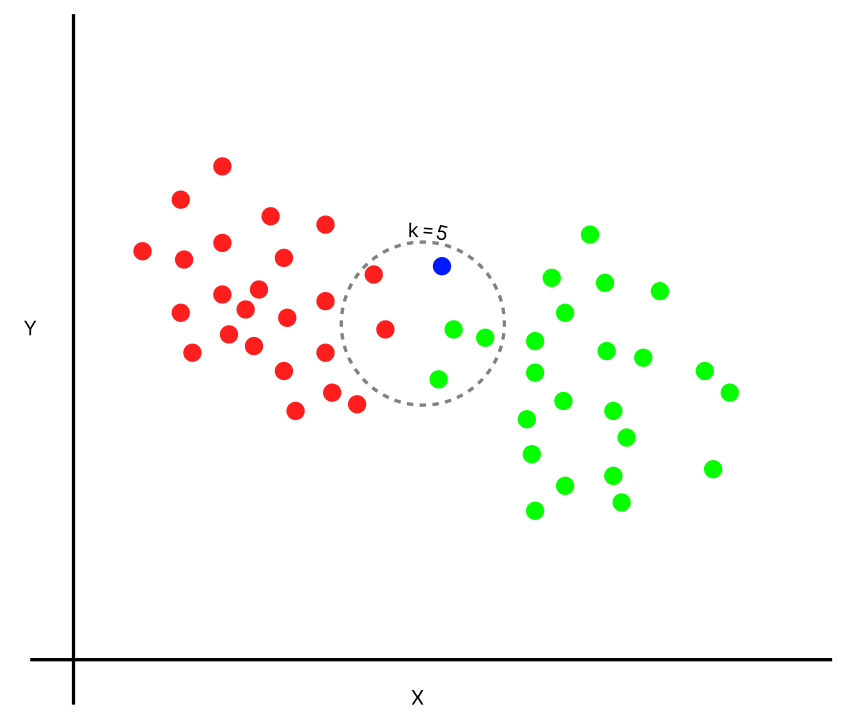
**Building a k-Nearest-Neighbours (kNN), Naïve Bayes (NB) Models & Hyperparameters Optimisation**

**Part (A): Building a k-Nearest-Neighbours (k-NN) Model with Scikit-learn**

k-Nearest-Neighbours (k-NN) is a supervised machine learning model. Supervised learning is when a model learns from data that is already labelled. A supervised learning model takes in a set of input objects and output values. The model then trains on that data to learn how to map the inputs to the desired output so it can learn to make predictions on unseen data.

k-NN models work by taking a data point and looking at the ‘k’ closest labelled data points. The data point is then assigned the label of the majority of the ‘k’ closest points. For example, if k = 5, and 3 of the points are ‘green’ and 2 are ‘red’, then the blue data point in question would be labelled ‘green’, since ‘green’ is the majority (as shown in the graph below, Fig.1).



**Fig.1.** Illustration of kNN working mechanism for k=5

Scikit-learn is a machine-learning library for Python. In this tutorial, we will build a k-NN model using Scikit-learn for detecting cyber-attacks with Machine Learning. This fundamental model demonstrates the machine learning capability of detecting malicious network traffic carrying a cyber-attack, AKA network intrusion detection, to build Intrusion Detection and Prevention Systems IDS and IPS.

**Step 1: Load your dataset**

For our k-NN model, the first step is to read the data we will use as input. For this example, we are using the **Network\_Intrusion\_Dataset**. To start, we will use Pandas to read the data. I will not go into detail on Pandas, but it is a library you should become familiar with if you’re looking to dive further into data science and machine learning.

