TABLE OF CONTENTS

Title

The prediction and prevention of phishing attacks by the classification of webpages into fraudulent (phishing) and legitimate categories using classification algorithms.

Introduction

Computers are the driving force responsible for the advancement of technology in the world today, and with their constantly increasing use comes the risk of cybercrimes. Ranking high among these cybercrimes is the scourge that is Phishing (Financial and cybercrimes top global police concerns, says new Interpol report, 2022)

Phishing is the attempt to manipulate a user into performing an action that leads to unwanted consequences such as downloading malicious software (malware) or redirection to a questionable website. (Phishing attacks: Defending your organisation, 2018)

While performing everyday online activities, different website links/webpages are encountered, and the ability to distinguish between phishing and authentic websites cannot be overemphasized.

Machine learning classification systems can help make the process of correctly identifying these websites automatic, and this can be implemented in the design of softwares that automatically predicts and prevents this cyber attack.

This report seeks to evaluate and compare the performance of models, fabricated using 2 classification algorithms (K-Nearest Neighbors and Neural Networks) on the dataset provided in the following section. These models are designed to classify websites into either phishing or legitimate categories.

Datasets

The dataset was published on the 24th of March, 2018 by Choon Lin Tan, and is titled, "Phishing Dataset for Machine Learning: Feature Evaluation", and was downloaded from https://data.mendeley.com/datasets/h3cgnj8hft/1

The data has 10,000 observations, which consist of 50 attributes taken from 5000 authentic websites (from Alexa and Common Crawl) and 5000 fraudulent websites (from PhishTank and OpenPhish) between January & May 2015 and May & June 2017.

The contributor, while obtaining the data, utilised the Selenium WebDriver, which is a browser automation framework, which provides improved webpage feature extraction techniques that delivers more accurate and reliable results than a parsing approach based on regular expressions. (Tan, 2018).

The dataset was downloaded in .arff format and was converted to .csv through the use of an open-source Github online converter.

In the class column (CLASS_LABEL), the phishing websites are represented by 1, the legitimate websites are represented by 0

Explanation and preparation of datasets

The dataset was downloaded and read into a variable called 'dataset', using Pandas,

The dataset contains 50 attributes. The first attribute represents the Id, which will later be removed, the following 48 attributes are various features of the website in each observation, and the last attribute represents the class label, which is eventually renamed 'Result'. See below

```
In [120]: column_list = dataset.columns.values.tolist()
          i_v = pd.DataFrame({'Independent Variables':column_list[:-1]})
          i_v.index+=1
```

	Independent Variables
1	id
2	NumDots
3	SubdomainLevel
4	PathLevel
5	UrlLength
6	NumDash
7	NumDashInHostname
8	AtSymbol
9	TildeSymbol
10	NumUnderscore
11	NumPercent
12	NumQueryComponents
13	NumAmpersand
14	NumHash
15	NumNumericChars
16	NoHttps
17	RandomString
18	IpAddress
19	DomainInSubdomains
20	DomainInPaths
21	HttpsInHostname
22	HostnameLength
23	PathLength
24	QueryLength
25	DoubleSlashInPath
26	NumSensitiveWords
27	EmbeddedBrandName
28	PctExtHyperlinks
29	PctExtResourceUrls
30	ExtFavicon
31	InsecureForms
32	RelativeFormAction
33	ExtFormAction
34	AbnormalFormAction
35	PctNullSelfRedirectHyperlinks
36	FrequentDomainNameMismatch
37	FakeLinkInStatusBar
38	RightClickDisabled
39	PopUpWindow
40	SubmitInfoToEmail
41	IframeOrFrame
42	MissingTitle
43	ImagesOnlyInForm
44	SubdomainLevelRT
45	UrlLengthRT
46	PctExtResourceUrlsRT
47	AbnormalExtFormActionR
48	ExtMetaScriptLinkRT
49	PctExtNullSelfRedirectHyperlinksRT

Dependent variable:

```
In [50]: column_list = dataset.columns.values.tolist()
         v = pd.DataFrame({'Dependent Variable':column_list[-1:]})
         v.index+=1
         v
```

Out[50]:

	Dependent Variable
1	CLASS_LABEL

Several preprocessing steps were followed.

The shape of the dataset was checked to see the number of records and attributes, then the '.head' function was used for a glance. See below

```
In [483]: dataset = pd.read_csv("C:\\Users\C\Downloads\Phishing_Legitimate_full.csv")

In [484]: dataset.shape #to check the number of records and attributes
Out[484]: (10000, 50)

In [485]: dataset.head()
Out[485]:
```

	id	NumDots	SubdomainLevel	PathLevel	UrlLength	NumDash	NumDashInHostname	AtSymbol	TildeSymbol	NumUnderscore	...	IframeOrFrame	MissingT
0	1	3	1	5	72	0	0	0	0	0	...	0	
1	2	3	1	3	144	0	0	0	0	2	...	0	
2	3	3	1	2	58	0	0	0	0	0	...	0	
3	4	3	1	6	79	1	0	0	0	0	...	0	
4	5	3	0	4	46	0	0	0	0	0	...	1	

5 rows × 50 columns

Next, the '.info' function was used as a quick means to check for missing values and data type, and the results show that all columns 'Non-Null Count' are 10,000 which matches the number of rows in the shape shown above, meaning there are no missing values.

The 'Dtype' result also shows that all columns are either float(float 64) or integer(int64) datatype. The absence of the dtype 'object' signifies there are no categorical variables. See below

```
In [486]: dataset.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 50 columns):
 #   Column                              Non-Null Count  Dtype
---  ------                              --------------  -----
 0   id                                  10000 non-null  int64
 1   NumDots                             10000 non-null  int64
 2   SubdomainLevel                      10000 non-null  int64
 3   PathLevel                           10000 non-null  int64
 4   UrlLength                           10000 non-null  int64
 5   NumDash                             10000 non-null  int64
 6   NumDashInHostname                   10000 non-null  int64
 7   AtSymbol                            10000 non-null  int64
 8   TildeSymbol                         10000 non-null  int64
 9   NumUnderscore                       10000 non-null  int64
 10  NumPercent                          10000 non-null  int64
 11  NumQueryComponents                  10000 non-null  int64
 12  NumAmpersand                        10000 non-null  int64
 13  NumHash                             10000 non-null  int64

In [487]: #From the result above, it can be seen that the dataset has no missing values since the non-null Count is equal to the number of
```

```
In [486]: dataset.info()
 33  AbnormalFormAction                  10000 non-null  int64
 34  PctNullSelfRedirectHyperlinks       10000 non-null  float64
 35  FrequentDomainNameMismatch          10000 non-null  int64
 36  FakeLinkInStatusBar                 10000 non-null  int64
 37  RightClickDisabled                  10000 non-null  int64
 38  PopUpWindow                         10000 non-null  int64
 39  SubmitInfoToEmail                   10000 non-null  int64
 40  IframeOrFrame                       10000 non-null  int64
 41  MissingTitle                        10000 non-null  int64
 42  ImagesOnlyInForm                    10000 non-null  int64
 43  SubdomainLevelRT                    10000 non-null  int64
 44  UrlLengthRT                         10000 non-null  int64
 45  PctExtResourceUrlsRT                10000 non-null  int64
 46  AbnormalExtFormActionR              10000 non-null  int64
 47  ExtMetaScriptLinkRT                 10000 non-null  int64
 48  PctExtNullSelfRedirectHyperlinksRT  10000 non-null  int64
 49  CLASS_LABEL                         10000 non-null  int64
dtypes: float64(3), int64(47)
memory usage: 3.8 MB

In [487]: #From the result above, it can be seen that the dataset has no missing values since the non-null Count is equal to the number of
```

Exploratory Data Analysis

In this stage, the '.describe' and '.value_counts' function was employed to have a glimpse into the distribution of the X-columns (features to be used to define the class) values, and the Y-variable (the class). See below

```
In [488]: dataset.describe()
```

Out[488]:

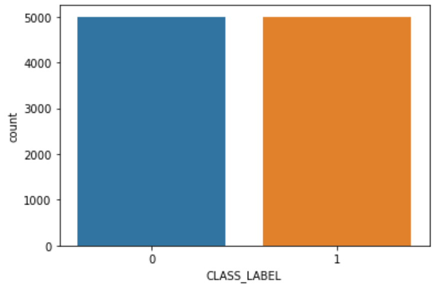
	id	NumDots	SubdomainLevel	PathLevel	UrlLength	NumDash	NumDashInHostname	AtSymbol	TildeSymbol	NumUndersco
count	10000.00000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.000
mean	5000.50000	2.445100	0.586800	3.300300	70.264100	1.818000	0.138900	0.000300	0.013100	0.323
std	2886.89568	1.346836	0.751214	1.863241	33.369877	3.106258	0.545744	0.017319	0.113709	1.114
min	1.00000	1.000000	0.000000	0.000000	12.000000	0.000000	0.000000	0.000000	0.000000	0.000
25%	2500.75000	2.000000	0.000000	2.000000	48.000000	0.000000	0.000000	0.000000	0.000000	0.000
50%	5000.50000	2.000000	1.000000	3.000000	62.000000	0.000000	0.000000	0.000000	0.000000	0.000
75%	7500.25000	3.000000	1.000000	4.000000	84.000000	2.000000	0.000000	0.000000	0.000000	0.000
max	10000.00000	21.000000	14.000000	18.000000	253.000000	55.000000	9.000000	1.000000	1.000000	18.000

8 rows × 50 columns

```
In [489]: #Doing some exploration on the Y-variable
          dataset.CLASS_LABEL.value_counts()

Out[489]: 1    5000
          0    5000
          Name: CLASS_LABEL, dtype: int64
```

The graph below shows that the Y-variable is evenly distributed:

