of amalgamating multiple weak learners—typically decision trees—to forge a robust model. This section delineates the reasons underpinning the exceptional suitability of XGBoost for tackling classification problems:

1. **Enhanced Accuracy**: XGBoost has garnered acclaim for its superior performance across a diverse array of classification tasks. It frequently surpasses competing algorithms, establishing itself as a benchmark for state-of-the-art outcomes in classification.
2. **Handling Imbalanced Datasets**: A prevalent challenge in classification tasks is the presence of imbalanced datasets, where certain classes are significantly underrepresented. XGBoost is equipped with capabilities to address this imbalance, thereby enhancing model performance and reliability.
3. **Built-in Regularization**: The algorithm incorporates L1 and L2 regularization techniques that serve to mitigate overfitting. This regularization framework is instrumental in bolstering the model's generalization abilities, thus ensuring its applicability to new, unseen data.
4. **Speed and Scalability**: The design and implementation of XGBoost are optimized for both computational speed and scalability, rendering it a viable option for processing large-scale datasets.

5. **Parameter Flexibility**: XGBoost offers a comprehensive range of tunable parameters, affording users the flexibility to tailor the algorithm to the specific demands of their classification problems. This adaptability enhances its utility across varied scenarios and data specifics.

The core of XGBoost lies in its ensemble technique, where a series of weak learners, typically decision trees, are sequentially introduced to iteratively minimize prediction errors. The ensemble prediction y_i is formulated as:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i)$$

where f_k represents individual trees contributing to the ensemble. Initially, the model begins with a simple guess (e.g., mean of y) and progressively refines this by adding trees that model the residuals of preceding predictions.

XGBoost minimizes a loss function $L(y, \hat{y})$ such as the mean squared error for regression tasks:

$$L(y, \hat{y}) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

During each iteration, a new tree is fitted to the negative gradients of the loss function, which denote the steepest descent direction for minimizing loss which represents the gradient of the loss function with respect to the predictions at iteration $t - 1$.

$$g_i = -\frac{\partial L(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}$$

A pivotal enhancement introduced by XGBoost is the inclusion of regularization terms in the optimization objective, aimed at preventing overfitting:

$$\text{Obj} = \sum_{i=1}^{n} L(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

The regularization term $\Omega(f)$ is defined as:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda |w|^2$$

where T represents the number of leaves in the tree, w denotes leaf weights, and γ and λ are regularization coefficients. This regularization framework intricately balances model complexity and training error.

XGBoost employs a quantile sketch algorithm to efficiently determine optimal split points during tree construction, using second-order gradient information (Hessian) to prioritize splits that maximize gain, adjusted for complexity:

$$H(f) = \left[\frac{\partial^2 f}{\partial x_i \partial x_j} \right]_{i,\, j=1}^{n}$$

Additionally, it offers robust mechanisms for managing missing data, wherein it learns the most beneficial direction (left or right in a tree branch) to handle absent values for maximizing information gain.

Random Forest

Random forests represent a significant advancement over traditional bagging by constructing a large ensemble of de-correlated decision trees, whose collective outcomes are then aggregated through averaging. This method effectively enhances the diversity within the model ensemble, thereby reducing the variance without substantially increasing bias. In numerous empirical studies, the performance of random forests has been shown to parallel that of boosting algorithms, despite being conceptually simpler and less demanding in terms of training and parameter tuning.

Central to the bagging methodology is the strategy of averaging a multitude of noisy yet ostensibly unbiased models, thereby facilitating a reduction in variance. Decision trees, in particular, are exceptionally well-suited for bagging, given their proficiency in delineating the intricate interaction structures inherent in data. With sufficient depth, such trees tend to exhibit a lower bias, which is advantageous for the bagging process.

The inherent stochastic nature of decision trees, often viewed as a source of noise, is significantly mitigated when these trees are aggregated. Given that each tree in a bagging ensemble is generated from an identical distribution, the aggregated expectation over BB trees mirrors the expectation of any individual tree. Consequently, the ensemble's bias corresponds with the bias of each constituent tree, and enhancements are primarily achieved through the diminution of variance. This aspect stands in stark contrast to the boosting approach, where trees are adaptively cultivated to attenuate bias, resulting in a non-identical distribution among them.

