STAT 574 Final Project: Phishing URL Detection Via an L1 Regularized Artificial Neural Network Pipeline

A Project By

Cory Suzuki

May 7, 2025

In Partial Fulfillment of the Requirements

For the Masters of Science in

Applied Statistics

California State University, Long Beach

California, United States of America

This Project has been approved by

Dr. Olga Korosteleva

Table of Contents

May 7, 2025

Contents

1 Introduction

The internet is widely used today by many people as a way to access information, conduct scientific research, and store large volumes of big data. Our credit cards and online banking applications also rely on a secure, safe connection to the internet. While this technology has furthered our progress and eased humanity's way of living, there are naturally consequences and shortcomings of such a beneficial tool. Namely, concerns of cybersecurity have arisen due to the risk of phishing scams and security breaches. For example, the anti-phishing software company KeepNet reported that 56% of companies have their data breached through phishing scams on a weekly basis [Keepnet, 2025].

The main intention of this project is to demonstrate the application of significant supervised machine learning models to classify and accurately predict if a particular URL address is a phishing scam or not. In our dataset, the PHIUSIIL Phishing Dataset provided by the UCI Machine Learning Repository through a research paper written by Prasad and Chandra in 2023. We denote 1 to indicate the URL to be a phishing URL and 0 otherwise. The variable "label" in our dataset holds this information and will be set as our target variable of interest for the implementation of our various supervised learning models. In this project, we introduce the Artificial Neural network architectural pipeline to determine the best, optimized model for binary classification of phishing URL's, and compare the unoptimized versus the optimized model via regularization parameters and dropout layers to analyze which one has the better prediction accuracy on unseen data. We will also take into consider validation and training set learning curves to analyze the loss over the training epochs.

2 Data Wrangling and Exploratory Data Analysis

The first step that was taken in the Exploratory Data Analysis (EDA) stage was to capture the distributional shapes and analyze the skewness of the features within the dataset. A perfect way of capturing such insight was to utilize the power of density histograms. Out of all 45 variables, one pattern that arose often was the fact that a majority of the distributions were either left or right-skewed with prominent outliers in the tails. For example, the distribution of Spacial Characters in URL's is right-skewed,

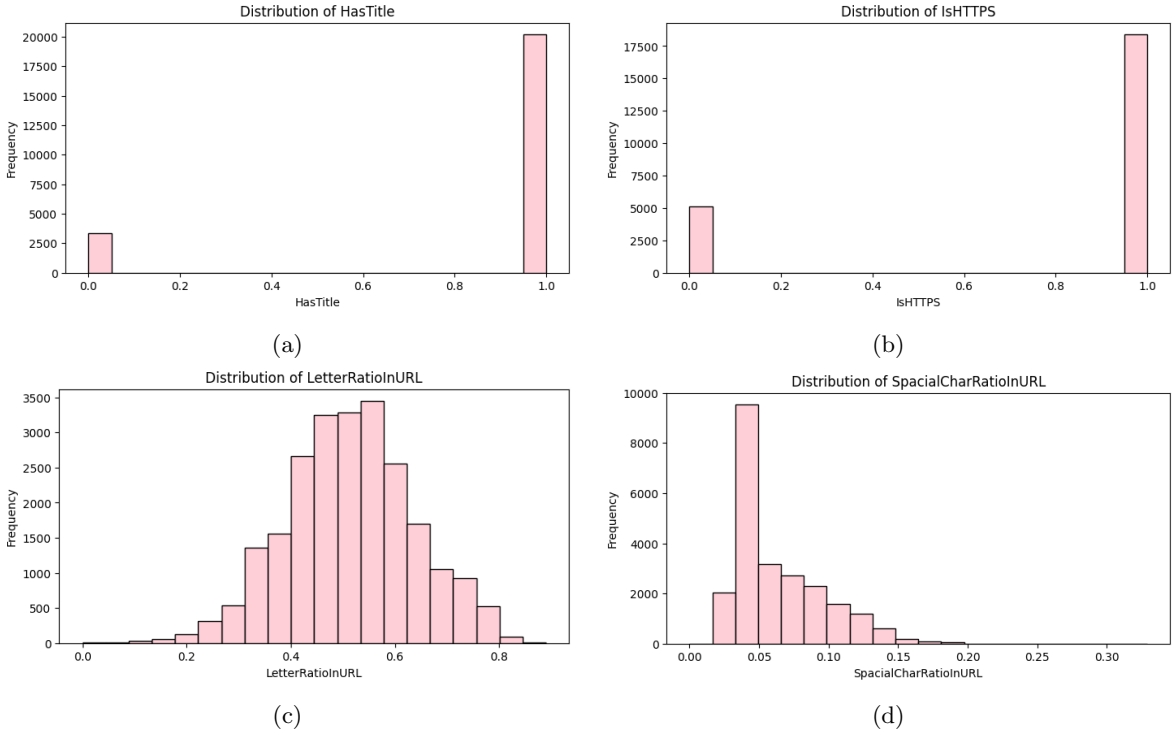


Figure 1: Distributional Histograms EDA

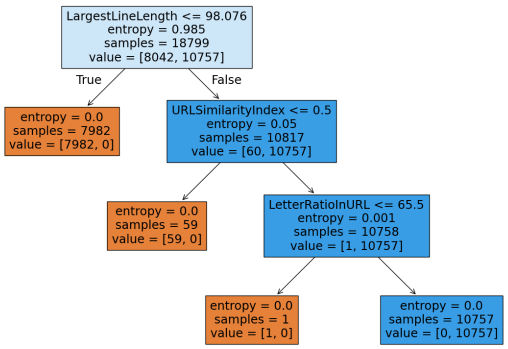
while one notable example of an approximately normal distribution was seen from the distribution belonging to that of the Letter Ratio in the URL's. This clearly indicates that a letter ratio of 50% occurs roughly 3450 times in terms of frequency. In addition, we provide two distributions of the frequencies of two binary features for the reader's reference to observe the trend that a minority of URL's do not have titles or HTTPS in them, which may imply the presence of phishing URL's. As part of the EDA, we use both feature importance loss metrics from a preliminary random forest model fitting and a Spearman Correlation matrix in order to perform feature selection. In this project, we end up considering the following variables: LargestLineLength, URLSimilarityIndex, LetterRatioInURL, URLCharProb, NoOfSubDomain, IsHTTPS, NoOfExternalRef, CharContinuationRate, HasTitle, NoOfSelfRedirect, URLLength, SpacialCharRatioInURL, NoOfDegitsInURL, and TLDLegitimateProb. Below is a table that summarizes each of these features that will be used in this paper.