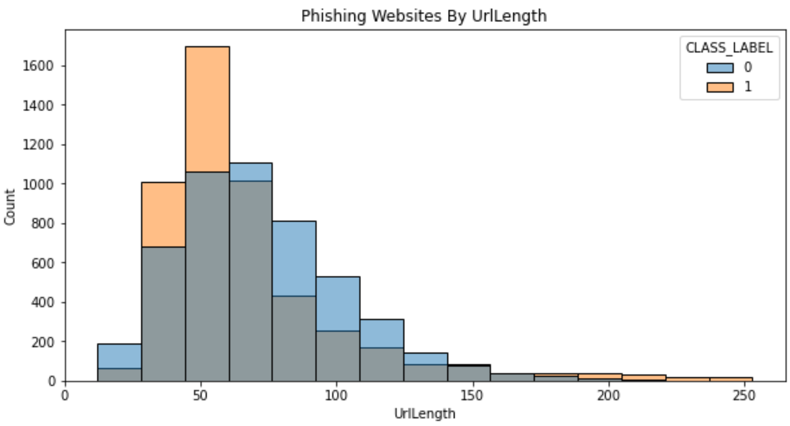
```
In [490]: #To view the Class_Label distribution
          sns.countplot(data=dataset, x='CLASS_LABEL')

Out[490]: <AxesSubplot:xlabel='CLASS_LABEL', ylabel='count'>
```




```
In [491]: #From the above result, it can be immediately seen that the Class Label is split evenly between legitimate and Phishing webpages
```

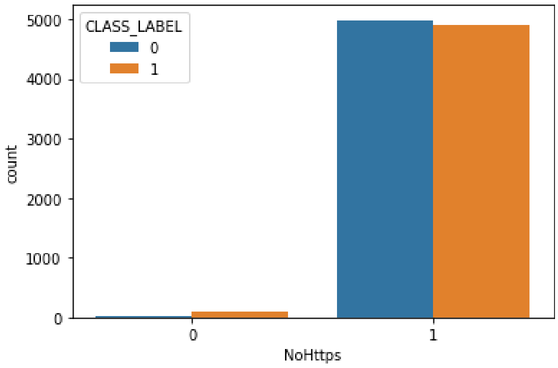
Next, some specific attributes were considered to check their distribution and proportion in the class label. Below:

```
In [492]: plt.figure(figsize=(10,5))
          plt.title('Phishing Websites By UrlLength')
          sns.histplot(data=dataset, hue='CLASS_LABEL', x='UrlLength', bins=15)
          plt.show()
```



From the distribution in the image above, pages with URL lengths of around 50 have the highest number of phishing links.

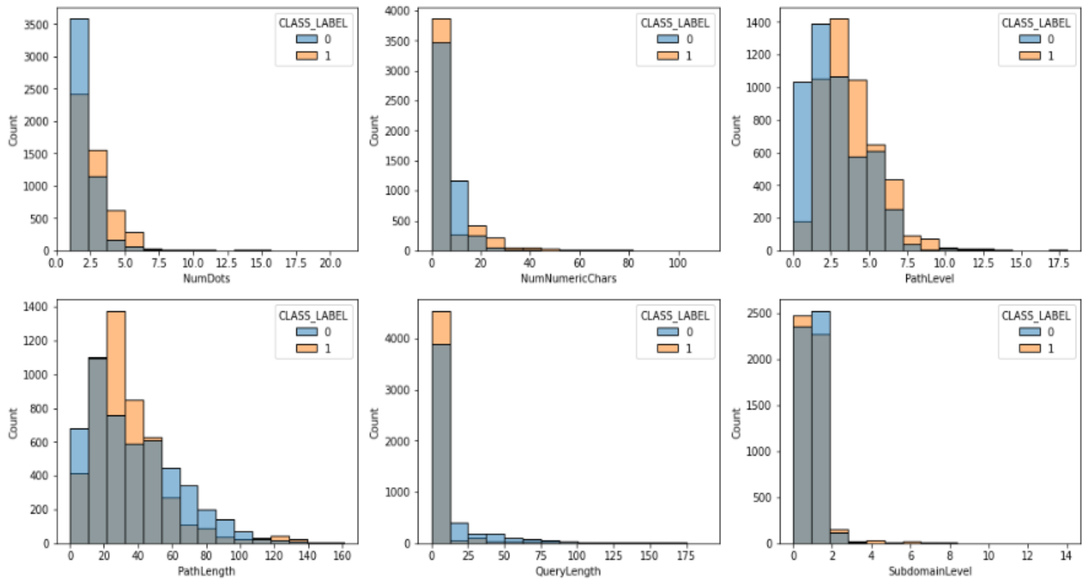
```
In [493]: #To see how the NoHttps attribute is distributed and its proportion in the CLASS_LABEL
          sns.countplot(data=dataset, x='NoHttps', hue='CLASS_LABEL')

Out[493]: <AxesSubplot:xlabel='NoHttps', ylabel='count'>
```



Next, a function to show the histogram distribution between selected variables and the class label. See below:

```
#Defining a function to show me the histogram and boxplot views of some selected Variables
plot_Var = ['NumDots', 'NumNumericChars', 'PathLevel', 'PathLength', 'QueryLength', 'SubdomainLevel']
def chart_plot(plot_type):
    if plot_type=='histogram':
        plt.figure(figsize=(18,9))
        for i in range(len(plot_Var)):
            plt.subplot(2,3,i+1)
            sns.histplot(data=dataset, hue='CLASS_LABEL', x=plot_Var[i], bins=15)
    elif plot_type=='boxplot':
        plt.figure(figsize=(18,9))
        for i in range(len(plot_Var)):
            plt.subplot(2,3,i+1)
            sns.boxplot(data=dataset, hue='CLASS_LABEL', x=plot_Var[i], showmeans=True)
    else:
        pass
```

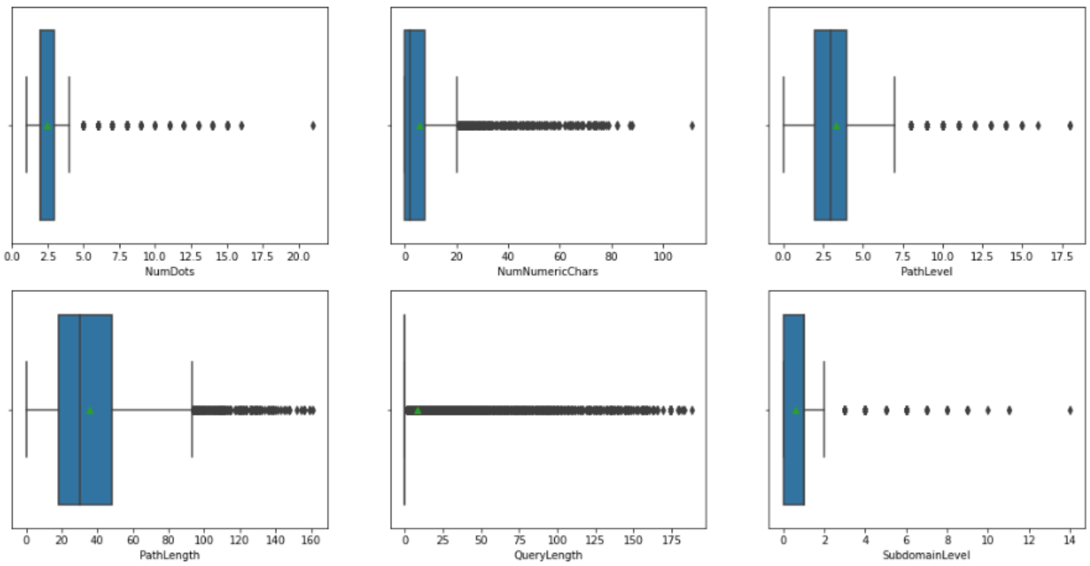
```
In [495]: #To quickly examine the distribution between some variables and the Class Label.
chart_plot('histogram')
```



The results above show that the distribution of most of the variables above is concentrated towards the lower values.

The boxplots below reveal the outliers in the above-selected variables.

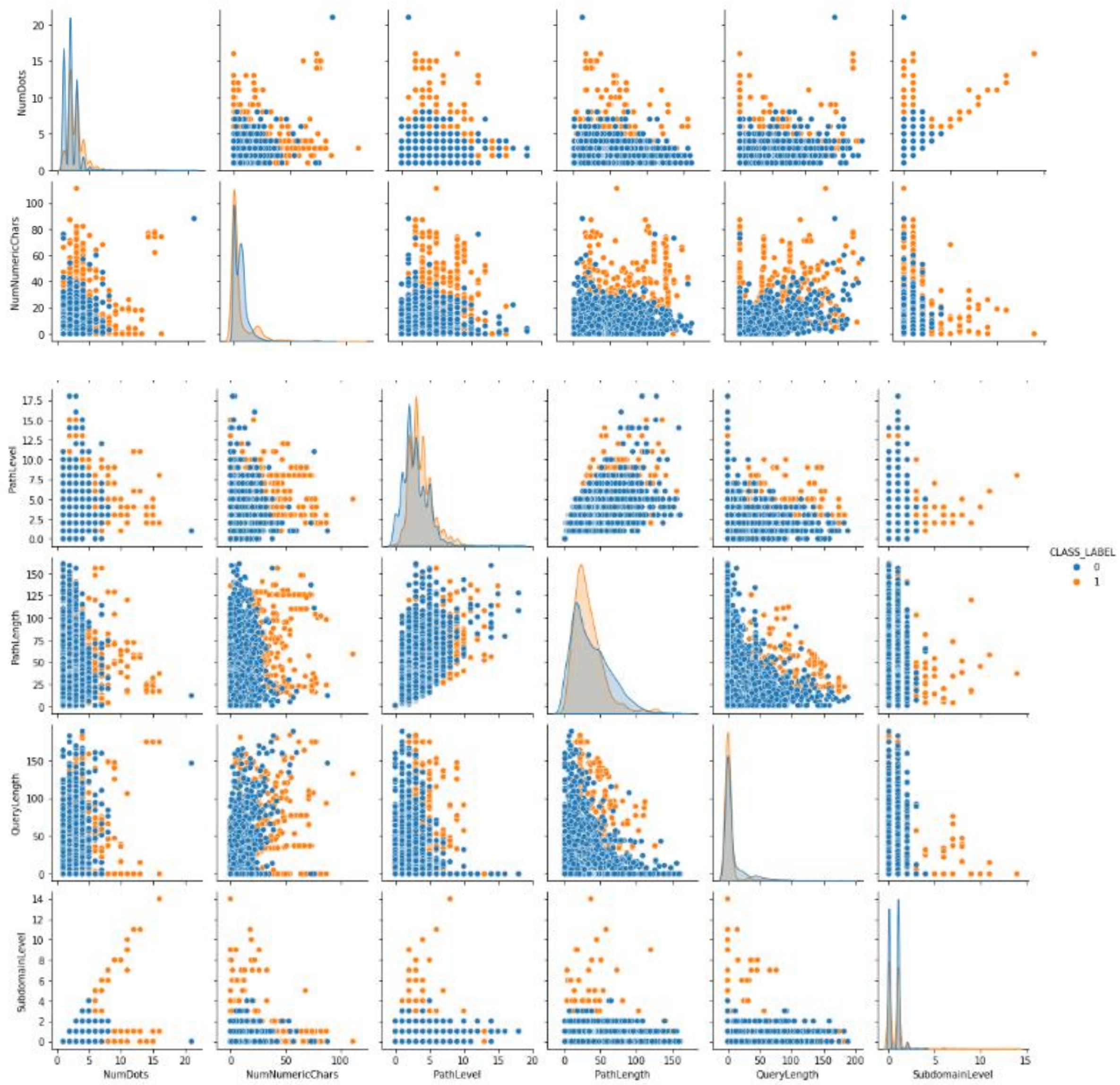
```
In [496]: #To have a quick view on outlier values using a boxplot
chart_plot('boxplot')
```