***Code cell***

import pandas as pd

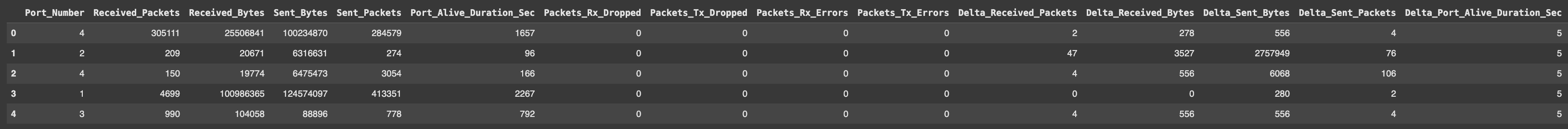
#Read in the data using pandas

df = pd.read\_csv('/content/Network\_Intrusion\_Dataset.csv')

#Check a sample of the dataset

df.head()

***Output cell***



**Step 2: Understand your dataset**

Next, let’s see how much data we have. List all features names and call the ‘shape’ function on our data frame to see how many rows and columns there are in our data. The rows indicate the number of examined network traffic for intrusion, and the columns indicate the number of features (ports, traffic load, rates, etc.) in the dataset for each patient.

***Code cell***

list(df.columns)

***Output cell***

A screen shot of a computer program

Description automatically generated

***Code cell***

#check the number of rows and columns in the dataset

df.shape

***Output cell***

A grey background with white numbers

Description automatically generated

We can see that we have 4998 rows of data (potential network intrusion incidents) and 33 columns (31 input features and 2 possible target outputs). The possible target outputs are a **binary target: Traffic\_Type** and a **multiclass target: Intrusion\_Traffic\_Type**

Visualise the frequency distribution for each target output to understand the possible predictions produced by the new IDS system built with k-NN.

***Code cell***

import plotly.express as px

#Construct a bar graph for Traffic\_Type target variable

Traffic\_Type\_fig = px.bar(df, x = 'Traffic\_Type', title = "Traffic Types")

#Construct a bar graph for Intrusion\_Traffic\_Type target variable

Intrusion\_Traffic\_Type\_fig = px.bar(df, x ='Intrusion\_Traffic\_Type', title =" Attacks Types")

#Construct a bivariate bar graph for both target variable

Intrusion\_Types\_Per\_Traffic\_fig = px.bar(df,x ='Traffic\_Type', color = 'Intrusion\_Traffic\_Type', title = "Attack Types In Intrusion Traffic", color\_discrete\_sequence = px.colors.qualitative.Vivid)

#Remove the bar outline by setting the marker.line.width attribute to 0

Traffic\_Type\_fig.update\_traces(dict(marker\_line\_width=0))

Intrusion\_Traffic\_Type\_fig.update\_traces(dict(marker\_line\_width=0))

Intrusion\_Types\_Per\_Traffic\_fig.update\_traces(dict(marker\_line\_width=0))

#Plot all constructed bar graphs

Traffic\_Type\_fig.show()

Intrusion\_Traffic\_Type\_fig.show()

Intrusion\_Types\_Per\_Traffic\_fig.show()

***Output cell***

A blue squares with white text

Description automatically generated

A graph with blue squares

Description automatically generated

A graph of different colored squares

Description automatically generated

Now, ensure that your variables hold the correct/expected data type to avoid any misleading interpretation of stats. If these are correct, examine the basic summary stats for your features; you should look for useless or unnecessary features to be dropped. Useless features are those which have no change in their values; unnecessary features are those that do not impact the target output directly (modifier variable) and/or do not influence any of the input features (confounder variable).

***Code cell***

df.info()

***Output cell***

A screenshot of a computer program

Description automatically generated

***Code cell***

df.describe().transpose()

***Output cell***

***A screenshot of a computer

Description automatically generated***

You will notice some values are written in **Python scientific notation**, which is a way of writing a large or a small number in powers of 10. Any number can be written in its scientific notification form by multiplying the number between 1 and 10 by a power of 10. Scientific notation reduces the format of the number representation.

For example, for very small numbers, like 0.0000000000001752, a negative exponent is used. The scientific notation would be 1.752 x 10-13, and the E-notation would be 1.752e-13. A negative exponent means to move the decimal place to the left instead of the right. On the other hand, for very large numbers, like 17520000000000, a positive exponent is used: 1.752e+13

To expand the scientific notation, we can control how pandas display float numbers by amending the **set\_options** function to format a string from columns.