Variable	**Type**	**Description**	**Responses**
Label (Target)	Binary	URL is spam or not	- 1 (yes) - 0 (no)
LargestLineLength	Continuous	Largest Line length of URL	Numeric Value
URLSimilarityIndex	Continuous	How similar a URL is to another	Numeric Value index
LetterRatioInURL	Continuous	Letter Ratio in URL	Numeric Value
URLCharProb	Continuous	Probability of character in URL	Numeric Value
NoOfSubDomain	Multinomial	Number of Subdomains in URL	Finite Integer Values
IsHTTPS	Binary	Does the URL have an secure HTTP?	- 1 (yes) - 0 (no)
NoOfExternalRef	Multinomial	Number of external reference encryptions in URL	Finite Integer Values
CharContinuationRate	Continuous	Rate of Character continuations in URL	Numeric Value
HasTitle	Binary	Does the URL have a title in it?	- 1 (yes) - 0 (no)
NoOfSelfRedirect	Multinomial	Number of Redirects URL deploys to itself	Finite Integer Values
URLLength	Continuous	Length of the URL	Numeric Values
SpacialCharRatioInURL	Continuous	Number of spaces are in a URL	Numeric Values
NoOfDegitsInURL	Multinomial	Number of Digits in URL	Finite Integer Values
TLDLegitimateProb	Continuous	Probability that a Top Level Domain (TLD) or informally the suffix domain of a URL is legitimate	Numeric Value

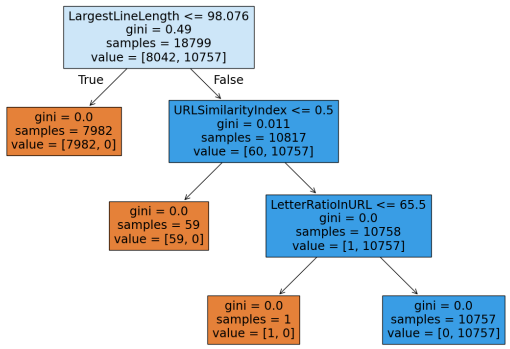
Table 1: Table Summary of Variables

After the exploratory data analysis, it was then appropriate to consider scaling all continuous features to fall within the interval [0,1]. The cleaning technique that was used was by instantiating two arrays, one that contains all numerical variables and another array that has the remaining nominal variables. Then, a column transformer pipeline was used which applied Sci-Kit Learn's Standard Scaler's transformation scaler function to convert the desired array of continuous features, while the "bypass" argument was used to not transform the nominal variables. Then, once the feature design matrix and target vector of phishing labels was defined, there is one concern that was addressed prior to model fitting.

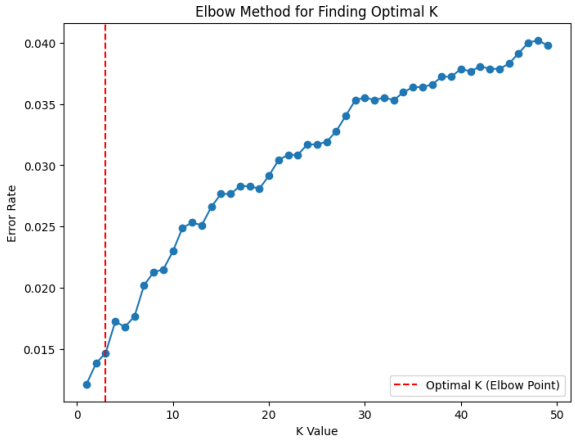
Instead of using the traditional 80% training and 20% testing set partition split, a variation of this split which includes the creation of a separate validation set split. In the literature *Hands-on Machine Learning*, Aurelion mentions that it is practical to partition a fraction of the test set into a validation set in order to reduce the possibility of model overfitting and to have an easier way to track the learning history over the training epochs, [Aurelion, 2017]. Therefore, we proceeded with the following partition scheme for our neural networks: 80% training, 10% validation, and 10% testing set splits. For the other supervised learning classifiers, the traditional 80%-20% data partition split was used, which include Decision Tree, Random Forest, Gradient Boosting, Naïve Bayes, K Nearest Neighbors,



(a) Entropy Criterion



(b) Gini Criterion



(c) Elbow Method to Select K

Figure 2: Decision Tree Hierarchy and KNN K-Selection

and Support Vector Machine classifiers. The accuracies of all our models will be compared in the next model comparisons section of this paper.

3 Model Comparisons Between Binary Classifiers

As a component of our analysis of the PhiUSIIL Phishing dataset, we first consider fitting the binary classifiers covered in the theory and code lecture notes from STAT 574 authored by Dr. Olga Korosteleva [Korosteleva, 2025]. For example, we consider fitting the following supervised models for binary classification which includes: Decision Trees (both Gini and Entropy Criterions considered), Random Forest Classifier, Gradient Boosting Classifier, K Nearest Neighbors Classifier, Naïve Bayes Classifier, and Support Vector Machine Classifier (linear, polynomial, radial, and sigmoidal kernels were all considered). A table of all the accuracies from these models has been provided for the reader's reference. One remark to take note of is that the highest accuracy for the Support Vector Machine model was attributed to the Linear Kernel with a remarkable accuracy of 100%, which may indicate that the SVM does not generalize the testing data well, which is also evident by the jump in lower accuracies for the other kernels.

As shown in the graph, the Elbow Method introduced in STAT 576 was used to choose the optimal value for the hyperparameter k. We choose the value k if it produces the smallest or sharpest change in the error rates out of potential testing values for k. Since our features have been reduced from the EDA, only the values from 1 to 50 were considered and iteratively fitted with the K Neighbors Classifier. Out of all 50 trials, it has been shown that k=3 is the optimal hyperparameter, which yielded a test accuracy of 99.98%. Out of all the baseline classifiers that were fit prior to fitting the Feed-Forward Neural Network, the results indicate that the Naïve Bayes classifier had the lowest accuracy of 99.17%,

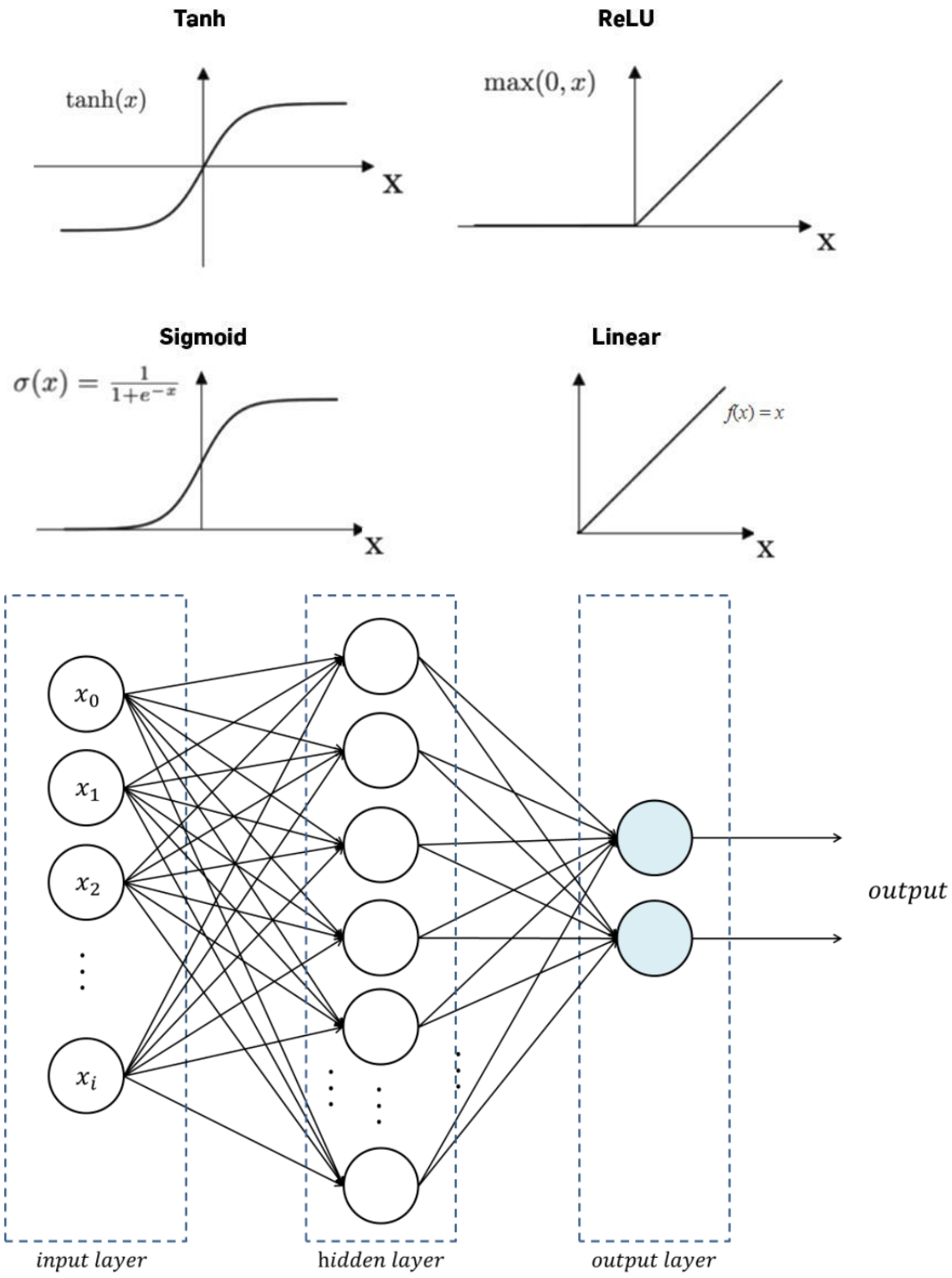
Binary Classifier	**Accuracy**	**Misclassification Rate**
Decision Tree (Entropy Criterion)	1.0000	0.0000
Decision Tree (Gini Criterion)	1.0000	0.0000
Random Forest (Entropy Criterion)	1.0000	0.0000
Random Forest (Gini Criterion)	1.0000	0.0000
K-Nearest Neighbors (KNN)	0.9853	0.0147
Gradient Boosting	0.9997	0.0003
Naïve Bayes	0.9881	0.0119
Support Vector Machine (Linear)	1.0000	0.0000
Support Vector Machine (Polynomial)	0.5915	0.4085
Support Vector Machine (Radial)	0.7651	0.2349
Support Vector Machine (Sigmoid)	0.3051	0.6949