To get a view of how the variables are related to themselves and the class label, we plot a scatterplot matrix. The diagonal in the matrix shows the density plot of the label. View the code & result below:

```
In [497]: #To get the relationship between the above selected variables and the Class Label, we plot a scatterplot matrix
plt.figure(figsize=(15,10))
sns.pairplot(dataset[['NumDots', 'NumNumericChars', 'PathLevel', 'PathLength', 'QueryLength', 'SubdomainLevel', 'CLASS_LABEL']],
            hue='CLASS_LABEL')
plt.show()
```

```
<Figure size 1080x720 with 0 Axes>
```



Feature Improvement

As shown earlier in **insert section here**, this dataset does not contain categorical variables, therefore no encoding using LabelEncoder, One-hot Encoder, etc is necessary.

Also, there are no missing values. As a result, there was no need to employ SimpleImputer to impute missing values

The 'Id' column was dropped since it doesn't affect the classification and the class label was renamed to 'Result'.

Feature Improvement

In [498]:
```
dataset1 = dataset.drop('id', axis=1) #Drop the ID column (since it is not an attribute to be considered in the classification)
dataset1 = dataset1.rename(columns={'CLASS_LABEL':'Result'}) #Renaming the CLASS_LABEL column
dataset1.head()
```

Out[498]:

	NumDots	SubdomainLevel	PathLevel	UrlLength	NumDash	NumDashInHostname	AtSymbol	TildeSymbol	NumUnderscore	NumPercent	...	IframeOrFrame
0	3	1	5	72	0	0	0	0	0	0	...	0
1	3	1	3	144	0	0	0	0	2	0	...	0
2	3	1	2	58	0	0	0	0	0	0	...	0
3	3	1	6	79	1	0	0	0	0	0	...	0
4	3	0	4	46	0	0	0	0	0	0	...	1

5 rows × 49 columns

Implementation in Python

Brief description of the algorithms used

K-Nearest Neighbors (KNN) algorithm

The KNN algorithm is a python classification algorithm that works using a simple idea but provides great results. Usually, the algorithm uses the Euclidean distance, d, to find the distance between 2 points (x_1, y_1) and (x_2, y_2).

$d = \sqrt{[(x_2 - x_1)^2 + (y_2 - y_1)^2]}$

To make a classification, the system uses observations from a training set, calculates the Euclidean distance between them, gets the k nearest neighbors (where k is the number of neighbors chosen), and uses the attributes of the neighbors to make a classification. (Baoli, Shiwen, & Qin, 2003)

Artificial Neural Networks

ANN classification is modeled loosely on the functionality of the human brain. ANN consists of processing nodes that are interconnected densely. These nodes are separated into layers called input, hidden, and output layers, and work by assigning weights to the input signals attributes, processing them, and then passing them on to other nodes in the next tier to eventually establish a classification. (Hardesty, 2017)

Applications of the Algorithms

K-Nearest Neighbors Classification

Application of the KNN algorithm to the dataset & Model design

First, the data columns were split into **X** and **y** representing the **Features** and the **Class** respectively.

Then, the datasets were split into training and testing data for both the X and the y parts using the train_test_split function from sklearn.model_selection.

The dataset was split into a 70:30 training-testing ratio, 'random_state' integer was specified so that the same train and test sets are gotten across different executions.

Next, the StandardScaler function was used to fit and transform the X training and just transform the X testing dataset (to apply the initial fitting parameters to the test dataset).

Conducting the fit_transform statistically means centering the data by subtracting the mean and dividing by the standard deviation.

$X' = (X - \mu)/\sigma$

n-jobs was set to -1 to activate parallel processing for faster run-times

k-Nearest Neighbors (KNN) Classification

```
In [75]: #Split the data columns into arrays containing attributes and another containing results
         X = dataset1.iloc[:, 0: len(dataset1.columns.tolist())-1]
         y = dataset1.iloc[:,-1]
```

```
In [76]: from sklearn.model_selection import train_test_split, GridSearchCV
         X_train, X_test, y_train, y_test = train_test_split(
             X, y, test_size=0.3, random_state=42, stratify=y) #Stratify keeps the split percentage constant in both training and testing
```

```
In [77]: #To fit_transform the train values and transform the test values
         sc=StandardScaler()
         X_train_s=sc.fit_transform(X_train) #This returns a numpy array, and that is because numpy arrays are faster to process than dat
         X_test_s=sc.transform(X_test)
```

```
In [78]: #Using the KNN classifier
         from sklearn.neighbors import KNeighborsClassifier
         class_model = KNeighborsClassifier(n_neighbors=3, metric='minkowski',p=2, n_jobs=-1) #n_jobs =-1 maximizes CPU capacity to make
         class_model.fit(X_train_s, y_train)

Out[78]: KNeighborsClassifier(n_jobs=-1, n_neighbors=3)
```

Experimental procedure

For the model, as seen in the figure above, the initial parameters used for the first experiment are:

N_neighbors = 3
Metric = Minkowski
P=2

The approach to validation used is the Hold-out method. This method was chosen because it is ideal for when there is a relatively large amount of data.

Result Visualization:

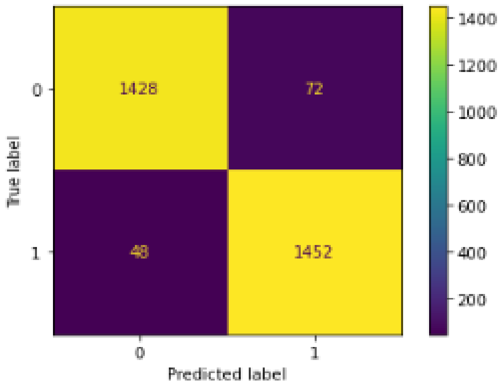
The model was deployed on the test data. The code & results are as shown:

Evaluating the Model Performance

```
In [79]: y_pred = class_model.predict(X_test_s)
         print(y_pred)

         [1 1 0 ... 0 1 1]
```

```
In [80]: acc=accuracy_score(y_test,y_pred)
         print('accuracy:%.2f\n\n'%(acc))
         cm=confusion_matrix(y_test,y_pred)
         print('Confusion Matrix')
         print(cm,'\n\n')
         print('=======================================================')
         result=classification_report(y_test,y_pred)
         print('Classification Report:')
         print('-------------------------------------------------------')
         print(result)
         disp=ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_model.classes_)
         disp.plot()
         plt.show()
```

```
accuracy:0.96


Confusion Matrix
[[1428   72]
 [  48 1452]]


=======================================================
Classification Report:
-------------------------------------------------------
              precision    recall  f1-score   support

           0       0.97      0.95      0.96      1500
           1       0.95      0.97      0.96      1500

    accuracy                           0.96      3000
   macro avg       0.96      0.96      0.96      3000
weighted avg       0.96      0.96      0.96      3000
```



To set and optimize hyper-parameters, the use of GridSearchCV was employed, and a graph of the various k-values against their corresponding accuracy is shown below:

In [81]:
```
knn2=KNeighborsClassifier()
hyperparameters={'n_neighbors': (1,12,1), 'metric': ('minkowski', 'chebyshev')}

knn_cv = GridSearchCV(knn2, hyperparameters, n_jobs=-1, verbose=1)
knn_cv.fit(X_train_s, y_train)
```

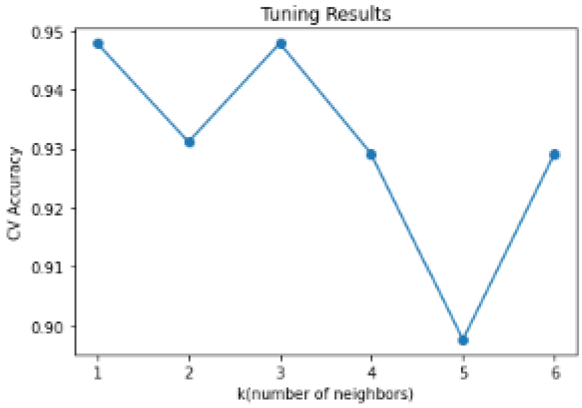
```
Fitting 5 folds for each of 6 candidates, totalling 30 fits
```

Out[81]:
```
GridSearchCV(estimator=KNeighborsClassifier(), n_jobs=-1,
             param_grid={'metric': ('minkowski', 'chebyshev'),
                         'n_neighbors': (1, 12, 1)},
             verbose=1)
```

In [82]:
```
knn_cv.best_params_
```

Out[82]:
```
{'metric': 'minkowski', 'n_neighbors': 1}
```

In [83]:
```
tuning_result=knn_cv.cv_results_['mean_test_score']
plt.plot(range(1,7), tuning_result, 'o-')
plt.ylabel('CV Accuracy')
plt.xlabel('k(number of neighbors)')
plt.title('Tuning Results')
plt.show()
```



From the figure above, the best parameters are:

Metric: Minkowski

k: 1

The graph above and the tuning result below show that k=1 and k=3 yield the same accuracy score.