***Code cell***

# To expand e scientific notation

pd.set\_option('display.float\_format', '{:.2f}'.format)

Now, rerun the summary statistics code and produce the summary stats in the **new format** without the panda’s scientific notation.

***Code cell***

df.describe().transpose()

***Output cell***

A screenshot of a computer

Description automatically generated

**Step 3: Prepare your dataset**

From the basic stats, you should notice instantly **11 useless variables** whose values do not change by simply examining the minimum and maximum values for each input feature. Remove all useless features from your dataset. We can drop the columns in a range since there are so many of them, using the **drop** function while applying the **loc[ ]** method. This is called slicing the data frame. To learn more about data frame slicing with **loc[ ]**, visit this helpful [link](https://www.geeksforgeeks.org/python-pandas-dataframe-loc/). Finally, display the basic summary stats with **include=”all”** for the retained 22 input and output features to ensure all useless features were dropped.

***Code cell***

df = df.drop(columns=df.loc[:, 'Packets\_Rx\_Dropped':'Packets\_Tx\_Errors'].columns)

df = df.drop(columns=df.loc[:, 'Delta\_Packets\_Rx\_Dropped':'Delta\_Packets\_Tx\_Errors'].columns)

df = df.drop(['Is\_Valid', 'Table\_ID', 'Max\_Size'], axis=1)

df.describe(include="all").transpose()

***Output cell***

A screenshot of a graph

Description automatically generated

Examine the portion of missing values in your dataset for each feature and observe there are no missing data values in the data frame for all input and output features.

***Code cell***

df.isna().sum()/len(df)\*100

***Output cell***

A screenshot of a computer

Description automatically generated

**Step 4: Save your prepared dataset**

Once the prepared dataset is saved in .csv format, ensure you download it to your local machine.

***Code cell***

df.to\_csv(r'/content/Prepared\_Network\_Intrusion\_Dataset.csv', index = False)

***Output cell***

A screenshot of a computer

Description automatically generated

**Step 4: Load your prepared dataset and assign the inputs and target output for modelling**

Start by assigning your input features to the environment and the target out which you want to model. In this tutorial, we try to model the **Traffic\_Type** binary outcome. *For your homework, you can try to repeat all the next steps to model the multi-class target output Intrusion\_Traffic\_type*

***Code cell***

df\_prepared = pd.read\_csv('/content/Prepared\_Network\_Intrusion\_Dataset.csv')

***Code cell***

#create a dataframe with all training data except the target column

X = df\_prepared.drop(columns=['Traffic\_Type','Intrusion\_Traffic\_Type'])

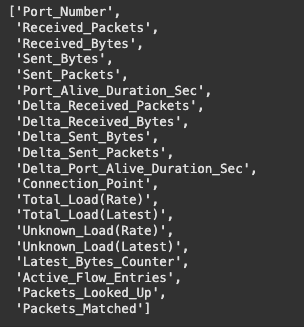
# here, we select one target variable to model, Traffic\_Type

y = df\_prepared['Traffic\_Type']

#check that the list of input variables

list(X)

***Output cell***



***Code cell***

#check that the list of target variable

y.head()

***Output cell***

A screenshot of a phone

Description automatically generated

**Step 5: Split the dataset into train and test data subsets**

Now, we will split the dataset into training data and testing data. The training data is the data that the model will learn from. The testing data is the data we will use to see how well the model performs on unseen data. Scikit-learn has a function we can use called ‘train\_test\_split’ that makes it easy for us to split our dataset into training and testing data.

***Code cell***

from sklearn.model\_selection import train\_test\_split

#split dataset into train and test data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=14, stratify=y)

#This is to show the number of instances and input features in the training and test sets

print('X\_train Instances', X\_train.shape)

print('X\_test Instances', X\_test.shape)

***Output cell***



**‘train\_test\_split’** takes in 5 parameters. The first two parameters are the input and target data we split up earlier. Next, we will set ‘test\_size’ to 0.2. This means that 20% of all the data will be used for testing, which leaves 80% of the data as training data for the model to learn from. Setting ‘random\_state’ to 1 ensures that we get the same split each time so we can reproduce our results.