Table 2: Comparison of Binary Classifiers based on Accuracy and Misclassification Rate

that is if we exclude the sigmoidal kernel for the support vector machine classifier which by far has yielded the least promising results.

4 Preliminaries: Theory of L1 Regularized Artificial Neural Networks for Binary Classification

The Artificial Neural Network (ANN) we introduce in this paper is an extension of the Artificial Neural Network, as covered in STAT 574. The model architecture of this model is therefore similar to the ANN but includes the ability to include more layers and neuronal node instantiations including L1 Regularization and Dropout [Moon, 2024]. For the baseline model, it is usually customary to define the input layer's number of nodes to be 2^m, where m is the number of layers initially defined in the ANN, and then for each layer, reduce the number of neurons by a factor of base 2. In each layer, an activation function can be explicitly defined, such as the Rectified Linear Unit Function (reLU), the Sigmoidal Function, the Leaky reLU Function, the Swish and Gish Functions, and the Softmax Function. It is ideal to first start with reLU activation functions in the input and hidden layers of the network and then select either the Sigmoidal or Softmax activation functions [Aurelion, 2017]. Since our dataset's target feature is defined as a binary variable, the sigmoidal function would be the best choice for the output layer's activation function since the sigmoidal function easily produces probabilities for the values 0 and 1, that is under the assumption of an appropriate predictive threshold value such as the standard, 0.5. This means that if the predicted probability is greater than 0.5, we classify the observation that produces that probability as a "1" and otherwise "0". If one is handling a multinomial classification problem, the Softmax activation function would be most ideal to produce a finite collection of probabilities that can split the observations into that same finite number of classes. Below, we provide the graphs of the hyperbolic tangent, reLU, sigmoid/logistic, and linear/identity activation functions.



In the above diagram, there are i many input data that is fed into the hidden layers of the network. For each input value in the input vector $\boldsymbol{x}$, there are weights and biases that are updated in each training epoch through the process of backpropagation. In reality, one can reframe the weights as recurringly updated coefficients and biases as penalties. The big idea is that ANN's share similarities with theoretical regression models [Moon, 2024]. After the weights are updated, the summation of the inner product between the inputs and the updated weights are then added to an updated bias vector term. This can be expressed in the following formula for each training epoch indexed by j:

$$\sum_{j=1}^{m} \boldsymbol{w_j}^T \boldsymbol{x_j} + \boldsymbol{b}$$

4.1 L1 Regularization and Dropout Layers

In the code for the ANN in this paper, some readers may be unfamiliar with the L1 regularization parameter in the Dense layers of our network architecture and the Dropout layers. Therefore, we will establish their theoretical significance here. To reduce overfitting from the ANN on unseen data, the L1 Regularizer in each Dense layer allows for larger weights to be penalized, so our chosen penalty parameter enables the network to maintain the robustness of our model in the L1 norm, which is very similar to the LASSO regularized regression algorithm introduced by Robert Tibshirani in 1996 [Lee, 2024]. There is also a dropout layer which uses the rate parameter as the ratio of input neurons in the next layer to be dropped or set to zero during training. This is also a common technique used to counter against model overfitting. With this theory established, the Artificial Neural Network can be applied to the Phishing dataset.

5 Model: Artificial Neural Network for Binary Classification (Python Implementation)

In this chapter, some basic components of the ANN algorithm can be broken down from the scripted Python code which is available for the reader's reference in Appendix A. The dataset is first loaded into the script via the Python library Pandas and is then preprocessed by splitting the dataset by the target vector and design matrix. As mentioned previously, our intention is to classify each observed email address and determine if the particular email address is a phishing URL or not. Therefore, the target vector $\boldsymbol{y}$ comprises of 1's and 0's. Our 45 features which include the number of spacial characters, and if the email address has been detected as robot-operated are isolated in the design matrix $\boldsymbol{X}$. Then after this, a Column Transformer is then used to standardize the dataset so only continuous features are scaled from [0,1]. The data is then partitioned into the appropriate training, validation, and testing sets prior to model fitting. Do note that we initialize the network with the following syntax:

```
        first_model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation="relu", input_shape=(45,),
                          kernel_regularizer=l2(0.001),
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Dense(16, activation="relu", kernel_regularizer=l2(0.001),
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

first_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                    loss="binary_crossentropy",
                    metrics=["accuracy"])

first_history = first_model.fit(X_train_transformed, y_train_transformed, epochs=100,
                                batch_size=32,
                                validation_data=(X_val_transformed,
                                                 y_valid_transformed))
```

After initializing the Dense input layer with the reLU activation function and the input shape, the network then utilizes a kernel regularizer in the L1 Norm to prevent overfitting during training. The kernel initializer is then set to the He Normal parameter, and in between each Dense layer, a Dropout

layer at a dropout rate of 0.5 is applied to prevent the network from overfitting. When compiling the model, the Adam optimizer is defined with an initial learning rate of 0.001 and the model is set to be trained on 100 epochs. Below, the learning performance curves for both the training and validation losses and accuracies are provided below.