Under the bagging framework, an ensemble composed of B identically distributed trees, each with variance σ, will exhibit a variance for the averaged model as follows:

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$$

where σ represents the positive pairwise correlation among the trees. As B escalates, the latter term diminishes, while the former, influenced by the correlation magnitude, persists, delineating the limits of variance reduction through averaging.

Random Forests enhance this principle by diminishing the correlation between individual trees without excessively escalating the variance. This is accomplished during tree development by selecting a random subset of m variables from the total p available variables prior to each bifurcation point within the tree.

In the definition of a Random Forest, after the cultivation of B such trees, denoted by $T(x; \Theta_b)$, the predictive model for the Random Forest in a regression setting is determined by:

$$\hat{f}_{rf}^{B}(x) = \frac{1}{B}\sum_{b=1}^{B} T(x; \Theta_b)$$

Here, Θ_b encapsulates the stochastic characteristics of the b-th tree within the Random Forest, encompassing the selected variables for splitting, the threshold values at each node, and the resultant values in the terminal nodes. The algorithm for the random forest can be seen below:

Algorithm: Random Forest for Regression or Classification

```
1. For $b = 1$ to $B$:
    (a) Draw a bootstrap sample $Z^*$ of size $N$ from the training data.
```

(b) Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size n_{min} is reached.

i. Select m variables at random from the p variables.
ii. Pick the best variable/split-point among the m.
iii. Split the node into two daughter nodes.

2. Output the ensemble of trees $\{T_b\}_1^B$.

To make a prediction at a new point x:

Regression: $\hat{f}_{rf}^B(x) = \frac{1}{B}\sum_{b=1}^B T_b(x)$

Classification: Let $\hat{C}_b(x)$ be the class prediction of the b-th random-forest tree. Then $\hat{C}_{rf}^B(x) = \text{majority vote}\left\{\hat{C}_b(x)\right\}_{b=1}^B$

Neural Network

An Artificial Neural Network (ANN), or Neural Network (NN), constitutes a complex network of interlinked nodes, each performing computational tasks to identify patterns within datasets. The structure of NNs is inspired by the biological neural networks in humans, with nodes designed to emulate the functions of biological neurons.

Within the framework of neural networks, an artificial neuron receives multiple inputs, each ascribed a specific weight reflecting its relative importance. The neuron processes these inputs by computing a weighted sum, which is then transformed by an activation function to produce an output. This output may either serve as an input to subsequent neurons in the network or, when emanating from the final layer, represent the network's aggregate output.

A single neuron, executes a mathematical operation where it transforms an input vector x by computing a weighted sum with its corresponding weight vector w, and an added bias term b. The resulting expression is as follows:

$$\mathbf{x} \mapsto \sum_{i=0}^{n} w_i x_i + b = \mathbf{w} \cdot \mathbf{x} + b$$

Equation above encapsulates a sequence of processes: each input value x_i is multiplied by an associated weight w_i, these products are then accumulated, and a bias b is incorporated into this summation. Subsequently, the aggregated weighted input plus the bias is subjected to an activation function g, which modulates the neuron's output as:

$$g(\mathbf{w} \cdot \mathbf{x} + b)$$

This operation culminates in the neuron's final output, establishing the basis for the neuron's activation and subsequent data propagation through the network architecture.

A Neural Network (NN) consists of numerous interconnected neurons organized into a series of layers. Given an n-dimensional input vector, the initial layer—referred to as the input layer—features n neurons corresponding to each dimension of the input. The terminal layer, known as the output layer, comprises t*t* neurons, tailored to the dimensionality of the desired output. The layers residing between the input and output layers are termed hidden layers. The total count of these layers constitutes the depth of the Neural Network, which is indicative of the model's complexity.