Setting ‘stratify’ to y makes our training split represent the proportion of each value in the y variable. For example, in our dataset, if 47% of Network Traffic has intrusion and 57% is normal, don’t include embedded attacks; setting ‘stratify’ to y will ensure that the random split has 47% of Network Traffic instances with intrusion and 57% without intrusion in each data subset, training and test.

**Step 6: Building and training the kNN model**

First, we will create a new k-NN classifier and set ‘n\_neighbors’ to 9. To recap, this means that if at least 5 out of the 9 nearest points to a new data point represent normal traffic without an attack, then the new data point will be labelled as ‘normal’ and vice versa. In other words, a new data point is labelled with the majority from the 9 nearest points.

We have set ‘n\_neighbors’ to 9 as a starting point. We will go into more detail below on how to better select a value for ‘n\_neighbors’ so that the model can improve its performance.

Next, we need to train the model. To train our new model, we will use the ‘fit’ function and pass in our training data as parameters to fit our model to the training data.

***Code cell***

from sklearn.neighbors import KNeighborsClassifier

# Create a KNN classifier

knn = KNeighborsClassifier(n\_neighbors = 9)

# Fit the classifier to the data

knn.fit(X\_train,y\_train)

***Output cell***

A close-up of a sign

Description automatically generated

**Step 7: Testing the kNN model**

Once the model is trained, we can use the ‘predict’ function on our model to make predictions on our test data. As seen when inspecting ‘y’ earlier, **“Normal”** indicates that the traffic does not have an attack, and **“Intrusion”** indicates that the network traffic has an embedded attack. To see a comparison between both actual and predicted labels, we can also build a comparison data frame in pandas.

***Code cell***

#Perform predictions on the test data

y\_pred=knn.predict(X\_test)

#Create a dataframe for comparing the actual vs predicted results by kNN mode

compare\_results\_knn\_df = pd.DataFrame({'Actual':y\_test, 'Predicted': y\_pred})

compare\_results\_knn\_df.to\_csv(r'/content/knn\_pred\_comparison.csv', index=True)

compare\_results\_knn\_df

***Output cell***

A screenshot of a phone

Description automatically generated

**Step 8: Evaluate the kNN model’s test results**

Here, you should look at the confusion matrix and the classification report for your first kNN model with k=9. Also, plot the roc\_auc to establish how well the model discriminates between normal and intrusion traffic.

To produce a classification report, we use the classification\_report method from sklearn.metrics package.

***Code cell***

from sklearn.metrics import classification\_report

print(classification\_report(y\_test, y\_pred))

***Output cell***

A screenshot of a computer

Description automatically generated

***Code cell***

#Import the packages for costructing the confusion matrix

from sklearn.metrics import confusion\_matrix

#Import the packages for plotting the confusion matrix

from sklearn.metrics import ConfusionMatrixDisplay

#Costruct the confusion matrix based on…

#comparing actual values (y\_test) vs predicted (y\_pred) in test data

cm\_knn = confusion\_matrix(y\_test, y\_pred, labels = knn.classes\_)

#Plot the confusion matrix

disp\_knn\_cm = ConfusionMatrixDisplay(cm\_knn, display\_labels=knn.classes\_)

disp\_knn\_cm.plot()

***Output cell***

A chart of a number of warning labels

Description automatically generated with medium confidence

***Code cell***

from sklearn.metrics import RocCurveDisplay

knn\_roc = RocCurveDisplay.from\_estimator(knn, X\_test, y\_test)

***Output cell***

A graph of a positive label

Description automatically generated

From the classification report, the roc\_auc is high at 94%, indicating very good discrimination between intrusion and normal traffic; notice that the model’s accuracy is high at 87%; from the confusion matrix, you can see there are false alarms these are False Positives (FP), but more worrying is to see real attacks or “intrusions” without any alarms because the system thinks they are normal traffic; these are known False Negatives (FN).