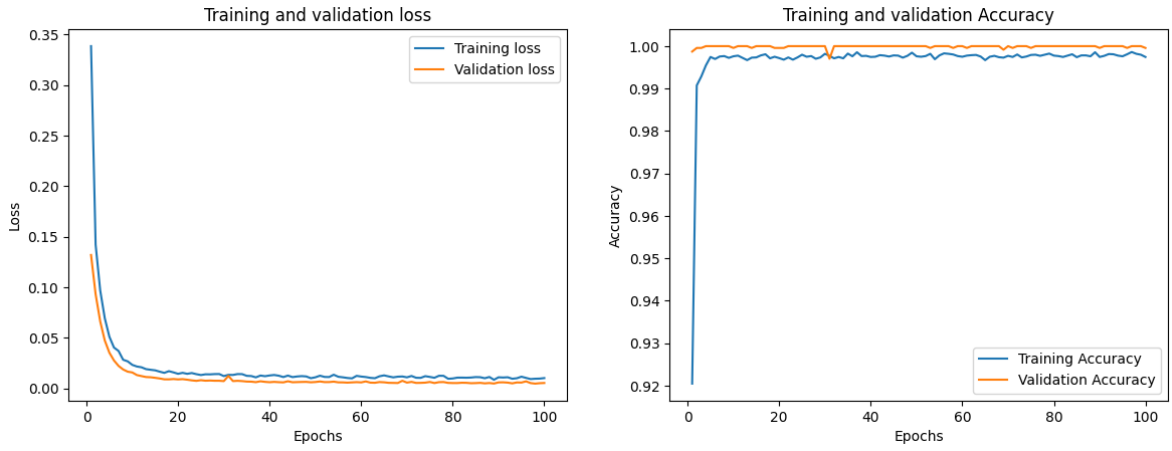
Both learning curves indicate that the first model fitted and trained on the dataset generalizes well, but some overfitting is present. This can be diagnosed by seeing the sudden increase and decrease or "spike" in the model's learning curves between the 20 and 40 epoch marks. Due to the presence of overfitting, it is now relevant to point out that the test accuracy for model A is 0.998, which is good but can be potentially improved upon with proper hyperparameter tuning in which the process and results promptly follow in the next chapter.

6 ANN Hyperparameter Tuning in Python

As previously mentioned, model A's learning curves provided some insight on potential overfitting issues with our initial hyperparameters. Therefore, the ANN pipeline now continues into the hyperparameter tuning of the number of neuronal nodes in totality, the number of Dense layers, L2 Kernel Regularizers, number of Dense layers, type of optimizer to compile, number of Dropout layers, and what type of activation functions should be used in each Dense layer. Since these hyperparameters can be computationally expensive to tune via the traditional cross-validation algorithm, the Keras library offers an alternative RandomTuner as introduced in Aurelion's Deep Learning textbook [Aurelion, 2017]. The RandomTuner is initialized in the Python script with the supplementary *skeleton model* which contains all possible combinations of hyperparameters for the ANN as shown.

```
def skeleton_model(hyper):
    n_hidden = hyper.Int("n_hidden", min_value=0, max_value=10, default=2)
    n_neurons = hyper.Int("n_neurons", min_value=16, max_value=256)
    learning_rate = hyper.Float("learning_rate", min_value=1e-4, max_value=1e-2,
    sampling="log")
    optimizer = hyper.Choice("optimizer",
                             values = ["SGD", "Adam", "RMSProp", "Nadam", "Adamax",
                                       ])

    if optimizer == "SGD":
        optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)
    elif optimizer == "Adam":
```