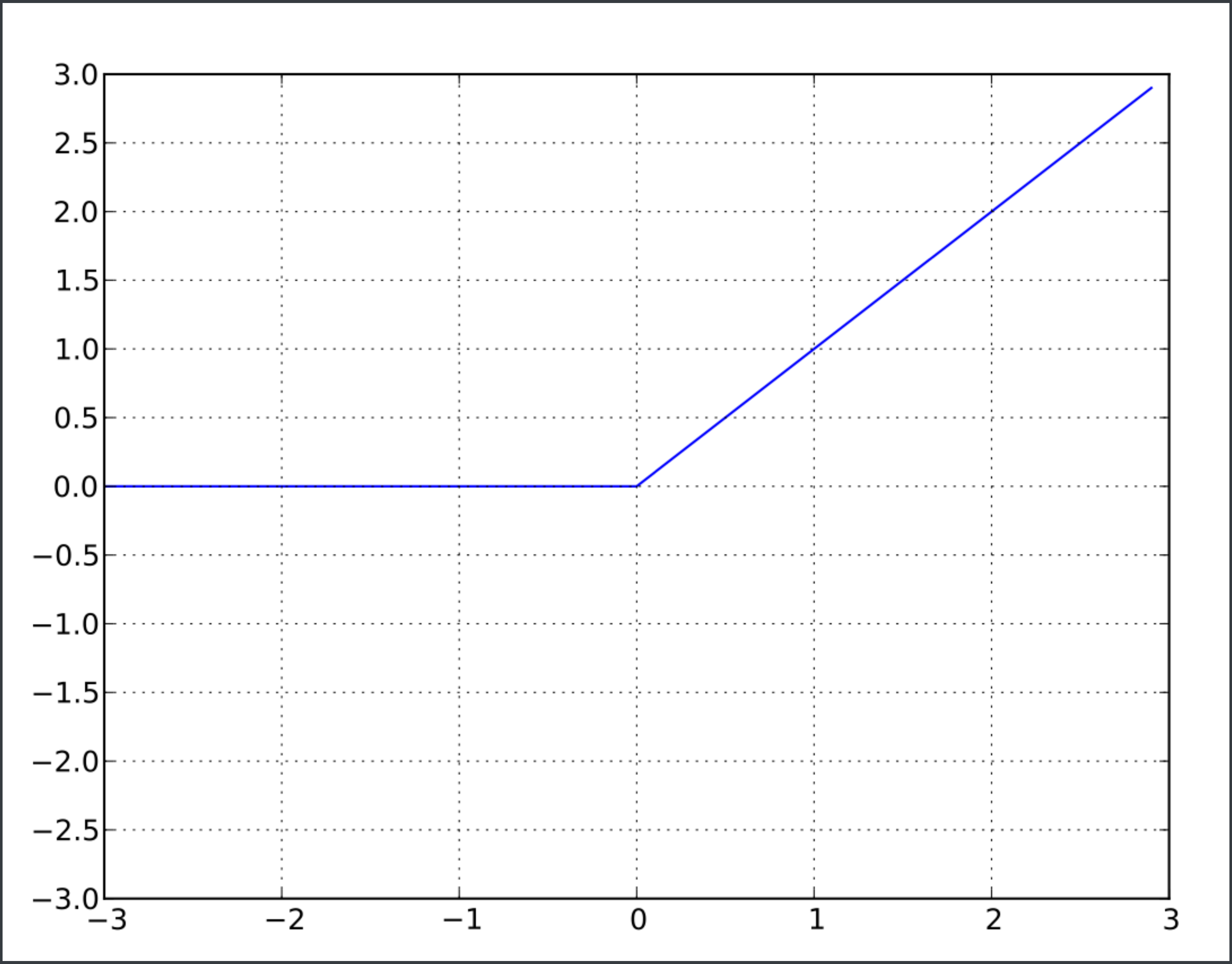
Activation Functions

Activation functions enable Neural Networks (NNs) with the dynamic capability to decode and articulate intricate patterns within data, thereby facilitating the representation of complex, non-linear relationships between inputs and outputs. The introduction of non-linearity via these activation functions enables the NN to approximate non-linear and highly sophisticated functional mappings. A critical attribute of an activation function is its differentiability, a property that is essential for the application of backpropagation algorithms. These algorithms compute gradients necessary for optimizing the network's weights. Through the iterative process of backpropagation and employing optimization techniques such as Gradient Descent, the network fine-tunes its weights, thereby incrementally minimizing the loss or error metric.

In the analytical exploration of the phishing dataset, we employ two pivotal activation functions: the Rectified Linear Unit (ReLU) and the Sigmoid. ReLU, recognized for promoting sparsity and efficient gradient propagation, serves as the activation function in the hidden layers, enhancing the network's capacity to model complex, non-linear decision boundaries. For the output layer, the Sigmoid function is utilized, especially due to its bounded and smooth gradient properties, which are ideal for binary classification tasks inherent in phishing detection. This combination harnesses the strengths of both activation functions, facilitating the network's ability to learn and generalize from the phishing dataset effectively.