In [84]:
```
tuning_result
```

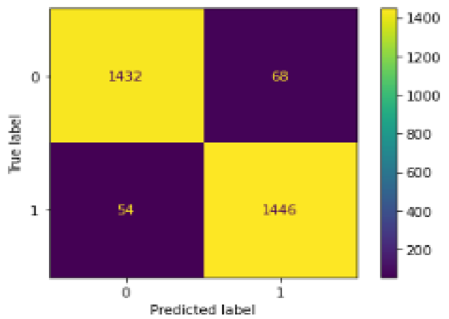
Out[84]:
```
array([0.94785714, 0.93114286, 0.94785714, 0.92914286, 0.89771429,
       0.92914286])
```

Next experiment using derived parameters k=1

<u>Results Visualization</u> below:

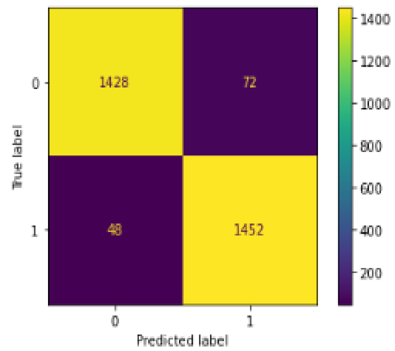
In [85]:
```
predict_best = knn_cv.best_estimator_.predict(X_test_s)
print(classification_report(y_test, predict_best))
print("==============================================")
print("Estimated Value Confusion Matrix\n**********************************************")
cm2 = confusion_matrix(y_test, predict_best)
disp=ConfusionMatrixDisplay(confusion_matrix=cm2, display_labels=class_model.classes_)
disp.plot()
plt.show()
```

```
              precision    recall  f1-score   support

           0       0.96      0.95      0.96      1500
           1       0.96      0.96      0.96      1500

    accuracy                           0.96      3000
   macro avg       0.96      0.96      0.96      3000
weighted avg       0.96      0.96      0.96      3000

==============================================
Estimated Value Confusion Matrix
**********************************************
```

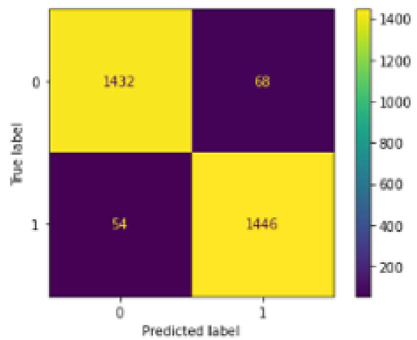


Comparing both results (k=3, k=1) side by side:

```
================================================
Classification Report:
------------------------------------------------
              precision    recall  f1-score   support

           0       0.97      0.95      0.96      1500
           1       0.95      0.97      0.96      1500

    accuracy                           0.96      3000
   macro avg       0.96      0.96      0.96      3000
weighted avg       0.96      0.96      0.96      3000
```




```
              precision    recall  f1-score   support

           0       0.96      0.95      0.96      1500
           1       0.96      0.96      0.96      1500

    accuracy                           0.96      3000
   macro avg       0.96      0.96      0.96      3000
weighted avg       0.96      0.96      0.96      3000

==============================================
Estimated Value Confusion Matrix
**********************************************
```



From the classification reports, both results are evenly matched. K=1 is finally chosen as it has a more balanced overall score.

Relevant Literature

In the journal article, “A Review of Data Classification Using K-Nearest Neighbour Algorithm” by Aman Kataria, M. D. Singh, the contributors alluded to the fact that even though the KNN model is efficient, it is limited by great calculation complexity, cost, operating speed, and complete dependence on the training set. The results of the experiment align with this allusion. (Kataria & Singh, 2013)

Artificial Neural Networks (ANNs) classification

Application of the ANN algorithm to the dataset

Since the process of splitting, fitting, and transforming the dataset is the same as the algorithm used above, the variables holding the features part of the training and testing data were copied.

```
In [86]: X_train_NN = X_train_s.copy()
         X_test_NN = X_test_s.copy()
```

Explanation of the experimental procedure

First, the number of features (columns) was taken to estimate the number of features for the input layer:

```
In [87]: X_train_NN.shape[1] #To get the number of features for the input Layer

Out[87]: 48
```

Next, a set_seed function was defined to make sure the results are reproducible each time the model is run. Random seeds were taken from Numpy, Python, and Tensorflow:

```
In [88]: #This step is to make sure the results can be reproduced and do not change each time the model is run
         def set_seed(seed=20):
             np.random.seed(seed) #to seed from everything in numpy
             random.seed(seed) #to seed from everything in Python itself
             tf.random.set_seed(seed) #seeds randomness from anything in tensorflow
```

The Sequential API in Keras was used as the model is creating layers in sequence i.e input – Hidden – Output layers and no branching/sub-classes are required:

Next, the input shape of the hidden layer was specified, and other layers' shapes are generated automatically. ‘Dense’ is used because all neurons from a preceding layer feed into each neuron of the next layer. The activation function used is ‘ReLU’, and an arbitrary number of neurons (30) was selected.

In the Output layer, the Sigmoid activation function was employed as we only have 2 mutually exclusive classes.

```
In [89]: set_seed()
         model = tf.keras.models.Sequential() #The Sequential here is the type of API we're using
         model.add(tf.keras.layers.Dense(30,activation='relu',input_shape=(X_train_NN.shape[1],))) #Only the input shape needs to be spec
         model.add(tf.keras.layers.Dense(1,activation='sigmoid')) #Sigmoid is used for binary classifications where we only have 2 mutual
         #when writing report, check how to determine the optimum number of input neurons ("20" in Line 2)
         model.summary()
         #The number of parameters is gotten by multiplying the number of hidden Layer nodes by the number of input features + bias (1 pe

Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_2 (Dense)             (None, 30)                1470

 dense_3 (Dense)             (None, 1)                 31

=================================================================
Total params: 1,501
Trainable params: 1,501
Non-trainable params: 0
_________________________________________________________________
```

<u>Model Parameters Used</u>

Optimizer = 'adam'. This was used as it selects a good rate at which the weights are updated

Loss = 'binary_crossentropy' This is because only 2 output classes are expected

Metrics = 'accuracy'. This was used because, as shown earlier, the output classes in the dataset are not imbalanced

Class weight wasn't specified as the output class is balanced.

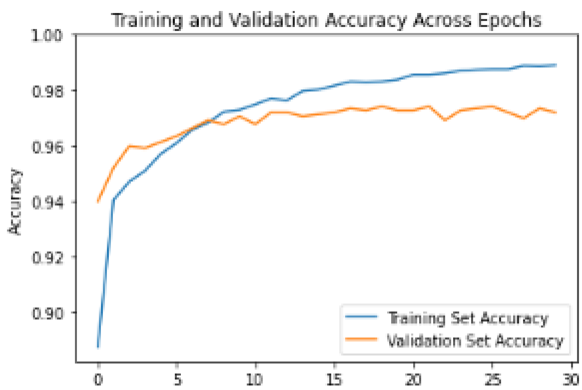
```
In [90]: model.compile(optimizer='adam',
                       loss='binary_crossentropy',
                       metrics='accuracy') #binary_crossentropy Because there are only 2 classes, and accuracy metrics because my class i
```

```
In [91]: history = model.fit(X_train_NN, y_train,
                             batch_size = 10,
                             epochs= 30,
                             verbose=2,
                             validation_split=0.2) #didn't specify class weight because my class is balanced

         Epoch 1/30
         560/560 - 4s - loss: 0.3010 - accuracy: 0.8875 - val_loss: 0.1710 - val_accuracy: 0.9400 - 4s/epoch - 7ms/step
         Epoch 2/30
         560/560 - 1s - loss: 0.1635 - accuracy: 0.9405 - val_loss: 0.1328 - val_accuracy: 0.9521 - 1s/epoch - 2ms/step
         Epoch 3/30
         560/560 - 1s - loss: 0.1381 - accuracy: 0.9471 - val_loss: 0.1183 - val_accuracy: 0.9600 - 985ms/epoch - 2ms/step
         Epoch 4/30
         560/560 - 1s - loss: 0.1240 - accuracy: 0.9511 - val_loss: 0.1110 - val_accuracy: 0.9593 - 960ms/epoch - 2ms/step
         Epoch 5/30
         560/560 - 1s - loss: 0.1106 - accuracy: 0.9571 - val_loss: 0.0990 - val_accuracy: 0.9614 - 965ms/epoch - 2ms/step
         Epoch 6/30
         560/560 - 1s - loss: 0.1009 - accuracy: 0.9611 - val_loss: 0.0942 - val_accuracy: 0.9636 - 969ms/epoch - 2ms/step
         Epoch 7/30
         560/560 - 1s - loss: 0.0915 - accuracy: 0.9659 - val_loss: 0.0891 - val_accuracy: 0.9664 - 982ms/epoch - 2ms/step
         Epoch 8/30
         560/560 - 1s - loss: 0.0847 - accuracy: 0.9684 - val_loss: 0.0848 - val_accuracy: 0.9693 - 982ms/epoch - 2ms/step
         Epoch 9/30
         560/560 - 1s - loss: 0.0781 - accuracy: 0.9723 - val_loss: 0.0829 - val_accuracy: 0.9679 - 954ms/epoch - 2ms/step
         Epoch 10/30
         560/560 - 1s - loss: 0.0735 - accuracy: 0.9730 - val_loss: 0.0808 - val_accuracy: 0.9707 - 949ms/epoch - 2ms/step
         Epoch 11/30
         560/560 - 1s - loss: 0.0691 - accuracy: 0.9750 - val_loss: 0.0804 - val_accuracy: 0.9679 - 1s/epoch - 2ms/step
         Epoch 12/30
         560/560 - 1s - loss: 0.0646 - accuracy: 0.9771 - val_loss: 0.0799 - val_accuracy: 0.9721 - 1s/epoch - 2ms/step
         Epoch 13/30
         560/560 - 1s - loss: 0.0620 - accuracy: 0.9764 - val_loss: 0.0788 - val_accuracy: 0.9721 - 1s/epoch - 2ms/step
         Epoch 14/30
         560/560 - 1s - loss: 0.0576 - accuracy: 0.9798 - val_loss: 0.0794 - val_accuracy: 0.9707 - 1s/epoch - 2ms/step
```

