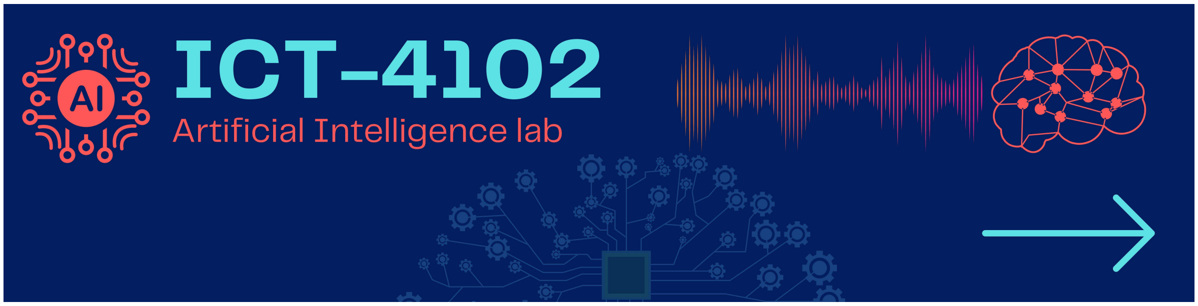
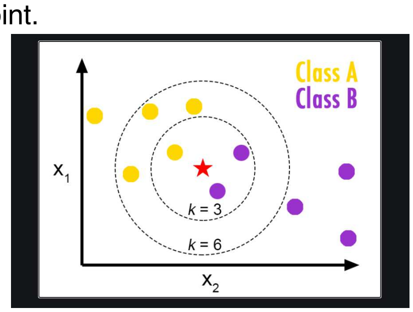


()

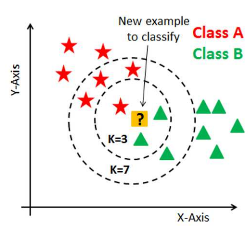
K Nearest Neighbors(K-NN)

Introduction

kNN is one of the simplest yet powerful supervised ML algorithms. It is widely used for classification problems as well as can be used for regression problems. The data-point is classified on the basis of its k Nearest Neighbors, followed by the majority vote of those nearest neighbors; a query point is assigned the data class which has the most representatives within the nearest neighbors of the point.



()



() K-NN is a non-parametric algorithm, which means it does not make any assumptions on underlying data. As in the case of Logistic Regression, it is assumed that all data-points are linearly separable (almost or completely).

Advantages :

- Easy and simple machine learning model.
- Few hyperparameters to tune.

Disadvantages :

- k should be wisely selected.
- Large computation cost during runtime if sample size is large.
- Proper scaling should be provided for fair treatment among features.

Hyperparameters :

KNN mainly involves two hyperparameters, K value & distance function.

K value :

how many neighbors to participate in the KNN algorithm. k should be tuned based on the validation error.

distance function :

Euclidean distance is the most used similarity function. Manhattan distance, Hamming Distance, Minkowski distance are different alternatives.

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

Task

You've been given a classified data set from a company! They've hidden the feature column names but have given you the data and the target classes.

We'll try to use KNN to create a model that directly predicts a class for a new data point based off of the features.

Let's grab it and use it!

Import Libraries

In [2]:
```
import pandas as pd
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline
```

Get the Data

Set index_col=0 to use the first column as the index.

In [3]:
```
df = pd.read_csv("Classified Data",index_col=0)
```

In [5]:
```
df
```

Out[5]:

	WTT	PTI	EQW	SBI	LQE	QWG	FDJ	PJF	HQE	
0	0.913917	1.162073	0.567946	0.755464	0.780862	0.352608	0.759697	0.643798	0.879422	1
1	0.635632	1.003722	0.535342	0.825645	0.924109	0.648450	0.675334	1.013546	0.621552	1
2	0.721360	1.201493	0.921990	0.855595	1.526629	0.720781	1.626351	1.154483	0.957877	1
3	1.234204	1.386726	0.653046	0.825624	1.142504	0.875128	1.409708	1.380003	1.522692	1
4	1.279491	0.949750	0.627280	0.668976	1.232537	0.703727	1.115596	0.646691	1.463812	1
...	...	...	...	...	...	...	...	...	...	
995	1.010953	1.034006	0.853116	0.622460	1.036610	0.586240	0.746811	0.319752	1.117340	1
996	0.575529	0.955786	0.941835	0.792882	1.414277	1.269540	1.055928	0.713193	0.958684	1
997	1.135470	0.982462	0.781905	0.916738	0.901031	0.884738	0.386802	0.389584	0.919191	1
998	1.084894	0.861769	0.407158	0.665696	1.608612	0.943859	0.855806	1.061338	1.277456	1
999	0.837460	0.961184	0.417006	0.799784	0.934399	0.424762	0.778234	0.907962	1.257190	1

1000 rows × 11 columns

Standardize the Variables

Because the KNN classifier predicts the class of a given test observation by identifying the observations that are nearest to it, the scale of the variables matters. Any variables that are on a large scale will have a much larger effect on the distance between the observations, and hence on the KNN classifier, than variables that are on a small scale.