ReLU Activation

The Rectified Linear Unit (ReLU) activation function is defined mathematically as $g(a) = reclin(a) = max(0, a)$. It possesses a lower bound at zero, ensuring that the output is non-negative across its domain. This characteristic of ReLU often results in sparse activation within the network, where a significant proportion of the neurons output zero, effectively deactivating those pathways. Additionally, the function is unbounded above, allowing for a range of activation without an upper limit. It also exhibits a strictly increasing nature, which aids in mitigating the vanishing gradient problem by maintaining strong gradients for positive input values.



Sigmoid Activation

The sigmoid activation function is a bounded, differentiable real function that is defined by the relation

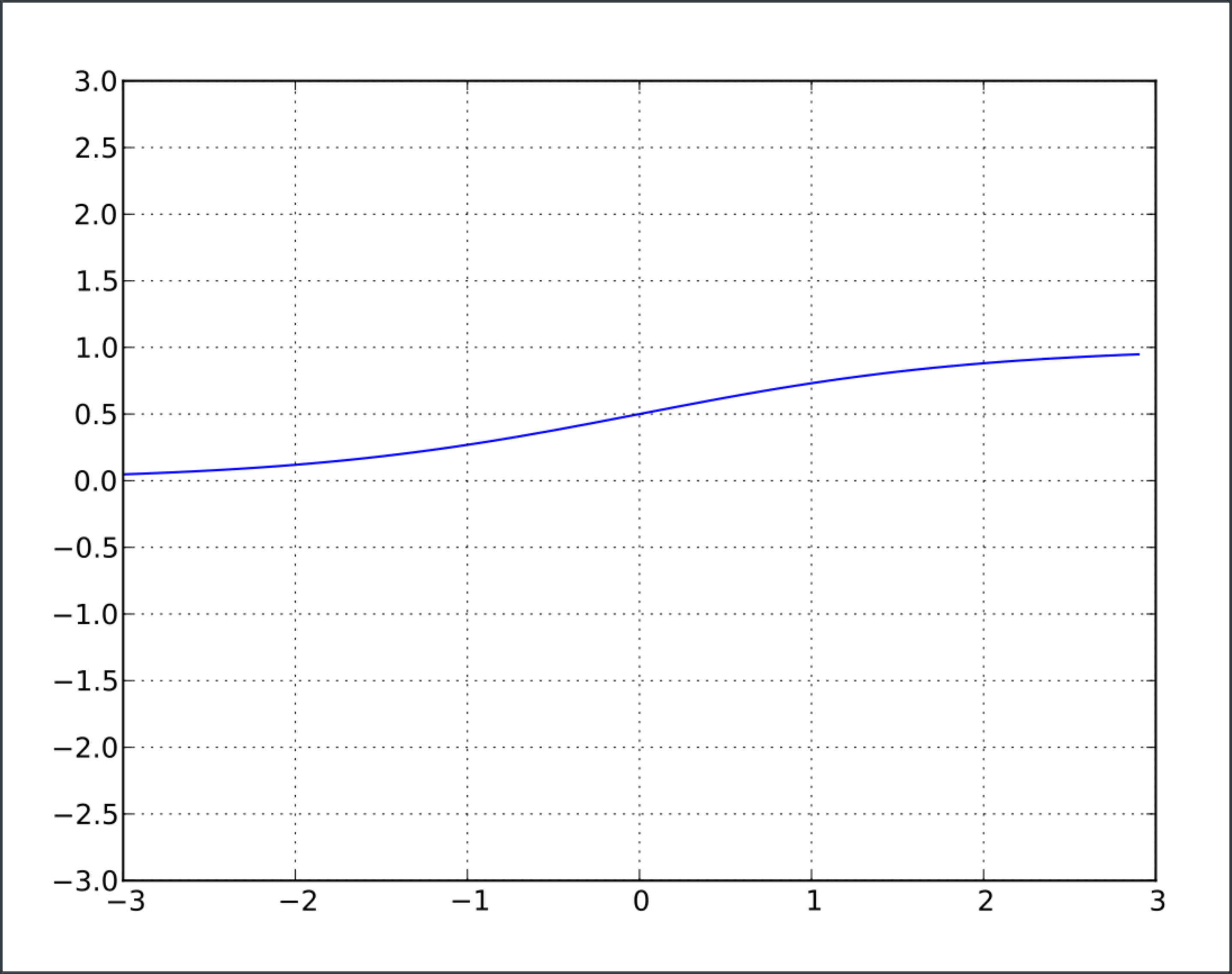
$$f(x) = \frac{1}{1+e^{-x}}$$

which maps any real-valued number into the interval (0, 1). Due to its smooth, monotonic, and S-shaped curve, it serves as an excellent threshold function. The derivative of the sigmoid function, which is essential for the backpropagation algorithm, is given by

$$f'(x) = f(x)(1 - f(x))$$

indicating that it is continuously differentiable and its slope is steepest at $x = 0$