**Step 9: Improving the kNN model’s test results with finding a different k value**

Let’s try to amend the value of the **hyperparameter k** and see if any improvement can be made to the above metrics. For that, we create a loop to create multiple kNN models with different values of k.

***Code cell***

# Calculating error for K values between 1 and 40

error = []

import numpy as np

import matplotlib.pyplot as plt

# Calculating error for K values between 1 and 40

for i in range(1, 40):

knn2 = KNeighborsClassifier(n\_neighbors=i)

knn2.fit(X\_train, y\_train)

pred\_i = knn2.predict(X\_test)

error.append(np.mean(pred\_i != y\_test))

plt.figure(figsize=(12, 6))

plt.plot(range(1, 40), error, color='red', linestyle='dashed', marker='o',

markerfacecolor='blue', markersize=10)

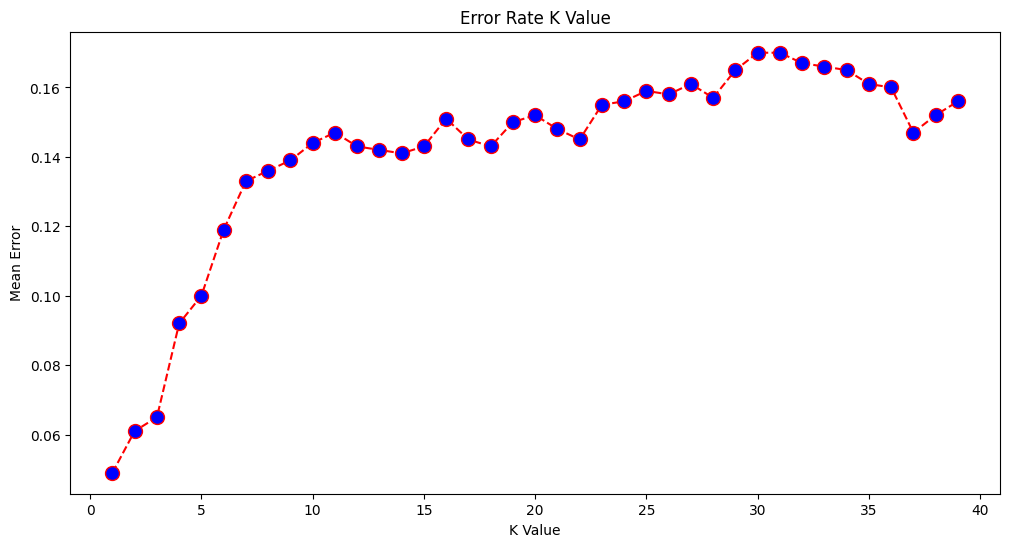
plt.title('Error Rate K Value')

plt.xlabel('K Value')

plt.ylabel('Mean Error')

Before building any KNN model, we start without having any prediction errors. As we build kNN models with different k values and test them, we should decide to use the k value, which produces the lowest prediction error.

***Output cell***



It seems that the lowest average error is produced when **k=1**; therefore, rebuild your kNN model with k=1 and observe any improvement in the evaluation results. Name your new model kNN1.

***Code cell***

from sklearn.neighbors import KNeighborsClassifier

# Create KNN classifier

knn1 = KNeighborsClassifier(n\_neighbors = 1)

# Fit the classifier to the data

knn1.fit(X\_train,y\_train)

#Perform predictions on the test data

y\_pred=knn1.predict(X\_test)

***Code cell***

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import ConfusionMatrixDisplay

cm\_knn1 = confusion\_matrix(y\_test, y\_pred, labels = knn1.classes\_)

disp\_knn1\_cm = ConfusionMatrixDisplay(cm\_knn1, display\_labels=knn1.classes\_)

disp\_knn1\_cm.plot()