Evaluating the Neural Network

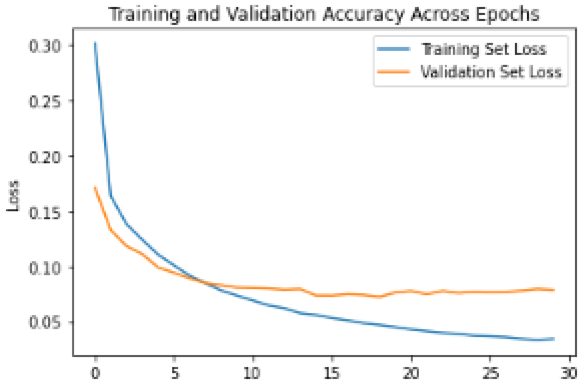
The training and validation accuracy shows an initial steep increase followed by a gradual increase in accuracy for the training set and a fluctuating constant for the validation set. See below:

```
In [92]: accuracy = history.history['accuracy']
         validation_accuracy = history.history['val_accuracy']
         plt.plot(accuracy, label='Training Set Accuracy')
         plt.plot(validation_accuracy, label='Validation Set Accuracy')
         plt.ylabel('Accuracy')
         plt.ylim([min(plt.ylim()),1])
         plt.title('Training and Validation Accuracy Across Epochs')
         plt.legend()

Out[92]: <matplotlib.legend.Legend at 0x25d866c0ee0>
```



The figure below shows an initial steep decline followed by a gradual decline in both the training and accuracy set loss

```
In [93]: loss = history.history['loss']
         validation_loss = history.history['val_loss']
         plt.plot(loss, label='Training Set Loss')
         plt.plot(validation_loss, label='Validation Set Loss')
         plt.ylabel('Loss')
         plt.title('Training and Validation Accuracy Across Epochs')
         plt.legend()

Out[93]: <matplotlib.legend.Legend at 0x25d86712190>
```



The model was used on the test data set and the results were rounded. If 0.5 and above, it returns 1, if less than 0.5, it returns 0. Then the classification results report was printed. See images below

```
In [94]: model.evaluate(X_test_NN, y_test)

         94/94 [==============================] - 0s 2ms/step - loss: 0.0788 - accuracy: 0.9737

Out[94]: [0.07876517623662949, 0.9736666679382324]

In [95]: y_pred = model.predict(X_test_NN)

         94/94 [==============================] - 0s 2ms/step

In [96]: nn_pred = np.round(model.predict(X_test_NN)) #to convert the probability results to binary output where less than 0.5 is 0, great

         94/94 [==============================] - 0s 2ms/step

In [97]: print(classification_report(y_test, nn_pred))

              precision    recall  f1-score   support

           0       0.97      0.97      0.97      1500
           1       0.97      0.97      0.97      1500

    accuracy                           0.97      3000
   macro avg       0.97      0.97      0.97      3000
weighted avg       0.97      0.97      0.97      3000
```

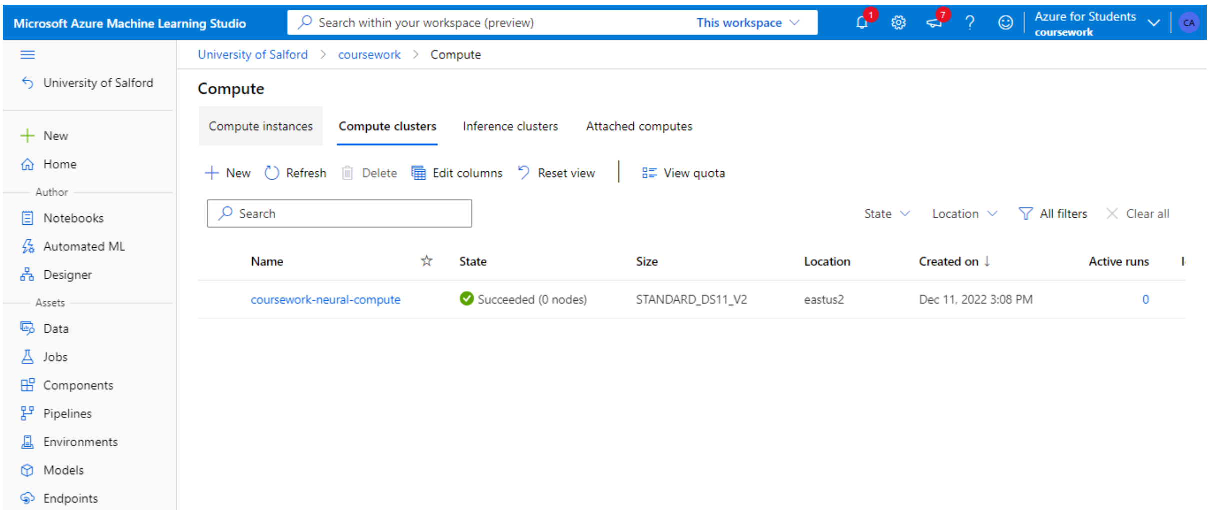
Relevant Literature

In the journal, “Basic Tenets of Classification Algorithms K-Nearest-Neighbor, Support Vector Machine, Random Forest and Neural Network: A Review”, the authors posit that even though Neural Networks are generally more accurate than kNN models, the latter is more commonly used as it is easier to configure and implement. (Boateng, Otoo, & Abaye, 2020)

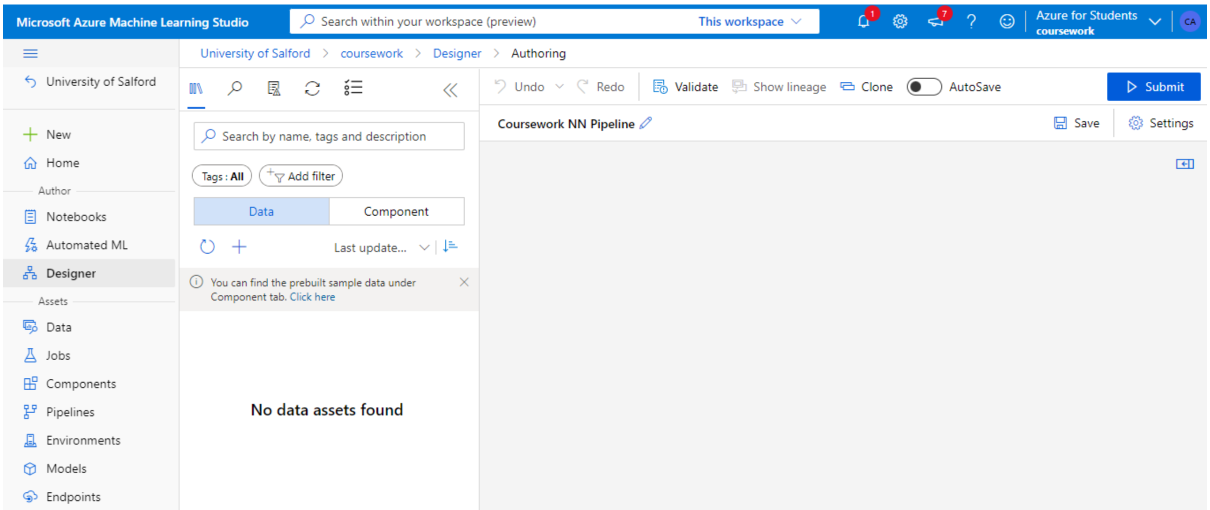
This experiment's results support the author above in showing that the Neural Network has more accuracy than the k-NN model.