(a) Learning Curve for Losses

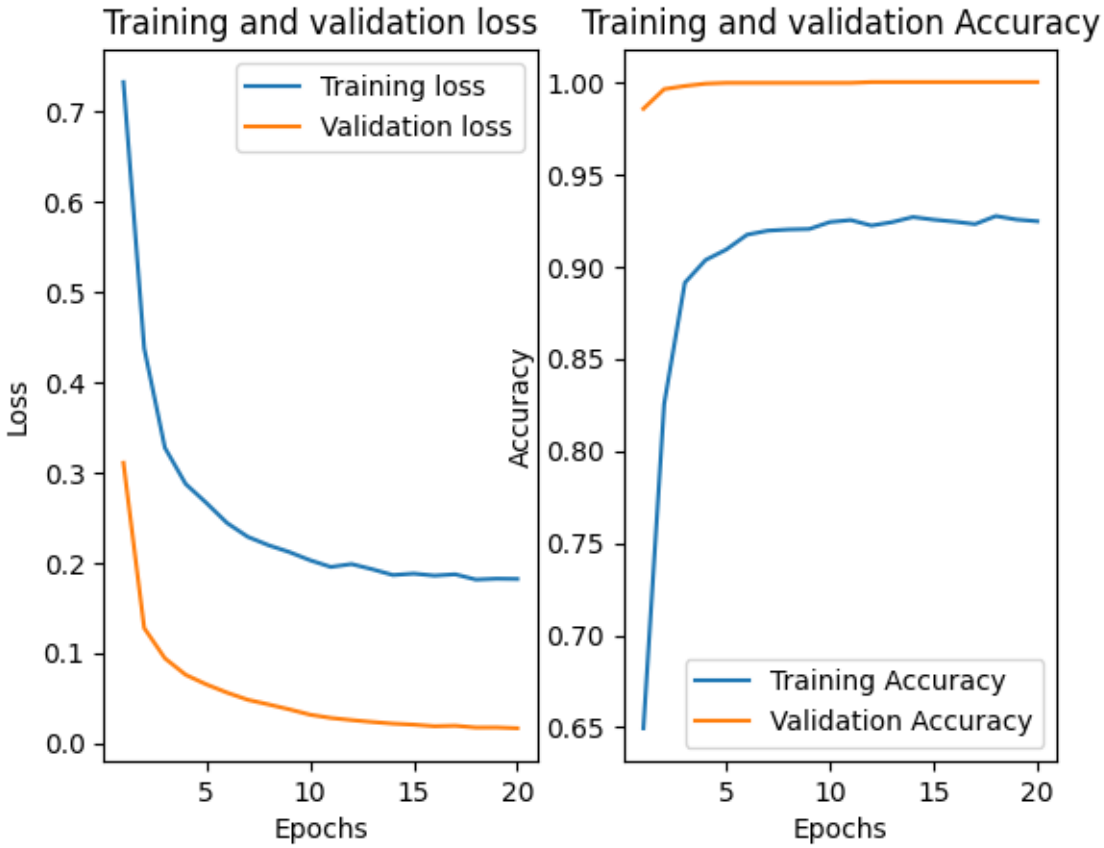
(b) Learning Curve for Accuracy

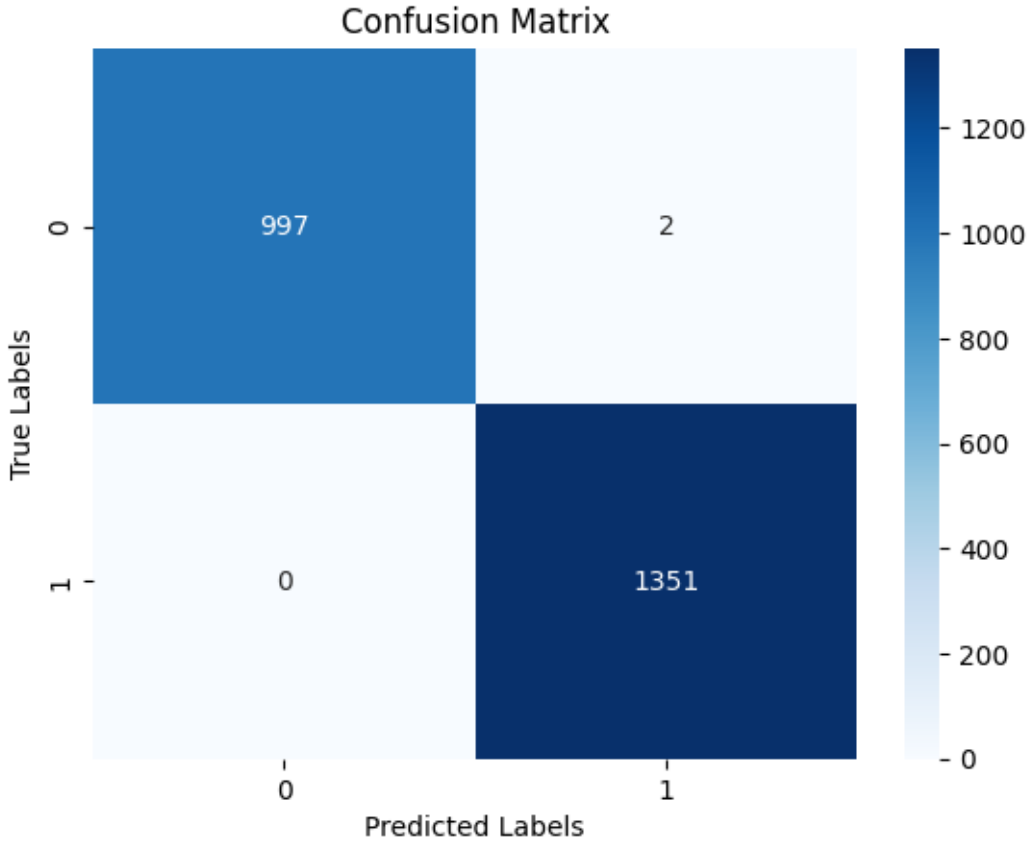
Figure 3: Learning Curves of Initial Model

```
            optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
        elif optimizer == "RMSProp":
            optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate,
            rho = 0.9)
        elif optimizer == "Nadam":
            optimizer = tf.keras.optimizers.Nadam(learning_rate=learning_rate)
        else:
            optimizer = tf.keras.optimizers.Adamax(learning_rate=learning_rate)

    samp_model = tf.keras.Sequential()
    samp_model.add(tf.keras.layers.Flatten())
    for _ in range(n_hidden):
        samp_model.add(tf.keras.layers.Dense(n_neurons, activation="relu"))
    samp_model.add(tf.keras.layers.Dense(1, activation="sigmoid"))
    samp_model.compile(loss="binary_crossentropy", optimizer=optimizer,
                       metrics=["accuracy"])
    return samp_model
```

After retrieving the top 3 hyperparameter combinations and rerunning our new final model, our learning curves exhibit no signs of overfitting after decreasing the number of training epochs and implementing the hyperparameters. In turn, this also consequently increased our test accuracy to 0.9991, which is significantly an overall better improvement as to model A. The final model's learning curves and confusion matrix is displayed, which surprisingly presents to us two False Positives in terms of model misclassifications. The next chapter will repeat the pipeline process presented here in R, and the differences in syntax and results will be discussed.





7 Artificial Neural Network for Binary Classification (R Implementation)

We now show the neural network implementation in R. In order to achieve this, the keras3 CRAN package was installed prior to model architecture construction. It can be noted that the below code snippet follows the similar syntax as Python, with the exception of using the pipe operator. Below is the code that uses the optimized hyperparameters from the Keras hyperparameter tuning performed in the previous section.

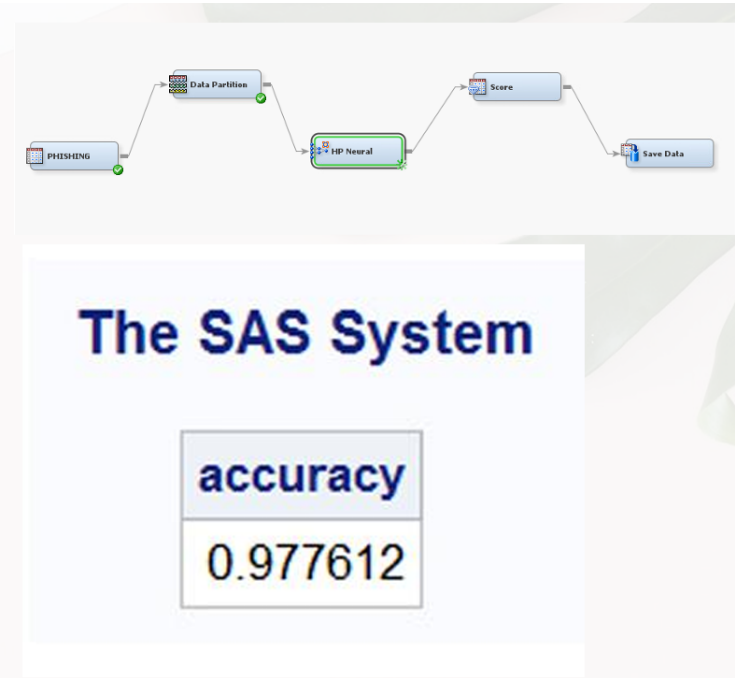
```
model_A = keras_model_sequential()
model_A %>% layer_dense(units=25, activation="relu",
input_shape=ncol(train_x),
kernel_regularizer=regularizer_l1(0.001),
kernel_initializer="he_normal")
model_A %>% layer_dropout(rate=0.5)
model_A %>% layer_dense(units=16, activation="relu",
kernel_regularizer=regularizer_l1(0.001),
kernel_initializer="he_normal")
model_A %>% layer_dropout(rate=0.5)
model_A %>% layer_dense(units=4, activation="relu",
kernel_regularizer=regularizer_l1(0.001),
kernel_initializer="he_normal")
model_A %>% layer_dropout(rate=0.5)
model_A %>% layer_dense(1, activation="sigmoid")

model_A %>% compile(loss="binary_crossentropy",
optimizer=optimizer_adam(learning_rate=0.0007),
metrics=c("accuracy"))
```

After computing the prediction accuracy on the testing data, the results indicate that our model has a 99.97% accuracy, which is much higher than the 99.91% test set accuracy achieved in Python. Therefore, it can be confirmed that our results are consistent with the output from the Python software.

8 Artificial Neural Network for Binary Classification (SAS Enterprise Miner Implementation)

After implementing the same hyperparameters, the following SAS Enterprise Miner Workflow diagram was executed to produce a prediction accuracy of 0.977612 on the testing data.



The advantage of using SAS Enterprise Miner for this dataset is that it can handle the entirety of the dataset, so no representative sampling is needed unlike how it was done in Python and R. It is important to note that our results here are considerably consistent with the high accuracies reported in the previous softwares utilized, hence validating the optimized ANN model. The SAS code snippet that was used to extract the test accuracy is provided in the Appendices.

9 Concluding Remarks and Future Work

In this paper, we explored the PHIUSIIL Phishing URL dataset by investigating the effectiveness of data mining techniques via supervised learning algorithms. After comparing the computed test set accuracies between various binary classification models such as Decision Trees, Random Forests, Gradient Boosting, Naïve Bayes, K Nearest Neighbors, all applicable kernel-based Support Vector Machines, and the Artificial Feed-Forward Neural Network, it has been established from our results that the ANN not only had the highest test set accuracy but generalized the test data well as evident from the learning curves. This conclusion was also validated in both Python and R, where similar results were produced. For future work, we wish to employ both supervised and unsupervised learning techniques from STAT 574 and 576 respectively and constructing an effective semi-supervised learning model as mentioned in the class lecture notes from STAT 576 [Lee, 2024]. In addition, we may consider reinforcement learning and transfer learning techniques which may allow for a better model analysis since we would be able to use cloud technologies to analyze the entire dataset as it was too large of a workload for personal machines.