A noteworthy aspect of the sigmoid function is that it is not symmetrical around the origin, leading to outputs that are always positive. Consequently, neuron activations will consistently be of the same sign, a phenomenon that can lead to undesirable effects in the gradient-based learning process. However, this can be mitigated by scaling the sigmoid function to adjust its range and centering its output around zero.



Loss Functions

The loss function serves to quantify the discrepancy between the anticipated outcomes of a model and the true data points.

This computed value, known as the loss, serves as an indicator of the model's predictive precision. Throughout the model's training phase, optimization algorithms such as backpropagation leverage the derivative of the loss function relative to the model's parameters. This process iteratively refines the parameters to reduce the loss, thereby enhancing the model's predictive accuracy on the given dataset.

Binary Cross Entropy

As we deal with a classification problem in the phishing dataset, we use Binary Cross Entropy (BCE) loss function as the loss function of preference. Also known in the literature as log loss, BCE is a prevalent loss function for dichotomous classification tasks. It quantifies the divergence between the predicted probability assigned to the occurrence of a class and the actual label of that class. Originating from the field of information theory, cross-entropy is a metric for assessing the dissimilarity of two probability distributions.

In the context of binary classification, the actual class label is typically encoded in a one-hot vector with the true category denoted by the value one, and the non-occurring category by zero. Conversely, the model's predictions are articulated as a probability vector, with $p(y = 1 \mid x)$ symbolizing the predicted likelihood of the positive class and $p(y = 0 \mid x)$ representing the complementary class.

The formulation of the BCE loss function is expressed as:

$$L(y, p) = -(ylog(p) + (1 - y)log(1 - p))$$

This expression can be conceptually bifurcated depending on the true class label y:

$-log(p)$ if $y = 1$
$-log(1 - p)$ if $y = 0$

Here, y denotes the true class label, and p signifies the predicted probability for the class of interest. Optimization of the model parameters seeks to minimize this loss function, ideally aligning the predicted probabilities pp with the true labels y.

The binary cross-entropy loss is esteemed for several properties: computational tractability, smooth gradients conducive to optimization, and a probabilistic interpretation of the classifier's outputs. Moreover, it tends to present a more forgiving optimization landscape and demonstrates resilience to anomalous data points relative to alternative loss functions. Nevertheless, it is not without its susceptibilities, notably to the issue of class imbalance wherein a preponderance of instances from one class could bias the model's learning trajectory.

Optimization Functions

Optimization represents a crucial sub-discipline within applied mathematics, playing an equally vital role in machine learning. The training of any machine learning model inherently involves optimization processes aimed at minimizing the loss function. In this context, the model parameters serve as decision variables that are iteratively adjusted during training. The dataset used for this purpose is generally treated as a fixed entity, providing a consistent framework against which the effectiveness of parameter adjustments is measured.

Optimization algorithms, for example gradient descent, are fundamentally designed to minimize an objective function. In the realm of machine learning, the process of "learning" is essentially synonymous with "optimization." Therefore, a deep comprehension of how optimization algorithms are implemented is vital. These algorithms are pivotal in reducing the training loss, thereby directly influencing the effectiveness of the learning process in machine learning models.

ADAM

Machine learning optimization techniques universally employ some variation of the descent direction method. Basic gradient descent, a fundamental example of this method, presents several limitations. Although some of these challenges are mitigated through the use of stochastic gradient descent or mini-batch processing, these adaptations still leave considerable scope for enhancement.

A significant advancement beyond basic gradient descent is the ADAM algorithm, an acronym for adaptive moment estimation. ADAM has emerged as the go-to method for training machine learning models in practical settings. Operating as a first-order optimization method, ADAM utilizes gradient information (first derivatives) to ascertain the optimal descent step. It innovatively integrates the concepts of momentum and adaptive learning rates for each parameter, thereby refining the computation of the descent step and enhancing overall optimization efficacy.