***Output cell***

A chart of different colored squares

Description automatically generated

***Code cell***

from sklearn.metrics import classification\_report

print(classification\_report(y\_test, y\_pred))

***Output cell***

A screenshot of a computer

Description automatically generated

***Code cell***

from sklearn.metrics import RocCurveDisplay

knn\_roc = RocCurveDisplay.from\_estimator(knn1, X\_test, y\_test)

***Output cell***

A graph of a positive label

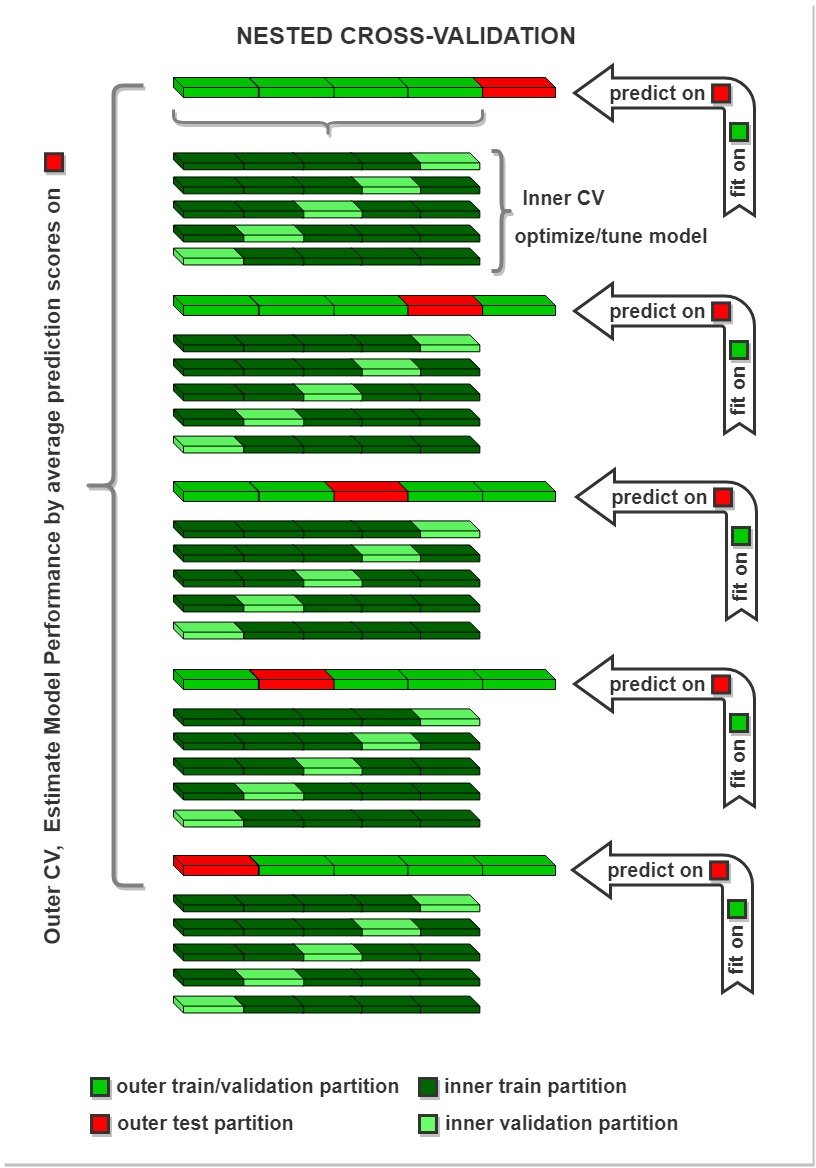
Description automatically generated

From the classification reports, the **roc\_auc** has slightly improved to 95%, indicating excellent discrimination between intrusion and normal traffic; notice that the model’s accuracy has improved to 95%; from the confusion matrix, you can see a reduction in false alarms, the False Positives (FP). It is less worrying to see a reduction in undetected real attacks or “intrusions” without any alarms; therefore, we have a reduction in False Negatives (FN).