10 Appendices: Source Codes in R and Python

10.1 Appendix A: Python Implementation for Feed-Forward Neural Network

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import keras as keras
import tensorflow as tf
import seaborn as sns
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder,
StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split, cross_val_score
import keras_tuner as kt
from sklearn.compose import make_column_transformer, ColumnTransformer
from tensorflow.keras.regularizers import l2
from sklearn.pipeline import Pipeline
from ucimlrepo import fetch_ucirepo
from sklearn.metrics import confusion_matrix, roc_auc_score, roc_curve

phishing_data = pd.read_csv("C:/Users/coryg/OneDrive/Desktop/STAT_574_Data_Mining
/PhiUSIIL_Phishing_URL_Dataset.csv", delimiter=",")
print(phishing_data.shape)
phishing_data.head(15)

phishing_data = phishing_data.drop(columns=["FILENAME", "URL", "Domain", "Title",
                                            "HasObfuscation", "NoOfObfuscatedChar",
                                            "ObfuscationRatio", "TLD",
                                            "NoOfAmpersandInURL","TLDLength",
                                            ], axis=1)
phishing_data.head(15)

phishing_data = phishing_data.drop_duplicates()
print(phishing_data.shape)
phishing_data.head(15)

# Take a representative random sample of the original
# population to reduce computational complexity and to ensure machine health.

phishing_samp = phishing_data.sample(n=23499, random_state=42)
phishing_samp.head(15)

# Check the distribution of each feature using histograms and describe()
for summary statistics.

print(phishing_samp.describe())

for column in phishing_samp:
    plt.figure(figsize=(8,4))
    sns.histplot(phishing_samp[column], bins=20, color="pink")
    plt.title(f"Distribution of {column}")
    plt.xlabel(column)
    plt.ylabel("Frequency")
    plt.show()
```

```
X_mat = phishing_samp.drop(["label"], axis=1)
y_target = phishing_samp.drop(["URLLength", "DomainLength", "IsDomainIP",
"CharContinuationRate","TLDLegitimateProb", "URLCharProb","NoOfSubDomain",
"NoOfLettersInURL", "LetterRatioInURL",
"NoOfDegitsInURL", "DegitRatioInURL", "NoOfEqualsInURL","NoOfQMarkInURL",
"NoOfOtherSpecialCharsInURL",
"SpacialCharRatioInURL", "IsHTTPS", "LineOfCode","LargestLineLength", "HasTitle",
"DomainTitleMatchScore","HasFavicon", "Robots", "IsResponsive", "NoOfURLRedirect",
"NoOfSelfRedirect", "HasDescription", "NoOfPopup",
"NoOfiFrame","HasExternalFormSubmit", "HasSocialNet", "HasSubmitButton",
"HasHiddenFields", "HasPasswordField", "Pay",
"Bank","Crypto", "HasCopyrightInfo", "NoOfImage", "NoOfCSS",
"NoOfJS", "NoOfSelfRef", "NoOfEmptyRef",
"URLSimilarityIndex", "URLTitleMatchScore",
"NoOfExternalRef"], axis=1)

X_mat.head(15)

print(y_target.value_counts())
y_target.head(15)

# Split data into training, validation, and testing sets.
# For this model, we want to balance the variances between the hyperparameter
# estimates and the performance statistics. So we want to have majority of the
# split attributing to training data, and then have even splits for the validation
# and test sets. Therefore we will proceed with a 80-10-10 split.

X_train, X_temp, y_train, y_temp = train_test_split(X_mat, y_target, test_size=0.2,
    random_state=364470)
X_valid, X_test, y_valid, y_test = train_test_split(X_temp, y_temp, test_size=0.5,
    random_state=426003)

numerical_cont = ["URLLength", "DomainLength", "URLSimilarityIndex",
"CharContinuationRate",
    "TLDLegitimateProb", "URLCharProb",
    "NoOfLettersInURL","LetterRatioInURL", "NoOfImage", "NoOfCSS",
    "NoOfJS","NoOfSelfRef","NoOfEmptyRef", "NoOfExternalRef",
    "NoOfSubDomain", "LargestLineLength", "LineOfCode", "NoOfDegitsInURL",
    "DegitRatioInURL", "NoOfEqualsInURL", "NoOfQMarkInURL",
    "NoOfOtherSpecialCharsInURL", "SpacialCharRatioInURL", "DomainTitleMatchScore",
    "URLTitleMatchScore", "NoOfURLRedirect", "NoOfSelfRedirect", "NoOfPopup",
    "NoOfiFrame"]

binary_feat = ["IsDomainIP", "Bank", "Pay", "Crypto", "HasCopyrightInfo",
        "IsHTTPS", "HasTitle", "HasFavicon", "Robots", "IsResponsive",
        "HasDescription", "HasExternalFormSubmit", "HasSocialNet",
        "HasSubmitButton", "HasHiddenFields", "HasPasswordField"]

preprocessor = ColumnTransformer(
    transformers=[
        ('continuous', StandardScaler(), numerical_cont),
        ('binary', 'passthrough', binary_feat)
    ])

X_train_transformed = preprocessor.fit_transform(X_train)
X_val_transformed = preprocessor.transform(X_valid)
```

```
X_test_transformed = preprocessor.transform(X_test)

print("Train shape:", X_train_transformed.shape)
print("Valid shape:", X_val_transformed.shape)
print("Test shape:", X_test_transformed.shape)

label_encoder = LabelEncoder()
y_train_transformed = label_encoder.fit_transform(y_train)
y_valid_transformed = label_encoder.transform(y_valid)
y_test_transformed = label_encoder.transform(y_test)

# AFNN Model A

first_model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation="relu", input_shape=(45,),
                          kernel_regularizer=l1(0.001),
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Dense(16, activation="relu", kernel_regularizer=l1(0.001),
                          kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

# Model A Compilation and Initial Run

first_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
                    loss="binary_crossentropy",
                    metrics=["accuracy"])

first_history = first_model.fit(X_train_transformed, y_train_transformed, epochs=100,
                                batch_size=32,
                                validation_data=(X_val_transformed,
                                                 y_valid_transformed))

# Model A Performance Metric Analysis: Training and Validation Loss Plots

loss = first_history.history['loss']
val_loss = first_history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss,  label='Training loss')
plt.plot(epochs, val_loss, label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()

y_preds_first = first_model.predict(X_test_transformed)
y_pred = np.where(y_preds_first>0.5, 1, 0)
cm_first = confusion_matrix(y_test_transformed, y_pred)
cm_first
```

```
sns.heatmap(cm_first, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()

roc_auc_sc1 = roc_auc_score(y_test_transformed, y_pred)
print(roc_auc_sc1)

def skeleton_model(hyper):
    n_hidden = hyper.Int("n_hidden", min_value=0, max_value=10, default=2)
    n_neurons = hyper.Int("n_neurons", min_value=16, max_value=256)
    learning_rate = hyper.Float("learning_rate", min_value=1e-4, max_value=1e-2,
                                sampling="log")
    optimizer = hyper.Choice("optimizer",
                             values = ["SGD", "Adam", "RMSProp", "Nadam", "Adamax",
                                       ])

    if optimizer == "SGD":
        optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)
    elif optimizer == "Adam":
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    elif optimizer == "RMSProp":
        optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate,
                                                rho = 0.9)
    elif optimizer == "Nadam":
        optimizer = tf.keras.optimizers.Nadam(learning_rate=learning_rate)
    else:
        optimizer = tf.keras.optimizers.Adamax(learning_rate=learning_rate)

    samp_model = tf.keras.Sequential()
    samp_model.add(tf.keras.layers.Flatten())
    for _ in range(n_hidden):
        samp_model.add(tf.keras.layers.Dense(n_neurons, activation="relu"))
    samp_model.add(tf.keras.layers.Dense(1, activation="sigmoid"))
    samp_model.compile(loss="binary_crossentropy", optimizer=optimizer,
                       metrics=["accuracy"])
    return samp_model

random_search_tuner = kt.RandomSearch(
    skeleton_model, objective="val_accuracy", max_trials=6,
    seed=42
)
random_search_tuner.search(X_train_transformed, y_train_transformed, epochs=10,
verbose=1,validation_data=(X_val_transformed, y_valid_transformed))
top_3_hps = random_search_tuner.get_best_hyperparameters(num_trials=3)

for i, hp in enumerate(top_3_hps):
    print(f"Top-{i+1}-hyperparameters:")
    print(hp.values)