1 - Exponential Smoothing

A fundamental characteristic of the ADAM algorithm and similar first-order optimization methods is their ability to adjust to the local characteristics of the loss landscape as the iterations progress. These methods utilize only the historical gradient information available up to the current iteration to inform the descent steps. This approach, known as adaptive optimization, modifies the descent steps based on the gradient history, often placing more weight on recent gradients to reflect the most current landscape dynamics.

Adaptive optimization commonly employs exponential smoothing, a technique widely utilized not only in optimization but also in time series analysis and other areas of data science and applied mathematics. Exponential smoothing works by updating a sequence of vectors, which can be gradients or squared gradients, through a formula that blends the current vector with the accumulated past vectors in a weighted manner.

The smoothed sequence is updated as follows:

$$\overline{u}(t+1) = \beta\overline{u}(t) + (1-\beta)u(t),$$

with $\overline{u} = 0$ and β ranging between 0 and 1. For $t \geq 0$, each smoothed vector $\overline{u}(t+1)$ is a convex combination of the previous smoothed vector $\overline{u}(t)$ and the most recent vector $u(t)$. The value of β determines the extent of smoothing: a β close to 0 leads to less smoothing, emphasizing recent changes more significantly, whereas a higher β results in greater smoothing, dampening the influence of recent updates.

By applying this update rule iteratively, the smoothed vector $\overline{u}(t+1)$ effectively becomes a convex combination of all prior vectors, with the influence of each vector diminishing exponentially over time. This method allows for a gradual integration of all historical data into the optimization process, providing a robust mechanism for adapting the step size in response to evolving data patterns. Such straightforward application of exponential smoothing significantly enhances the adaptive capabilities of optimization algorithms.

2 - Momentum

It is well-understood that the step size in basic gradient descent is proportional to the negative gradient. This approach can stall in flat regions of the loss function, or at saddle points and local minima, where the gradient is negligible. To address these limitations, the concept of momentum is integrated into the optimization process.

To conceptualize momentum in this setting, we can consider the analogy of a ball rolling down a slope. As it descends, it accumulates speed, particularly along steeper sections, allowing it to maintain velocity even as it reaches more level ground. Similarly, the inclusion of a momentum term in the optimization process helps to maintain update velocity, effectively preventing premature stagnation and encouraging continuous progress toward the loss function's minima.

Momentum is mathematically introduced to the optimization as an additional step, defined as follows:

$v^{(t+1)} = \beta v^{(t)} + (1-\beta)\nabla C(\theta^{(t)})$ (Momentum Update)

$\theta^{(t+1)} = \theta^{(t)} - \alpha v^{(t+1)}$ (Parameter Update)

where β is the momentum parameter in the range [0, 1], α is the learning rate, and the initialization is typically $v(0) = 0$.

This approach can be likened to the method of exponential smoothing, where the momentum update can be seen as the smoothing of gradient vectors. With β set to zero, the update reverts to the standard gradient descent. As β increases, the optimization increasingly relies on the aggregate of past gradients, enhancing the stability and efficacy of the descent, particularly within the context of noisy gradients encountered in stochastic gradient descent or mini-batch methods.

The sequence $v(t)$ referred to in the momentum update captures the weighted sum of all preceding gradients,

$v^{(t+1)} = (1-\beta)\sum_{\tau=0}^{t} \beta^{t-\tau} \nabla C(\theta^{(\tau)})$ for $t = 1,\ 2,\ 3,\ \ldots$

with the weight of each gradient diminishing exponentially over time. Thus, with a non-zero β, the direction of the parameter update is influenced by the momentum of accumulated gradients, which helps in navigating the parameter space more effectively, particularly in complex loss landscapes.

3 - Adaptive Learning Rates

In the classical approach to gradient descent optimization, a single learning rate is applied universally to all parameter updates within the vector θ. However, the optimization process can often be enhanced by assigning unique learning rates to each parameter θ_i. This allows for tailored adjustments to the learning process, accommodating the idiosyncrasies of each parameter's influence on the model's performance.

To address this, it is advantageous to modulate the descent step for each parameter independently. This can be achieved by scaling the gradient vector with a unique factor for each element. Thus, rather than taking a standard gradient descent step, $\theta_s = -\alpha \nabla C(\theta)$, we adapt the update to be

$\theta_s = -\alpha r \odot \nabla \mathcal{L}(\theta)$

where r represents a vector of scaling factors, and $\odot$ denotes element-wise multiplication.