**Step 10: Improving the kNN model’s performance with Hyperparameter Tuning**

When building our initial k-NN model, we set the parameter ‘n\_neighbors’ to 3 as a starting point with no real logic behind that choice.

Hyper tuning parameters is when you go through a process to find the optimal combination of parameters for your model to improve accuracy, for example. In our case, we will use GridSearchCV to find the optimal value for **‘n\_neighbors’** in combination with the best **distance** method. GridSearchCV, AKA Nested Cross Validation (Figure 2), works by training our model multiple times on a range of parameters that we specify. That way, we can test our model with each parameter and figure out the optimal values to get the best accuracy results.



**Fig.2.** Illustration of Nested Cross-Validation k=5

For our model, we will specify a range of values for k ‘n\_neighbors’ in order to see which value works best for our model, in combination with the best distance metrics that can be used to measure the distance between points. To do this, we will create a dictionary called **Parameter Grid**, setting ‘n\_neighbors’ as the key and using the **numpy package** to create an array of values from 1 to 24.

Our new model using grid search will take in a new k-NN classifier, our param\_grid and a cross-validation value of 5 to find the optimal combination for the ‘n\_neighbors’ value and ‘distance’ method Euclidean vs Manhattan.

***Code cell***

from sklearn.model\_selection import GridSearchCV

import numpy as np

#create new a knn model

knn = KNeighborsClassifier()

#create a dictionary of all values we want to test for n\_neighbors and distances

param\_grid = {'n\_neighbors': np.arange(1, 25), 'metric': ['euclidean', 'manhattan']}

#use gridsearch to test all values for n\_neighbors

knn\_gscv = GridSearchCV(knn, param\_grid, cv=5, scoring = 'roc\_auc')

#fit model to data

knn\_gscv.fit(X, y)

***Output cell***



After training, we can check which of our values for ‘n\_neighbors’ in combination with ‘metric distance’ that we tested performed the best. To do this, we will call ‘best\_params\_’ on our model. To view the best combination of parameters for the hyper-tuned model

***Code cell***

# Check top performing n\_neighbors’ value

knn\_gscv.best\_params\_

***Output cell***



Perform predictions on the test dataset with the newly formed hyper-tuned model, then evaluate its performance. Observe any improvements in detecting intrusion traffic and reduction in false alarms.

***Code cell***

# Perform testing on test dataset

y\_pred = knn\_gscv.predict(X\_test)

# Construct a confusion matrix

cm\_knn\_gscv = confusion\_matrix(y\_test, y\_pred, labels = knn\_gscv.classes\_)

disp\_knn\_gscv\_cm = ConfusionMatrixDisplay(cm\_knn\_gscv, display\_labels=knn\_gscv.classes\_)

disp\_knn\_gscv\_cm.plot()

# Display the classification report

print(classification\_report(y\_test, y\_pred))

***Output cell***

A screenshot of a computer

Description automatically generated

***Code cell***

from sklearn.metrics import RocCurveDisplay

knn\_gscv\_roc = RocCurveDisplay.from\_estimator(knn\_gscv, X\_test, y\_test)

***Output cell***

A graph of a positive label

Description automatically generated

When comparing the outputs from step 8 and step 10, with hyperparameter tuning we successfully reduced the number of false alarms significantly down from 55 to just 8. Also reduced the undetected attacks from 84 to 8.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Part (B): Building a Gaussian Naive Bayes Classifier Model with Scikit-learn**

**Types of Naive Bayes Classifiers**

There are three types of Naive Bayes Classifiers –

1. Gaussian Naive Bayes. This classifier is used when the predictors' values are continuous, and it is assumed that they follow Gaussian distribution.
2. Bernoulli Naive Bayes. This classifier is used when the predictors are Boolean in nature, and it is assumed they follow the Bernoulli distribution.
3. Multinomial Naive Bayes. This classifier uses multinomial distribution and is mostly used for document or text classification problems.