# Using the best hyperparameter set out of top 3 displayed.

first_model = tf.keras.Sequential([
    tf.keras.layers.Dense(25, activation="relu", input_shape=(45,),
                          kernel_regularizer=l1(0.001),
```

```
                            kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Dense(16, activation="relu", kernel_regularizer=l1(0.001),
                            kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Dense(4, activation="relu", kernel_regularizer=l2(0.001),
                            kernel_initializer="he_normal"),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

first_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0007),
                    loss="binary_crossentropy",
                    metrics=["accuracy"])

first_history = first_model.fit(X_train_transformed, y_train_transformed, epochs=20,
                                batch_size=32,
                                validation_data=(X_val_transformed,
                                                 y_valid_transformed))

loss = first_history.history['loss']
val_loss = first_history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.subplot(1,2,1)
plt.plot(epochs, loss,  label='Training loss')
plt.plot(epochs, val_loss, label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

accuracy = first_history.history['accuracy']
val_accuracy = first_history.history['val_accuracy']

plt.subplot(1,2,2)
plt.plot(epochs, accuracy,  label='Training Accuracy')
plt.plot(epochs, val_accuracy, label='Validation Accuracy')
plt.title('Training and validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()

test_loss, test_accuracy = first_model.evaluate(X_test_transformed, y_test_transformed)

print("Test loss {:.4f}".format(test_loss))
print("Test accuracy {:.4f}".format(test_accuracy))

y_preds_first = first_model.predict(X_test_transformed)
y_pred = np.where(y_preds_first >0.5, 1, 0)
cm_first = confusion_matrix(y_test_transformed, y_pred)

sns.heatmap(cm_first, annot=True, fmt='d', cmap='Blues')
```

```
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

10.2 Appendix B: Python Implementation of Supervised Binary Classifiers

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier

spam_data = pd.read_csv("C:/Users/coryg/OneDrive/Desktop/STAT_574_Data_Mining/
PhiUSIIL_Phishing_URL_Dataset.csv", delimiter=",")
spam_data = spam_data.drop(columns=["FILENAME", "URL", "Domain", "Title",
                                    "HasObfuscation", "NoOfObfuscatedChar",
                                    "ObfuscationRatio", "TLD",
                                    "NoOfAmpersandInURL","TLDLength",
                                    ], axis=1)

spam_data = spam_data.drop_duplicates()
spam_data = spam_data.sample(n=23499, random_state=115712)

X_mat = spam_data.drop(["label"], axis=1)
y_target = phishing_samp.drop(["URLLength", "DomainLength", "IsDomainIP",
"CharContinuationRate","TLDLegitimateProb", "URLCharProb","NoOfSubDomain",
"NoOfLettersInURL", "LetterRatioInURL",
"NoOfDegitsInURL", "DegitRatioInURL", "NoOfEqualsInURL","NoOfQMarkInURL",
"NoOfOtherSpecialCharsInURL",
"SpacialCharRatioInURL", "IsHTTPS", "LineOfCode","LargestLineLength", "HasTitle",
"DomainTitleMatchScore",
"HasFavicon", "Robots",
"IsResponsive", "NoOfURLRedirect",
"NoOfSelfRedirect", "HasDescription", "NoOfPopup", "NoOfiFrame",
"HasExternalFormSubmit", "HasSocialNet", "HasSubmitButton",
"HasHiddenFields", "HasPasswordField", "Pay",
"Bank","Crypto", "HasCopyrightInfo", "NoOfImage", "NoOfCSS",
"NoOfJS", "NoOfSelfRef", "NoOfEmptyRef",
"URLSimilarityIndex", "URLTitleMatchScore",
"NoOfExternalRef"],axis=1)

X_train, X_test, y_train, y_test = train_test_split(X_mat, y_target, test_size=0.2,
                                                    random_state=410205)
# Decision Tree Classifier

spam_gini_bintree = DecisionTreeClassifier(max_leaf_nodes=45, criterion="gini",
                                           random_state=364370)
spam_gini_fit = spam_gini_bintree.fit(X_train, y_train)
```

```
fig = plt.figure(figsize=(15, 10))
tree.plot_tree(spam_gini_fit, feature_names=["URLLength", "DomainLength", "IsDomainIP",
                "CharContinuationRate",
                "TLDLegitimateProb", "URLCharProb",
                "NoOfSubDomain", "NoOfLettersInURL", "LetterRatioInURL",
                "NoOfDegitsInURL", "DegitRatioInURL", "NoOfEqualsInURL",
                "NoOfQMarkInURL", "NoOfOtherSpecialCharsInURL",
                "SpacialCharRatioInURL", "IsHTTPS", "LineOfCode",
                "LargestLineLength", "HasTitle", "DomainTitleMatchScore",
                "HasFavicon", "Robots", "IsResponsive",
                "NoOfURLRedirect",
                "NoOfSelfRedirect", "HasDescription", "NoOfPopup", "NoOfiFrame",
                "HasExternalFormSubmit", "HasSocialNet", "HasSubmitButton",
                "HasHiddenFields", "HasPasswordField", "Pay", "Bank",
                "Crypto", "HasCopyrightInfo", "NoOfImage", "NoOfCSS",
                "NoOfJS", "NoOfSelfRef", "NoOfEmptyRef",
                "URLSimilarityIndex", "URLTitleMatchScore",
                "NoOfExternalRef"],
    filled=True)

# Computing prediction accuracy of Gini tree on testing set.

y_pred = spam_gini_fit.predict(X_test)
accuracy = accuracy_score(y_pred, y_test)
accuracy

# Feature Importance via RandomForest Classifier

rf_spam_gini = RandomForestClassifier(n_estimators=100, criterion="gini",
                random_state=234090, max_depth=50, max_features=45)
rf_spam_gini.fit(X_train, y_train)

feature_importances = pd.Series(rf_spam_gini.feature_importances_,

index=X_train.columns)
feature_importances = feature_importances.sort_values(ascending=False)
print(feature_importances)

y_pred_rf = rf_spam_gini.predict(X_test)
accuracy_rf = accuracy_score(y_pred_rf, y_test)
print(accuracy_rf)

# Feature Importance via Gradient Boosting Classifier

gbspam_params = {'n_estimators':1000, 'max_depth':45,
                 'learning_rate':0.1}
gbspam_class = GradientBoostingClassifier(**gbspam_params)
gbspam_class.fit(X_train, y_train)

# Variable Importance

var_names=pd.DataFrame(["URLLength", "DomainLength", "IsDomainIP",
                                "CharContinuationRate",
                                "TLDLegitimateProb", "URLCharProb",
                                "NoOfSubDomain", "NoOfLettersInURL", "LetterRatioInURL",
                                "NoOfDegitsInURL", "DegitRatioInURL", "NoOfEqualsInURL",
```