Implementation in Azure Machine Learning Studio

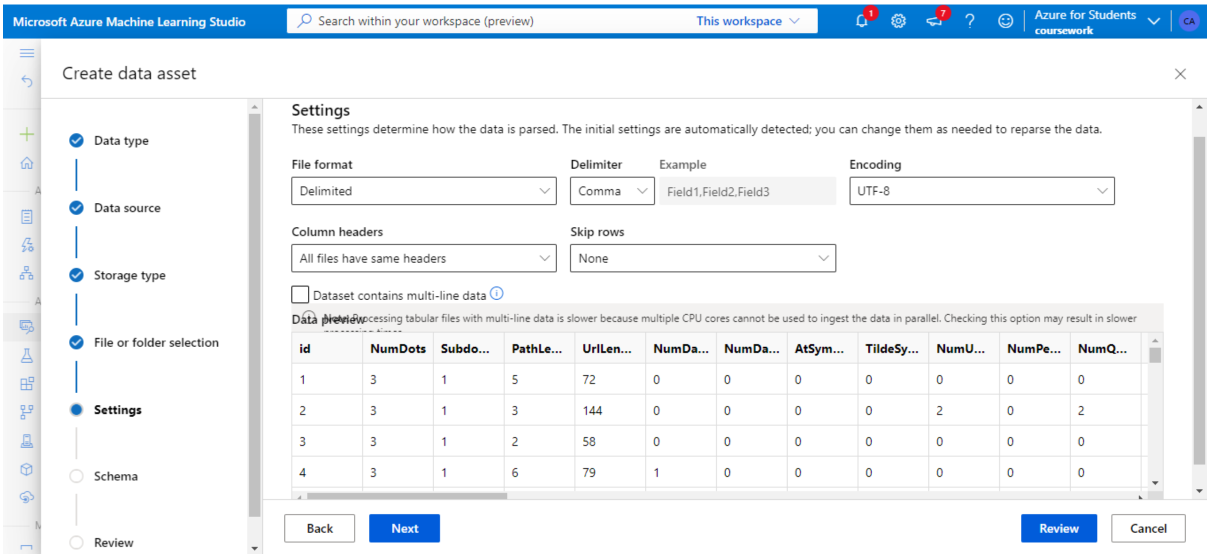
First, the compute cluster was created upon which the implementations would be run.



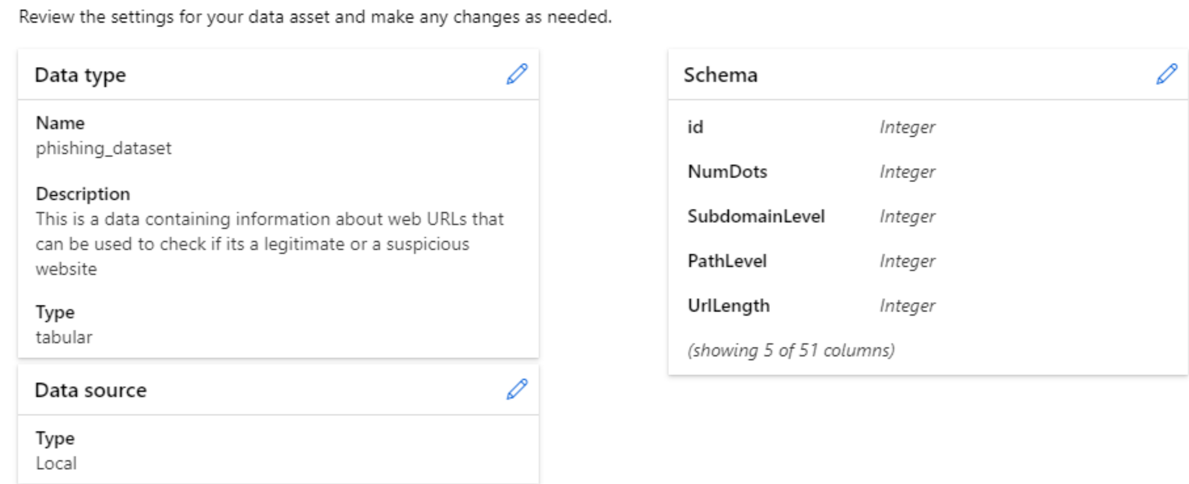
Next, a pipeline was created for the classification task. See below:

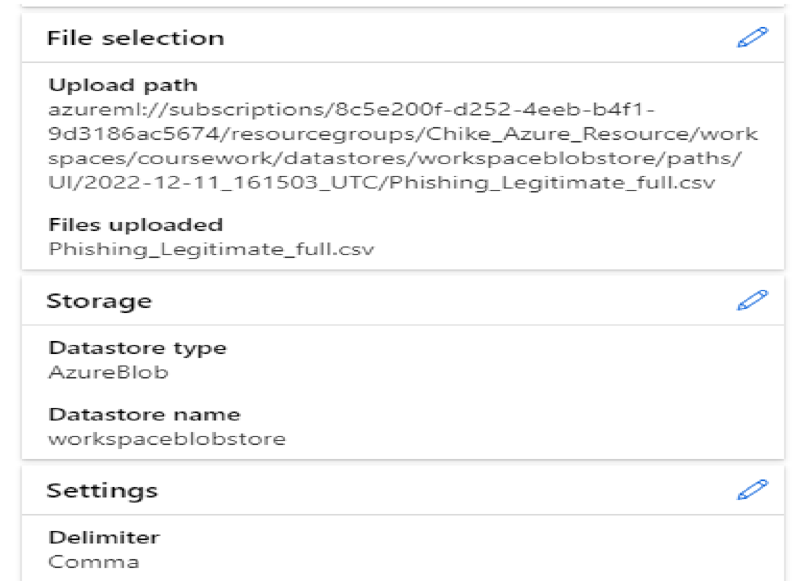


Afterward, a data asset was created by first specifying the Name, description, and type as a table, then the data source was specified as a local file, the storage type was in Azure blob storage, then the data was uploaded, and then a preview of the dataset is shown:

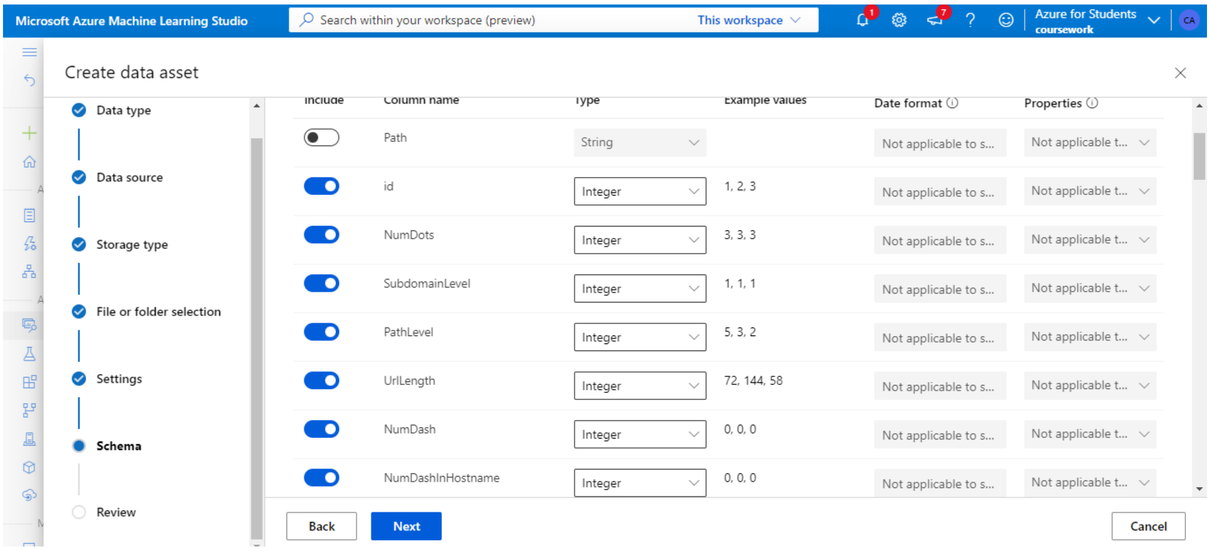


A review of the data was checked before the data was created, and the result is shown below:

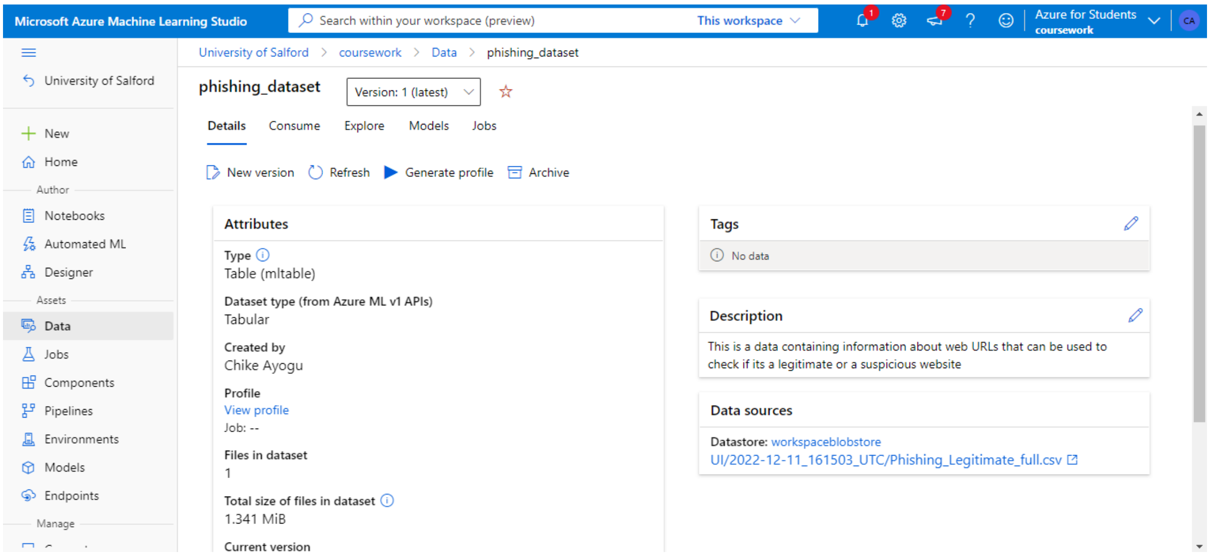




A quick look at the schema also gives some valuable information about the dataset like the datatypes for each column. It was also carefully scrutinized to ensure all datatypes were correct. The top part is shown below:

