# KNN

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.

## Reading data

df = pd.read\_csv("C:/Users/G01212601/Downloads/Py-DS-ML-Bootcamp-master/Refactored\_Py\_DS\_ML\_Bootcamp-master/14-K-Nearest-Neighbors/Classified Data.csv",index\_col=0)  
print(df.info())

## Applying feature scaling

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.

# Performing scaling on the features  
scaler = StandardScaler()  
scaler.fit(df.drop("TARGET CLASS",axis=1))  
scaled\_features = scaler.transform(df.drop("TARGET CLASS",axis=1))  
df\_scaled\_feat = pd.DataFrame(scaled\_features,columns=df.columns[:-1])  
print(("Head of scaled features - "))  
print(df\_scaled\_feat.head())

## Specifying train and test data

# Specifying X and y and doing train test split  
X = df\_scaled\_feat  
y = df["TARGET CLASS"]  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30, random\_state=101)

## Applying model with k =1

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.

# Using KNN  
knn = KNeighborsClassifier(n\_neighbors=1)  
knn.fit(X\_train,y\_train)  
pred = knn.predict(X\_test)

## Evaluating basic model with k = 1

# Evaluation with k = 1  
# Evaluating model using scikit's classification report  
print("\*\*\*\*\*Classification report\*\*\*\*\*")  
print(classification\_report(y\_true=y\_test,y\_pred=pred))  
print()  
  
# Evaluating model using confusion matrix  
print("\*\*\*\*\*Confusion matrix\*\*\*\*\*")  
print(confusion\_matrix(y\_true=y\_test,y\_pred=pred))  
print()

## Deciding the value of k

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

# Choosing a K Value using elbow method  
error\_rate = []  
for i in range(1,50):  
 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))  
  
plt.figure(figsize=(10,6))  
plt.plot(range(1,40),error\_rate,color='blue', linestyle='dashed', marker='o',markerfacecolor='red', markersize=10)  
plt.title("K vs Error rate")  
plt.xlabel("K")  
plt.ylabel("Error rate")

Here we can see that that after around K = 40 the error rate just tends to hover around 0.06-0.05 Let's retrain the model with that and check the classification report!

## Final prediction

knn = KNeighborsClassifier(n\_neighbors=40)  
knn.fit(X\_train,y\_train)  
pred\_40 = knn.predict(X\_test)  
  
# Evaluation with k = 40  
print("\*\*\*\*\*Classification report with k = 40 \*\*\*\*\*")  
print(classification\_report(y\_true=y\_test,y\_pred=pred\_40))  
print()  
  
# Evaluating model using confusion matrix  
print("\*\*\*\*\*Confusion matrix\*\*\*\*\*")  
print(confusion\_matrix(y\_true=y\_test,y\_pred=pred\_40))  
print()

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