```
                                        "NoOfQMarkInURL", "NoOfOtherSpecialCharsInURL",
                                        "SpacialCharRatioInURL", "IsHTTPS", "LineOfCode",
                                        "LargestLineLength", "HasTitle", "DomainTitleMatchScore",
                                    "HasFavicon", "Robots", "IsResponsive", "NoOfURLRedirect",
                            "NoOfSelfRedirect", "HasDescription", "NoOfPopup", "NoOfiFrame",
                                    "HasExternalFormSubmit","HasSocialNet","HasSubmitButton",
                                        "HasHiddenFields", "HasPasswordField", "Pay", "Bank",
                                        "Crypto", "HasCopyrightInfo", "NoOfImage", "NoOfCSS",
                                        "NoOfJS", "NoOfSelfRef", "NoOfEmptyRef",
                                         "URLSimilarityIndex", "URLTitleMatchScore",
                                            "NoOfExternalRef"], columns=['var_name'])
loss_reduction=pd.DataFrame(gbspam_class.feature_importances_,
columns=['loss_reduction'])
var_importance=pd.concat([var_names,loss_reduction], axis=1)
print(var_importance.sort_values("loss_reduction", axis=0, ascending=False))

y_pred_gb = gbspam_class.predict(X_test)
accuracy_gb = accuracy_score(y_pred_gb, y_test)
print(accuracy_gb)

# Gaussian Naive Bayes Classifier

nb_spam = GaussianNB()
nb_spam.fit(X_train, y_train)
y_pred_nb = nb_spam.predict(X_test)

accuracy_nb = accuracy_score(y_pred_nb, y_test)
print(accuracy_nb)

k_values = range(1, 200)
acc_rates = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred_elbow = knn.predict(X_test)
    acc_rate = 1 - accuracy_score(y_test, y_pred_elbow)
    acc_rates.append(acc_rate)

plt.figure(figsize=(8, 6))
plt.plot(k_values, acc_rates, marker='o')
plt.xlabel('K-Value')
plt.ylabel('Error-Rate')
plt.title('Elbow-Method-for-Finding-Optimal-K')
plt.axvline(x=3, linestyle='--', color='red', label="Optimal-K-(Elbow-Point)")
plt.legend()
plt.show()

# K Neighbors Classifier

knn_spam = KNeighborsClassifier(n_neighbors=5)
knn_spam.fit(X_train, y_train)

y_pred_knn = knn_spam.predict(X_test)
accuracy_knn = accuracy_score(y_pred_knn, y_test)
print(accuracy_knn)
```

```
# Support Vector Classifier (using linear, polynomial, radial, and sigmoidal kernels)

svc_lin_spam = SVC(kernel="linear").fit(X_train, y_train)
svc_poly_spam = SVC(kernel="poly").fit(X_train, y_train)
svc_radial_spam = SVC(kernel="rbf").fit(X_train, y_train)
svc_sigmoid_spam = SVC(kernel="sigmoid").fit(X_train, y_train)

y_pred_lin_svc = svc_lin_spam.predict(X_test)
y_pred_poly_svc = svc_poly_spam.predict(X_test)
y_pred_rad_svc = svc_radial_spam.predict(X_test)
y_pred_sigmoid_svc = svc_sigmoid_spam.predict(X_test)

accuracy_linsvc = accuracy_score(y_pred_lin_svc, y_test)
accuracy_polysvc = accuracy_score(y_pred_poly_svc, y_test)
accuracy_radsvc = accuracy_score(y_pred_rad_svc, y_test)
accuracy_sigmoidsvc = accuracy_score(y_pred_sigmoid_svc, y_test)

print("Linear:", round(accuracy_linsvc, 4))
print("Polynomial:", round(accuracy_polysvc, 4))
print("Radial:", round(accuracy_radsvc, 4))
print("Sigmoid:", round(accuracy_sigmoidsvc, 4))
```

10.3 Appendix C: R Implementation for Artificial Neural Networks

```
library(readr)
library(dplyr)
library(keras3)

spam_data = read.csv("C:/Users/coryg/OneDrive/Desktop/STAT_574_Data_Mining
/PhiUSIIL_Phishing_URL_Dataset.csv",
header=T, sep=",")

spam_data = subset(spam_data, select=-c(FILENAME, URL, Domain, Title,
HasObfuscation, NoOfObfuscatedChar, ObfuscationRatio, TLD,
NoOfAmpersandInURL, TLDLength))

scale01 <- function(x) {
    (x-min(x))/(max(x)-min(x))
}

spam_data = spam_data %>% mutate_all(scale01)

# Split the data into 80% training, 10% validation, and 10% testing sets.

set.seed(312329)
sample = sample(c(T,F), nrow(spam_data), replace=T, prob=c(0.8,0.2))
train = spam_data[sample,]
split = spam_data[!sample,]

set.seed(251820)
sample2 = sample(c(T,F), nrow(split), replace=T, prob=c(0.5, 0.5))
validation = split[sample2,]
test = split[!sample2,]

train_x = data.matrix(train[-46])
```

```
train_y = data.matrix(train[46])
validation_x = data.matrix(validation[-46])
validation_y = data.matrix(validation[46])
test_x = data.matrix(test[-46])
test_y = data.matrix(test[46])

# Fitting AFNN model in R using the Keras library.

model_A = keras_model_sequential()
model_A %>% layer_dense(units=25, activation="relu",
input_shape=ncol(train_x),
kernel_regularizer=regularizer_l1(0.001),
kernel_initializer="he_normal")
model_A %>% layer_dropout(rate=0.5)
model_A %>% layer_dense(units=16, activation="relu",
kernel_regularizer=regularizer_l1(0.001),
kernel_initializer="he_normal")
model_A %>% layer_dropout(rate=0.5)
model_A %>% layer_dense(units=4, activation="relu",
kernel_regularizer=regularizer_l1(0.001),
kernel_initializer="he_normal")
model_A %>% layer_dropout(rate=0.5)
model_A %>% layer_dense(1, activation="sigmoid")

model_A %>% compile(loss="binary_crossentropy",
optimizer=optimizer_adam(learning_rate=0.0007),
metrics=c("accuracy"))

model_A %>% fit(train_x, train_y, batch_size=32, epochs=20)

pred_prob_A = model_A %>% predict(test_x)
match = c()
pred_prob_class = c()
for (i in 1:nrow(pred_prob_A)) {
    pred_prob_class[i] = ifelse(pred_prob_A[i]>0.5,1,0)
}
combined = cbind(test_y, pred_prob_class)
for (i in 1:nrow(combined)) {
    match[i] = ifelse(combined[i,1]==combined[i,2],1,0)
}
accuracy = sum(match)/nrow(combined)
print(round(accuracy, 4))
```

10.4 Appendix D: SAS Implementation for Artificial Neural Networks

```
data accuracy
set tmp1.em_save_test;
match=(em_classification=em_classtarget);
run;

proc sql;
select mean(match) as accuracy
from accuracy;
quit;
```

11 References

Aurelion, Geron. Hands-On Machine Learning with SciKit-Learn and Tensorflow: Concepts, Tools, and Techniques to Build Intelligent Systems. Accessed April 8, 2025.

Keepnet Solutions. *2025 Phishing Statistics: Top Phishing Statistics and Trends You Must Know in 2025.* Accessed April 8, 2025.

Korosteleva, Olga. STAT 574: Data Mining Lecture Notes. *Artificial Neural Networks.* Accessed April 8, 2025.

Lee, Seungjoon. STAT 576: Data Informatics Lecture Notes. *Unsupervised and Semi-Supervised Learning.* Accessed April 20, 2025.

Moon, Hojin. STAT 479: Applied Neural Networks and Deep Learning Lecture Notes. *Feed-Forward Neural Networks and Activation Functions.* Accessed April 8, 2025.

Prasad, A., & Chandra, S. (2023). *PhiUSIIL: A Diverse Security Profile Empowered Phishing URL Detection Framework Based on Similarity Index and Incremental Learning. Computers & Security*, 103545. DOI: https://doi.org/10.1016/j.cose.2023.103545.