In [6]:
```
from sklearn.preprocessing import StandardScaler
```

In [7]:
```
scaler = StandardScaler()
```

In [9]:
```python
scaler.fit(df.drop('TARGET CLASS',axis=1))
```

Out[9]:
```
StandardScaler()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [10]:
```python
scaled_features = scaler.transform(df.drop('TARGET CLASS',axis=1))
```

In [11]:
```python
df_feat = pd.DataFrame(scaled_features,columns=df.columns[:-1])
df_feat.head()
```

Out[11]:

	WTT	PTI	EQW	SBI	LQE	QWG	FDJ	PJF	HQI
0	-0.123542	0.185907	-0.913431	0.319629	-1.033637	-2.308375	-0.798951	-1.482368	-0.94971
1	-1.084836	-0.430348	-1.025313	0.625388	-0.444847	-1.152706	-1.129797	-0.202240	-1.82805
2	-0.788702	0.339318	0.301511	0.755873	2.031693	-0.870156	2.599818	0.285707	-0.68249
3	0.982841	1.060193	-0.621399	0.625299	0.452820	-0.267220	1.750208	1.066491	1.24132
4	1.139275	-0.640392	-0.709819	-0.057175	0.822886	-0.936773	0.596782	-1.472352	1.04077

Train Test Split

In [12]:
```python
from sklearn.model_selection import train_test_split
```

In [13]:
```python
X_train, X_test, y_train, y_test = train_test_split(scaled_features,df['TARGET
                                                    test_size=0.30)
```

Using KNN

Remember that we are trying to come up with a model to predict whether someone will TARGET CLASS or not. We'll start with k=1.

In [14]:
```python
from sklearn.neighbors import KNeighborsClassifier
```

In [15]:
```python
knn = KNeighborsClassifier(n_neighbors=1)
```

In [16]:
```
knn.fit(X_train,y_train)
```

Out[16]:
```
KNeighborsClassifier(n_neighbors=1)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [17]:
```
pred = knn.predict(X_test)
```

Predictions and Evaluations

Let's evaluate our KNN model!

In [18]:
```
from sklearn.metrics import classification_report,confusion_matrix
```

In [19]:
```
print(confusion_matrix(y_test,pred))
```

```
[[145  15]
 [  9 131]]
```

In [20]:
```
print(classification_report(y_test,pred))
```

```
              precision    recall  f1-score   support

           0       0.94      0.91      0.92       160
           1       0.90      0.94      0.92       140

    accuracy                           0.92       300
   macro avg       0.92      0.92      0.92       300
weighted avg       0.92      0.92      0.92       300
```