In machine learning, where the parameter space is extensive and the mapping between parameters and features may be obscure, it's often practical to adopt adaptive methods for adjusting the descent step. These methods calibrate the update steps across iterations.

We consider two such adaptive strategies: adaptive subgradient methods, commonly known as Adagrad, and Root Mean Square Propagation (RMSprop). For a given parameter θ_i at the t-th iteration, the update rules for Adagrad and RMSprop are formalized as follows:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\alpha}{\sqrt{s_i^{(t+1)} + \epsilon}} \frac{\partial C(\theta^{(t)})}{\partial \theta_i}$$

where s_i is a sequence of non-negative values updated adaptively and ϵ is a small constant to prevent division by zero.

In this formula, $s_i^{(t+1)}$ embodies a cumulatively adjusted term reflecting the magnitude of past gradients. The term r_i, representing the ratio of the step size to s_i essentially inverts the gradient magnitude, effectively scaling the update inversely to the aggregate gradient magnitudes. This ensures a controlled and informed update for each parameter.

Expressed in vector form, the parameter update becomes:

$$s_i^{(t+1)} = \sum_{\tau=0}^{t} \left(\frac{\partial C(\theta^{(\tau)})}{\partial \theta_i} \right)^2$$

where $s^{(t)}$ is vectorized, and all operations are performed element-wise.

For Adagrad, $s_i^{(t)}$ is the accumulation of the squares of the gradients, and for RMSprop, it is a moving average that gives more weight to the recent gradients:

$$s_i^{(t+1)} = \sum_{\tau=0}^{t} \left(\frac{\partial C(\theta^{(\tau)})}{\partial \theta_i} \right)^2 \text{ for Adagrad}$$

$$s_i^{(t+1)} = \gamma s_i^{(t)} + (1-\gamma) \left(\frac{\partial C(\theta^{(t)})}{\partial \theta_i} \right)^2 \text{ for RMSprop,}$$

with RMSprop's initial condition $s_i^{(0)} = 0$ and γ as the decay parameter. RMSprop's $s_i^{(t)}$ thus represents an exponentially weighted average of the squared gradients.

Such adaptive methods embody the essence of exponential smoothing applied to optimization in deep learning, capturing the historical information of gradients to refine the learning process effectively.

Adagrad is often perceived as more straightforward than RMSprop because it doesn't require any complex recursive formulation or additional hyperparameters like γ. It's essentially just an accumulation of squared gradients. Yet, the major limitation with Adagrad is that its accumulated term $\{s_i^{(t)}\}$ is monotonically increasing, leading to a continuously diminishing effective learning rate. Consequently, the learning rate can become excessively small before the model has fully converged. RMSprop addresses this issue by applying exponential smoothing to the squared gradients, preventing the learning rate from shrinking too rapidly.

For RMSprop, the updated value of ss at each step is explicitly represented as:

$$s^{(t+1)} = (1-\gamma)\sum_{\tau=0}^{t} \gamma^{t-\tau} \left(\nabla C(\theta^{(\tau)}) \odot \nabla C(\theta^{(\tau)})\right),$$

whereas, Adagrad's update lacks the decaying factor $(1-\gamma)$ and the exponential term $\gamma^{t-\tau}$. This difference is fundamental in allowing RMSprop to adjust the learning rate more flexibly and responsively throughout the training process.

4 - Bias Correction

In both momentum and RMSprop algorithms, exponential smoothing is applied to gradients and squared gradients, starting with a zero-initial condition at the first iteration $(t = 0)$. For momentum, we initialize $v(0) = 0$ and for RMSprop, $s(0) = 0$. While these zero-initial conditions are logically sound, they can introduce a temporary bias, as the initial outputs of the smoothing are skewed towards zero, which may not reflect the true values especially if the parameters β or γ are set close to 1. Over time, the influence of this bias diminishes, but it can still affect the early stages of the optimization.