In this section, we will take you through an end-to-end example of the Gaussian Naive Bayes classifier in Python Sklearn using a cancer dataset. We will be using the Gaussian Naive Bayes function of Sklearn, i.e. **GaussianNB(),** for our example.

**Step 1: Import the required Python libraries**

We will start by loading some initial libraries to load and visualise the dataset.

***Code cell***

import numpy as np

import pandas as pd

import plotly.express as px

**Step 2: Load your dataset**

Now we will upload the cancer detection dataset that we have obtained from Kaggle for performing our Naive Bayes classification.

***Code cell***

dataset = pd.read\_csv('/content/BC\_Data.csv')

**Step 3: Understand and explore your dataset**

Let us take an initial look into the dataset with the help of the **head()** function.

***Code cell***

dataset.head()

***Output cell***



Next, we will explore the columns present in the dataset through the **info()** function.

***Code cell***

dataset.info()

***Output cell***

A screenshot of a computer

Description automatically generated

You can visualise your dataset with **plotly.express** package. Let’s visualise the **malignant and benign** tumours by looking at the mean radius and mean texture of these tumours. To visualise the dataset, we will first split the scatter plot into two parts. The first part contains information about patients who have a malignant (dangerous) tumour and the other half data for benign (safe) tumours.

***Code cell***

fig = px.scatter(dataset, x="radius\_mean",y="texture\_mean", color = "diagnosis", width=800, height=800)

fig.show()

***Output cell***

A graph with red and blue dots

Description automatically generated

**Step 4: Prepare and preprocess your dataset**

From the above information shows that the **id** and **unnamed:32** columns are not useful, so we can remove them.

***Code cell***

dataset = dataset.drop(["id"], axis = 1)

dataset = dataset.drop(["Unnamed: 32"], axis = 1)

We now break our data frame into x and y. The x contains all the independent predictor variables, and y contains the prediction of diagnosis.

***Code cell***

X = dataset.drop(["diagnosis"], axis = 1)

y = dataset['diagnosis']

Let‘s perform Minimum – Maximum Normalisation, here we scale all the input features values between 0 and 1

***Code cell***

# Perform Minimum - Maximum Normalization:

X1 = (X - np.min(X)) / (np.max(X) - np.min(X))

Now, with the help of the sklearn library train\_test\_split module, we will split the dataset into training and testing parts.

***Code cell***

from sklearn.model\_selection import train\_test\_split

X1\_train, X1\_test, y\_train, y\_test = train\_test\_split(X1, y, test\_size = 0.3, random\_state = 42)

**Step 5: Build a Naïve Bayes model for cancer tumour prediction**

Now, we will import the Gaussian Naive Bayes module of Sklearn GaussianNB and create an instance of it. We can pass x\_train and y\_train to fit the model.

***Code cell***

from sklearn.naive\_bayes import GaussianNB

nb = GaussianNB()

nb.fit(X1\_train, y\_train)

y\_pred=nb.predict(X1\_test)

**Step 6: Evaluate your Naïve Bayes model for cancer tumour prediction**

The following classification report on the test data shows how well our model Sklearn Gaussian Naive Bayes model has performed for predicting cancer. The accuracy is 94%. And you can also plot the confusion matrix for the predictions on the test dataset

***Code cell***

from sklearn.metrics import classification\_report

print(classification\_report(y\_test, y\_pred))

***Output cell***

A screenshot of a computer

Description automatically generated

***Code cell***

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import ConfusionMatrixDisplay

nb\_cm = confusion\_matrix(y\_test, y\_pred, labels = nb.classes\_)

nb\_cm = ConfusionMatrixDisplay(nb\_cm, display\_labels = nb.classes\_)

nb\_cm.plot()

***Output cell***

A chart of different predictions

Description automatically generated with medium confidence

***Code cell***

from sklearn.metrics import RocCurveDisplay

nb\_roc = RocCurveDisplay.from\_estimator(nb, X1\_test, y\_test)

***Output cell***

A graph of a positive label

Description automatically generated