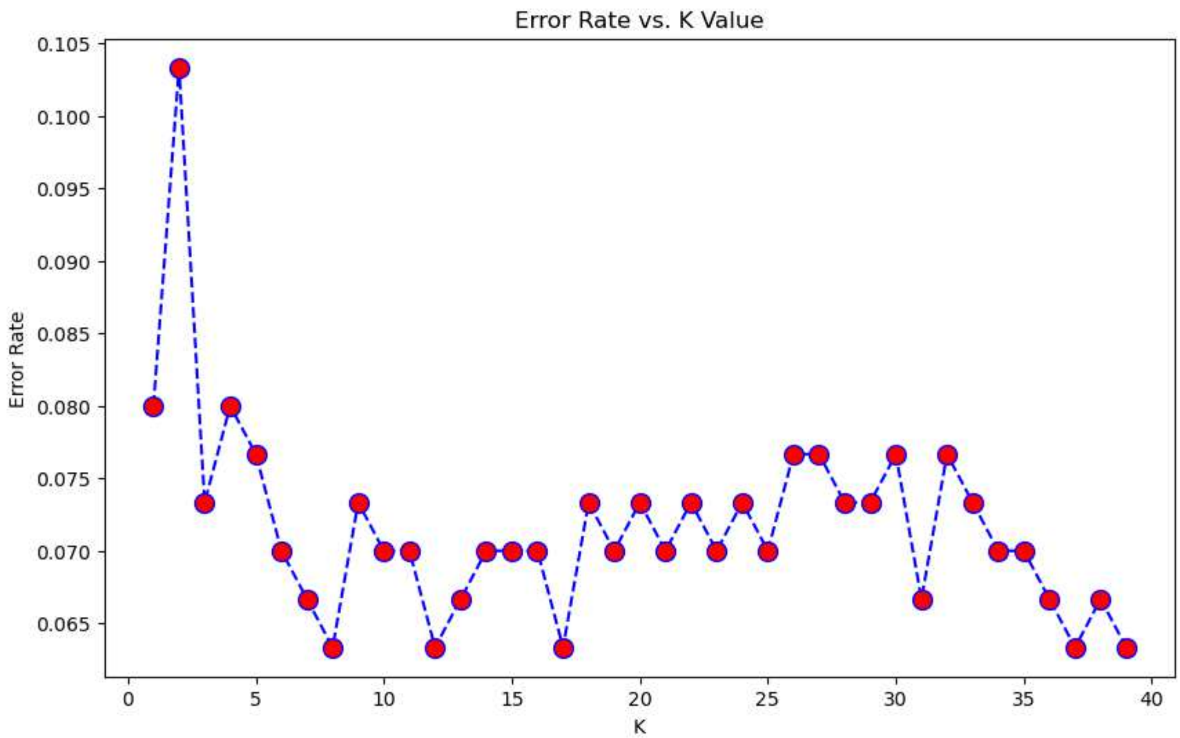
Choosing a K Value

Let's go ahead and use the elbow method to pick a good K Value:

In [21]:
```
error_rate = []

# Will take some time
for i in range(1,40):

    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_train,y_train)
    pred_i = knn.predict(X_test)
    error_rate.append(np.mean(pred_i != y_test))
```

In [22]:
```
plt.figure(figsize=(10,6))
plt.plot(range(1,40),error_rate,color='blue', linestyle='dashed', marker='o',
         markerfacecolor='red', markersize=10)
plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
```

Out[22]:
```
Text(0, 0.5, 'Error Rate')
```



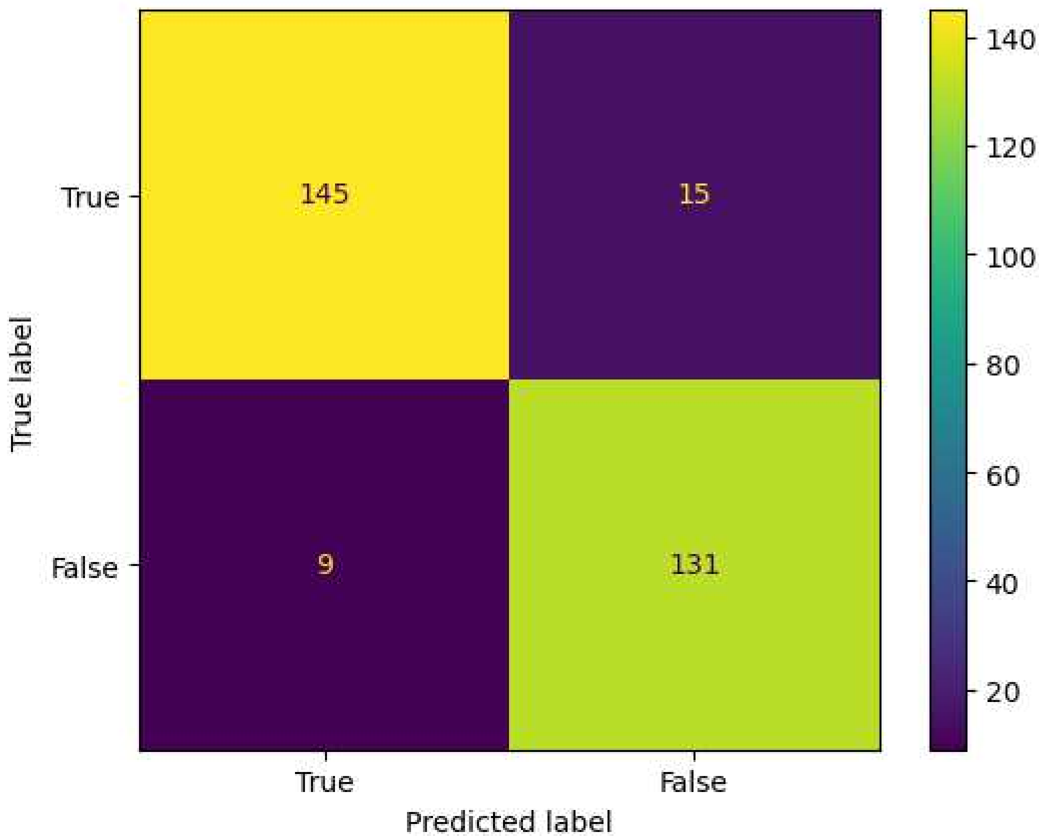
Here we can see that that after arouns K>19 the error rate just tends to hover around 0.07-0.75 Let's retrain the model with that and check the classification report!