Bias correction is employed to counteract this initial bias. For instance, considering $\overline{u}(0) = 0$, the first update of the exponentially smoothed sequence is $\overline{u}^{(1)} = (1-\beta)u^{(0)}$. Since β is usually close to 1, $\overline{u}^{(1)}$ remains near zero even if $u0$ is not. This trend persists in $\overline{u}^{(2)}$, staying close to zero due to the equation

$$\overline{u}^{(2)} = \beta\overline{u}^{(1)} + (1-\beta)u^{(1)}$$

The early terms of the sequence $\overline{u}^{(0)},\ \overline{u}^{(1)},\ \ldots$ therefore tend to be biased towards zero, regardless of the starting vector $u^{(0)},\ u^{(1)},\ \ldots$ values. To address the bias in cases where the input vectors in the sequence are constant, say $u(t) = u$, we can derive the smoothed vector as

$\overline{u}^{(t+1)} = (1-\beta)\sum_{\tau=0}^{t}\beta^{\tau}u = (1-\beta^{t+1})u$, by computing the geometric sum of the series.

This bias correction mechanism is vital for accurately reflecting the momentum and rate of change in the parameter updates during the initial phase of optimization.

To correct for the initial bias introduced by exponential smoothing in cases where the input $u(t)$ is constant, one can normalize the smoothed value $\overline{u}(t)$ by dividing it by $(1-\beta^{t+1})$. This adjustment ensures that the smoothed values reflect a constant vector more accurately when $u(t) = u$

Furthermore, even when the input sequence $u(0), u(1), u(2), \ldots$ is not strictly constant but varies slightly around a constant value, applying this bias correction—using the sequence $\overline{u}(t)/(1-\beta^{t})$—improves the approximation of the true values, particularly at the start of the sequence. As the number of iterations tt increases, the term β^{t} approaches zero, and the impact of the bias correction diminishes, becoming negligible as the algorithm proceeds.

Bias correction can be aptly applied to optimization techniques such as momentum and RMSprop to address the distortions in early iterations. The adjustments, as specifically detailed in previous formulas, result in the following bias-corrected expressions for the smoothed values v and s:

$\hat{v}^{(t+1)} = \frac{v^{(t+1)}}{1-\beta^{t+1}}$ for bias-corrected momentum term,

$\hat{s}^{(t+1)} = \frac{s^{(t+1)}}{1-\gamma^{t+1}}$ for bias-corrected RMSprop term

for each iteration $t = 0, 1, 2, \ldots$ These bias-corrected terms provide a more accurate estimate early on by compensating for the initial conditions, ensuring that the momentum and RMSprop updates more closely reflect the true gradient information as training progresses.

5 - ADAM Algorithm

Synthesizing the concepts of momentum, RMSprop, and bias correction, we arrive at the adaptive moment estimation method, widely known as ADAM. To draw an analogy, if momentum can be likened to a ball rolling down a slope, then ADAM represents a more complex scenario where the ball, heavy with friction, negotiates the incline.

The central update formula for ADAM is described as:

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\alpha}{\sqrt{\hat{s}^{(t+1)}}+\epsilon}\hat{v}^{(t+1)}$$

In this equation, vector operations such as division, square root, and the epsilon addition are executed on an element-wise basis. The terms $v^{(t+1)}$ and $s^{(t+1)}$ represent the bias-corrected exponentially smoothed estimates of the first and second moments, respectively, ensuring more accurate step sizes during the optimization process. Here is the algorithm for ADAM:

ADAM

```
    Input: Dataset $D = \{(x^{(1)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\}$,
           objective function $C(\cdot) = C(\cdot; D)$, and
           initial parameter vector $\theta_{init}$
    Output: Approximately $optimal\ \theta$

  $t \leftarrow 0$                                        (Initialize iteration counter)

  $\theta \leftarrow \theta_{init}$  $v \leftarrow 0$  $s \leftarrow 0$                     (Initialize state vectors)

  repeat
  $g \leftarrow \nabla C(\theta)$                                   (Compute gradient)
```

```
$v \leftarrow \beta v + (1-\beta) g$                          (Momentum update)
$s \leftarrow \gamma s + (1-\gamma)(g \odot g)$               (Second moment update)

$\hat{v} \leftarrow \frac{v}{1-\beta^{t+1}}$                  (Bias correction)
$\hat{s} \leftarrow \frac{s}{1-\gamma^{t+1}}$                 (Bias correction)

$\theta \leftarrow \theta - \alpha \frac{\hat{v}}{\sqrt{\hat{s}}+\epsilon} \hat{v}$     (Update parameters)

  $t \leftarrow t + 1$
  **until** $termination\ condition$ is satisfied
  **return** $\theta$
```