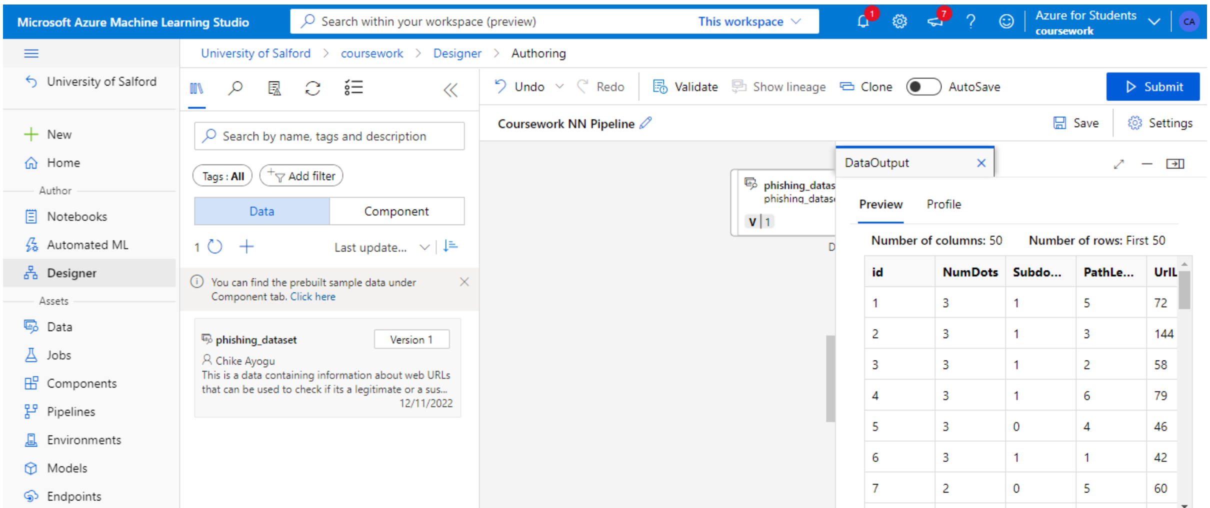


The final data asset details are shown below:



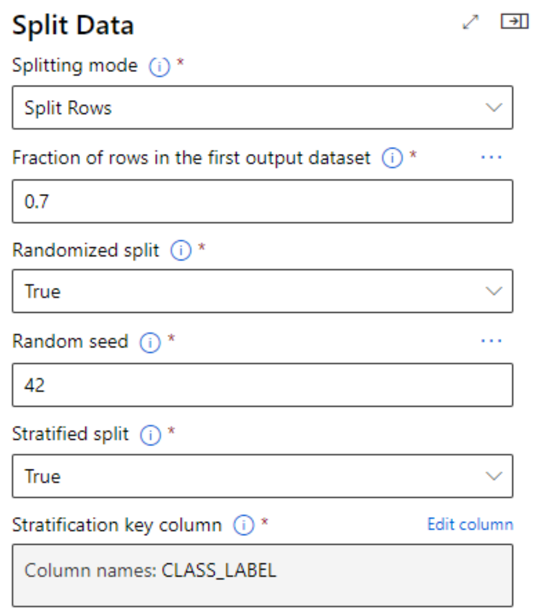
Pipeline Population

The imported dataset was added to the pipeline canvas and was previewed

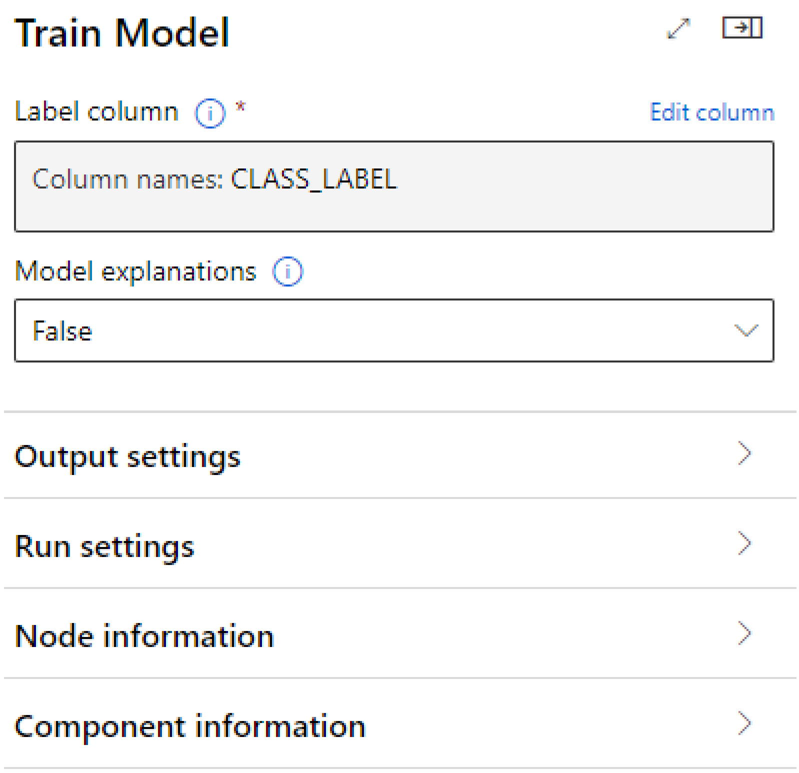


Since there was no missing data in the initial dataset, there was no need to use the 'clean missing data' module.

Next, the 'split Data' module was used to split the data into 70% training and 30% testing. Randomized split was set to True, and the random seed 42, the stratified split was set to true to keep the percentages constant, and the stratified column was set to my target variable.

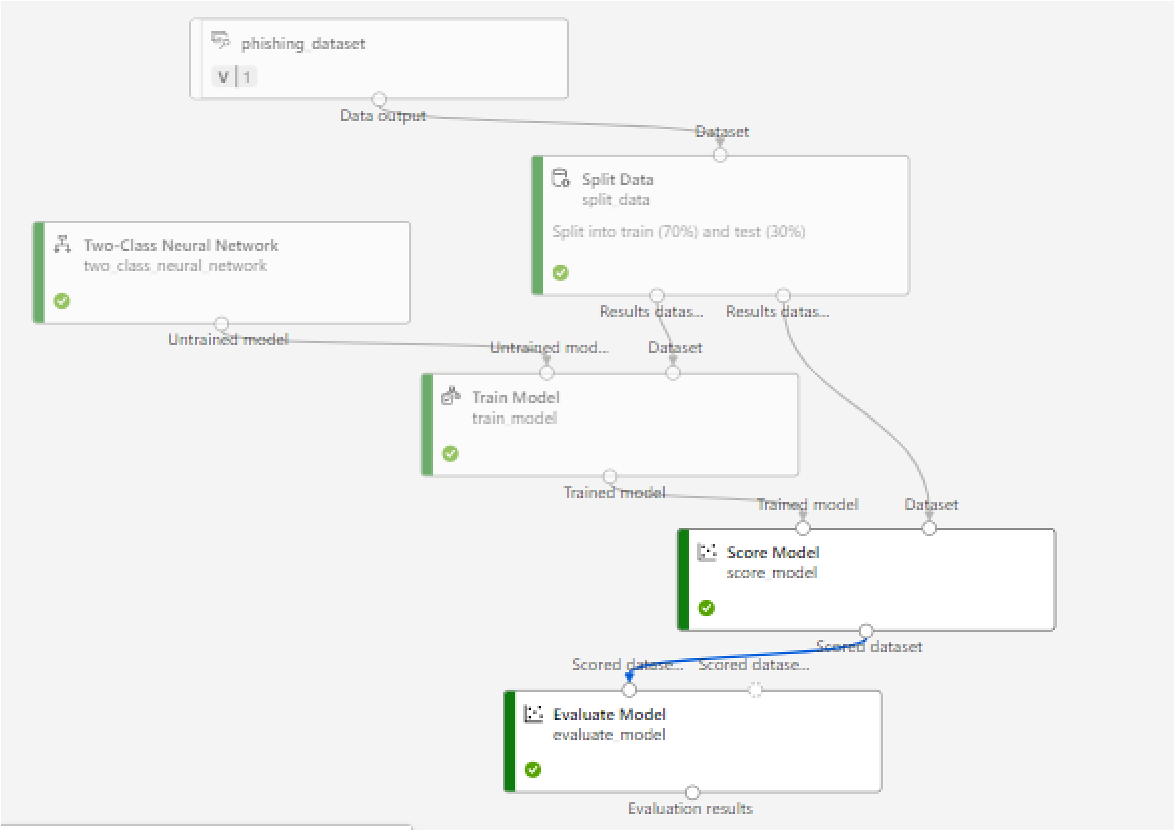


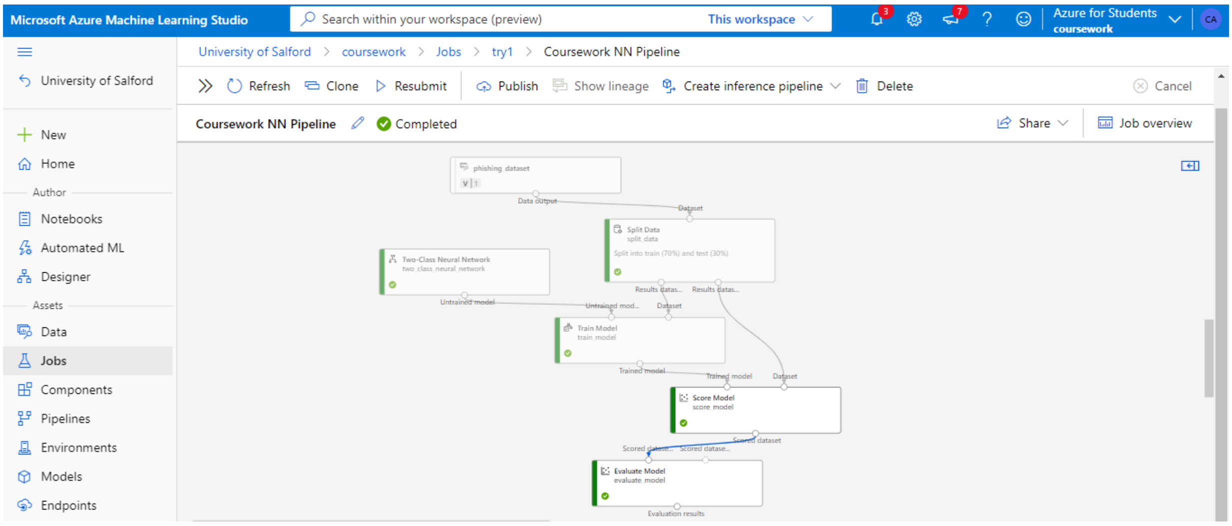
Next, the train model module was selected and the label column was specified as 'CLASS_LABEL'



Afterward, a 'Two-class Neural Network' module was introduced, the number of hidden nodes was specified to 50, and was connected to the pipeline. Then, the 'Evaluate model' module was added, and the entire pipeline run.