In [31]:

```
# FIRST A QUICK COMPARISON TO OUR ORIGINAL K=1
knn = KNeighborsClassifier(n_neighbors=1)

knn.fit(X_train,y_train)
pred = knn.predict(X_test)

print('WITH K=1')
print('\n')
print(confusion_matrix(y_test,pred))
print('\n')
print(classification_report(y_test,pred))
```

```
WITH K=1


[[145  15]
 [  9 131]]


              precision    recall  f1-score   support

           0       0.94      0.91      0.92       160
           1       0.90      0.94      0.92       140

    accuracy                           0.92       300
   macro avg       0.92      0.92      0.92       300
weighted avg       0.92      0.92      0.92       300
```

In [32]:
```
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

conf_matrix = confusion_matrix(y_test, pred)
vis = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels = [Tr

vis.plot()
plt.grid(False)
plt.show()
```

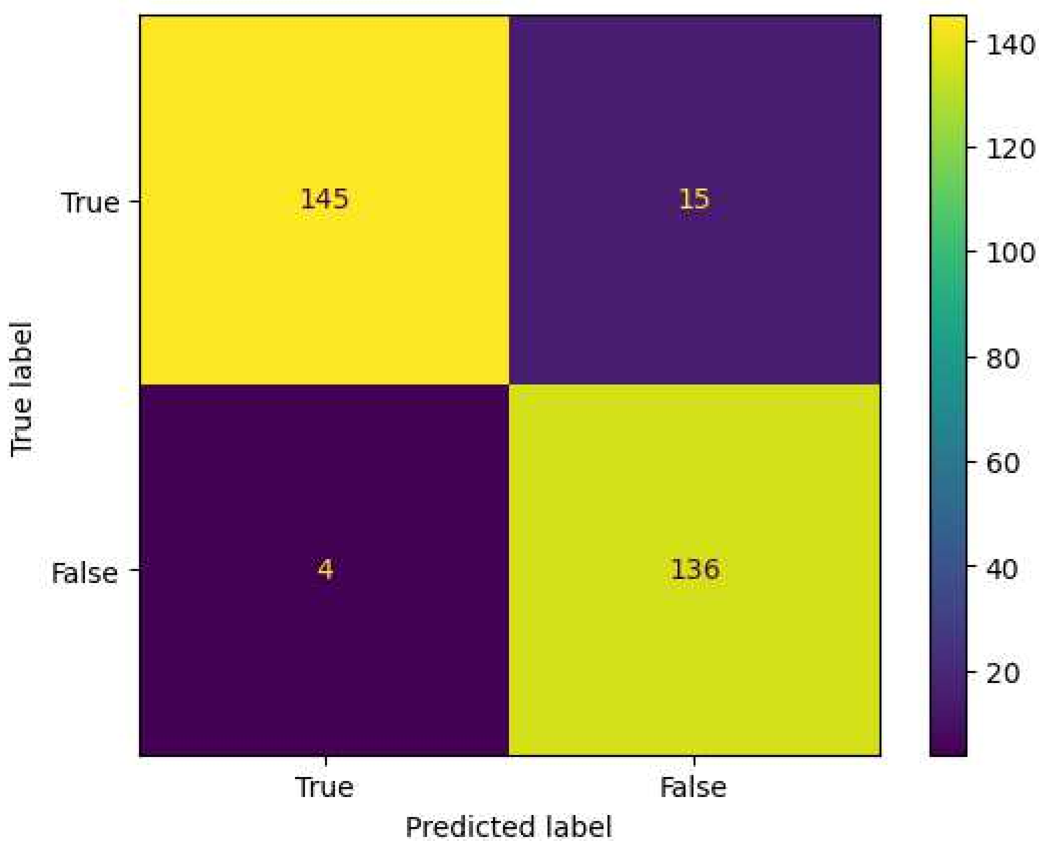


In [28]:

```
# NOW WITH K=23
knn = KNeighborsClassifier(n_neighbors=17)

knn.fit(X_train,y_train)
pred = knn.predict(X_test)

print('WITH K=17')
print('\n')
print(confusion_matrix(y_test,pred))
print('\n')
print(classification_report(y_test,pred))
```

```
WITH K=17


[[145  15]
 [  4 136]]


              precision    recall  f1-score   support

           0       0.97      0.91      0.94       160
           1       0.90      0.97      0.93       140

    accuracy                           0.94       300
   macro avg       0.94      0.94      0.94       300
weighted avg       0.94      0.94      0.94       300
```

In [30]:
```
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

conf_matrix = confusion_matrix(y_test, pred)
vis = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels = [Tı

vis.plot()
plt.grid(False)
plt.show()
```



Great job!

We were able to squeeze some more performance out of our model by tuning to a better K value!