The final pipeline is shown below:





When the pipeline was run, a new column was appended at the end with the predicted values. A sample is shown below:

Rows	Columns
3,000	52

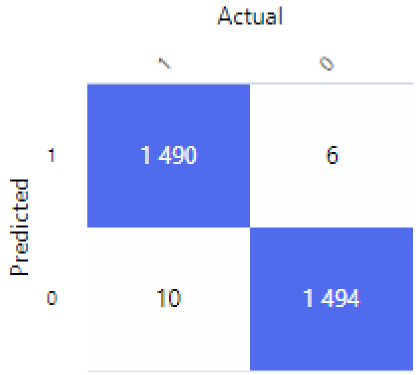
tHyper	CLASS_LABEL	Scored Labels	Scored Probabilities
	0	0	0.000045
	1	1	0.999604
	1	1	0.999274
	1	1	0.999814
	1	1	0.999999
	0	0	0.000301
	1	1	0.99035
	0	0	0.000201
	0	0	0.000001
	1	1	0.998609
	0	0	0.00015

To view, select a column in the table

Close examination of the results of the 'evaluation model' module shows the results score and the classification model as seen below.

Threshold 0.5

Accuracy	0.995
Precision	0.996
Recall	0.993
F1 Score	0.995
AUC	1



Results analysis and discussion

Performance metric used

The main performance metric used in evaluating both models is Accuracy. This was used because the dataset class output is well-balanced at 50:50.

A Confusion Matrix was also employed to have a broader look at the results.

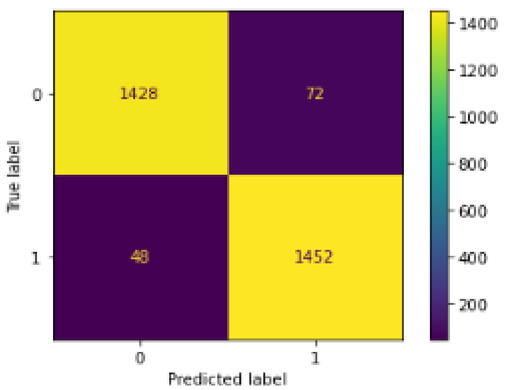
Presentation of results

KNN Classification

The confusion matrix and classification report using:

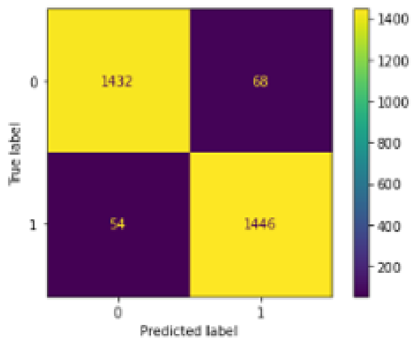
Randomly chosen (k=3). Below:

```
accuracy:0.96


Confusion Matrix
[[1428   72]
 [  48 1452]]


======================================================
Classification Report:
------------------------------------------------------
              precision    recall  f1-score   support

           0       0.97      0.95      0.96      1500
           1       0.95      0.97      0.96      1500

    accuracy                           0.96      3000
   macro avg       0.96      0.96      0.96      3000
weighted avg       0.96      0.96      0.96      3000
```



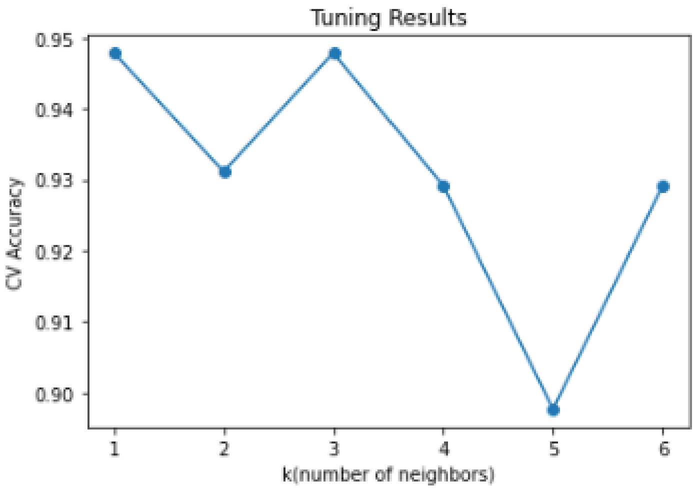
Derived best parameter (k=1) is shown below:

```
              precision    recall  f1-score   support

           0       0.96      0.95      0.96      1500
           1       0.96      0.96      0.96      1500

    accuracy                           0.96      3000
   macro avg       0.96      0.96      0.96      3000
weighted avg       0.96      0.96      0.96      3000

===============================================
Estimated Value Confusion Matrix
***********************************************
```



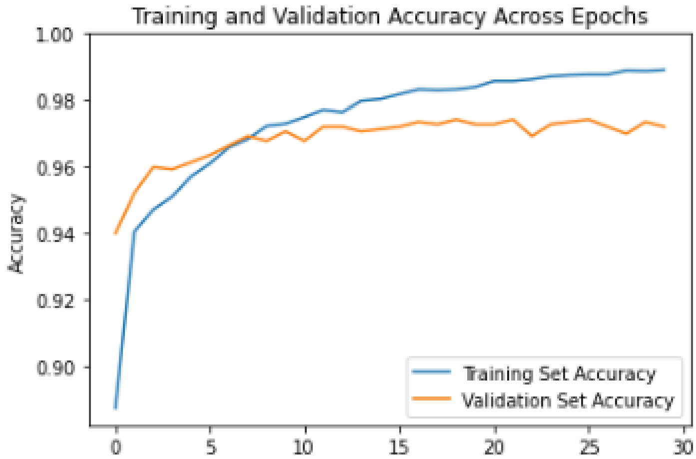
The figure below shows the graph of the best parameters and their corresponding accuracy after hyperparameter tuning

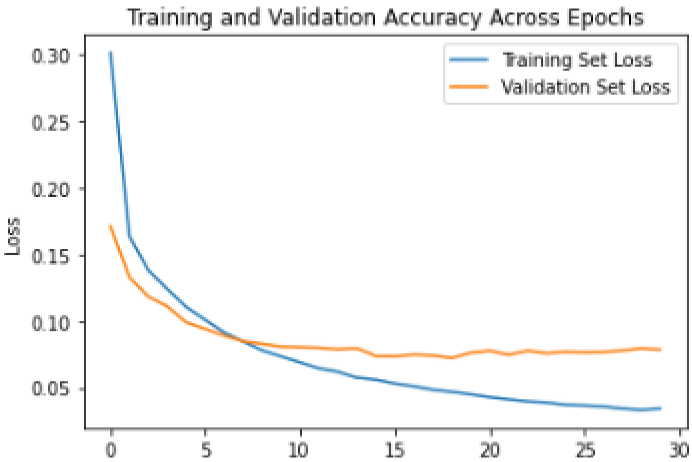
k(Number of Neighbors)	Accuracy
1	0.94785714
2	0.93114286
3	0.94785714
4	0.92914286
5	0.89771429
6	0.92914286



Artificial Neural Network

View graphs of Training and Validation set accuracy and losses below:





Classification report of the neural network:

```
In [97]: print(classification_report(y_test, nn_pred))
              precision    recall  f1-score   support

           0       0.97      0.97      0.97      1500
           1       0.97      0.97      0.97      1500

    accuracy                           0.97      3000
   macro avg       0.97      0.97      0.97      3000
weighted avg       0.97      0.97      0.97      3000
```

Azure Machine Learning

The scores and the classification reports are seen below:

<u>Comparison and discussion of results</u>

A side-by-side comparison of the results of the K-Nearest Neighbors and Neural Networks algorithms has been drafted, and is shown below:

	k-Nearest Neighbors	Neural Network
Accuracy	0.959333	0.973667
Precision	0.955086	0.973351
Recall	0.964000	0.974000
f1	0.959522	0.973675

From the results shown above, the Neural Network performs better than the K-NN algorithm judging by the listed performance metrics. Its accuracy, which was the main focus, was greater than that of the kNN by 1.43%. It also surpassed the k-NN model in Precision, Recall, and F1-Score by 1.83%, 1%, and 1.42% respectively.

The Azure machine learning model gives the best results among all three with an accuracy score of 99.5%

This metric (accuracy) measures the percentage of correctly predicted values (both phishing and legitimate websites) to the true values.

k(Number of Neighbors)	Accuracy
1	0.94785714
2	0.93114286
3	0.94785714
4	0.92914286
5	0.89771429
6	0.92914286

It can also be seen from the 'k-Neighbors against accuracy' table above, a slight change in the k-number can lead to a drastic change in the accuracy of the model. This implies that the model might not be suitable for future use with an entirely different dataset.

Ethical, legal, and professional considerations

Some considerations made in the

1. The dataset poses no risks to individuals or organisations by making sure the URL names were excluded.
2. The data involved was collected from legitimate, publicly available means.
3. All parties involved in the collection of the dataset were duly listed

Conclusion

Phishing attacks are a plague in this computer-driven world and these classification and prediction models with high accuracy can curb this scourge. All models showed good performance with the Neural Network having a slight edge over the k-NN model, and the Azure machine learning model performing best. This does not guarantee that these lofty scores will be replicated when applied to future datasets, however, it shows how useful these classification models can be in solving the phishing menace.