[**Link to Presentation**](https://www.canva.com/design/DAGEMbB-NWg/Zx4y8J7AtJwXHrXIqNmAKA/edit?utm_content=DAGEMbB-NWg&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton)

**IB Computer Science HL - KNN Project**

**Salary Predictions Based on Personal Attributes - Margaret Jackson and Zara Tekmen**

Introduction

**Research Objective**

In this project, we are exploring measurement data for seven types of dried beans in order to better classify future unknown beans. The beans are described by the following criteria: area, perimeter, major axis length, minor axis length, aspect ratio, eccentricity, convex area, equivalent diameter, extent, solidity, roundness, and compactness, as well as 4 shape factors.

Our research objective is to use a kNN model trained on the bean dataset to classify beans based on their characteristics. The kNN model will measure similarity of attributes to predict a bean type for a given specimen. This will help bean manufacturers enforce quality standards and regular people to identify random dried beans they might have. It could also help machines to sort beans by type. To increase accuracy and avoid bias, we will build a kNN model. When a bean manufacturer needs to identify their beans, they can use our model to more accurately sort the beans. For home cooks, the type of bean is important for use in recipes. Bombay beans are the only bean used for mediterranean salads. Both barbunya and dermason beans are noted for having high amounts of Vitamin B. Sira, seker, and cali beans are specifically used for stews and soups. Barbunya and horoz beans are good for baked beans. The type of bean used can dramatically alter a recipe, so cooks need to be able to distinguish between them if they get mixed up.

**Research Questions**

1. Can we use a limited set of quantitative attributes: area, roundness, compactness, eccentricity, and solidity to categorize beans accurately?
2. Can we optimize the classification of a dried bean by comparing alternative distance algorithms? - Manhattan and Cosine

**Data Source**

The dataset lists contains measurements for over 13000 individual beans. It contains numerical values detailing various qualities of the beans like area and circumference based on a picture. Each type of bean has a string value containing the type of bean for the data. We sourced the data from kaggle.com. We chose it because we thought classifying beans by measurements could be very interesting, especially if beans are similar in size. The names of the beans are in Turkish, which was a fun surprise. This particular was selected because it is very usable for KNN and clean. Original data source: <https://www.kaggle.com/datasets/muratkokludataset/dry-bean-dataset>.

The dataset needed to be altered to run through our kNN model. Specifically, we needed to remove values to bring the data set to a manageable size. We iterated through the data and removed every third value. We also created a new column to assign each type of bean a numerical value to eliminate the categorical string value of bean type. [0→7] : Seker(1), Barbunya(2), Bombay(3), Cali(4), Horoz(5), Sira(6), and Dermason(7).

**Altered Data Source:** [Adjusted Dry Bean Dataset](https://docs.google.com/spreadsheets/d/1hrAS-hLjZJbkec5S7eMyzUgHJRq8swAsyjp0q4_JX90/edit?usp=sharing)

***Categorical Variable Histogram of All Variables***

import matplotlib.pyplot as plt

import pandas as pd

from pandas.plotting import scatter\_matrix

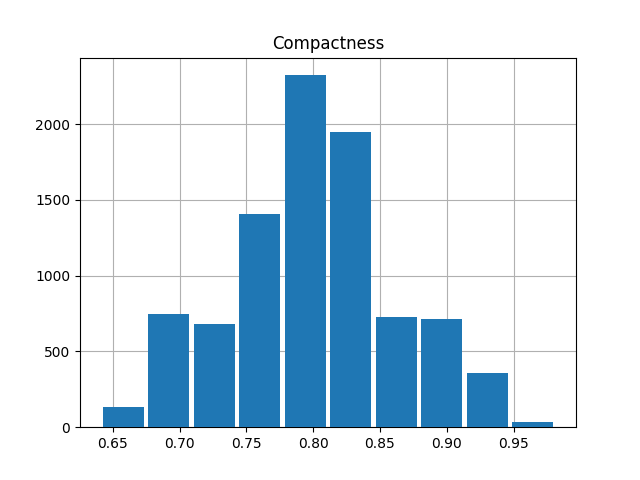
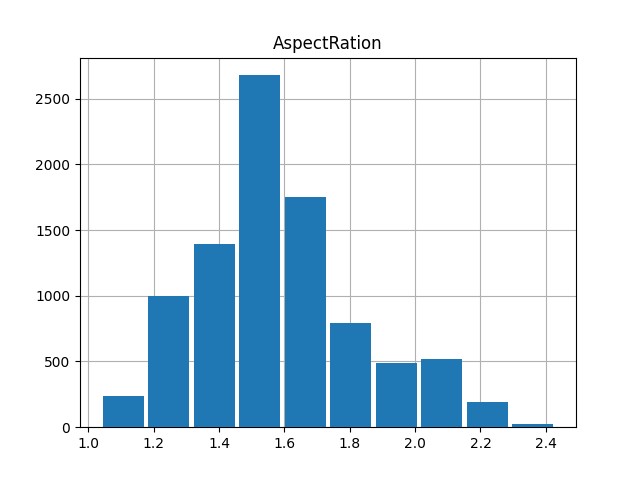
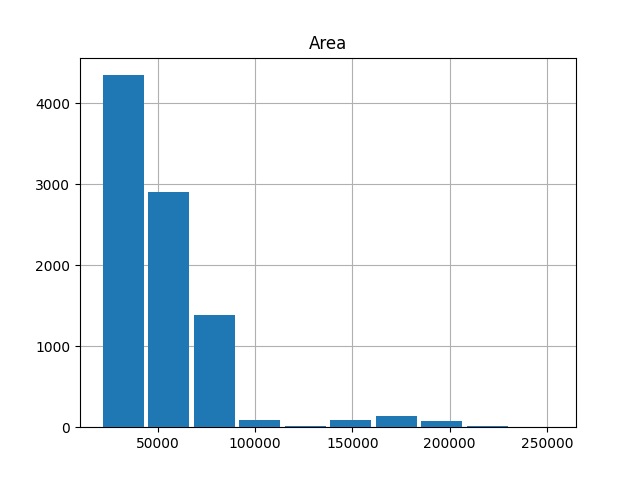
df = pd.read\_csv(r"Adjusted Dry Bean Dataset.csv")

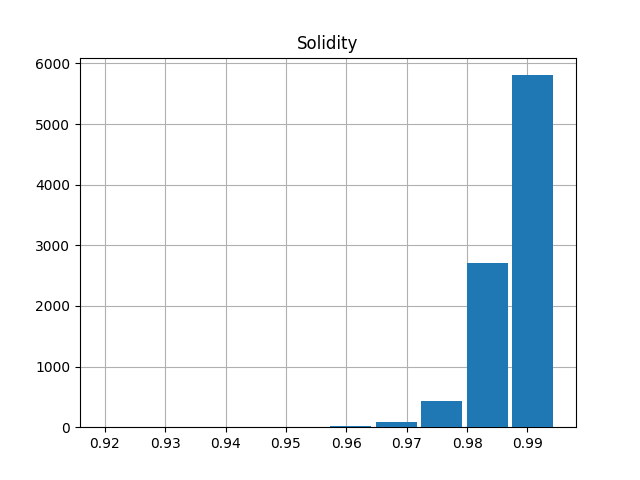
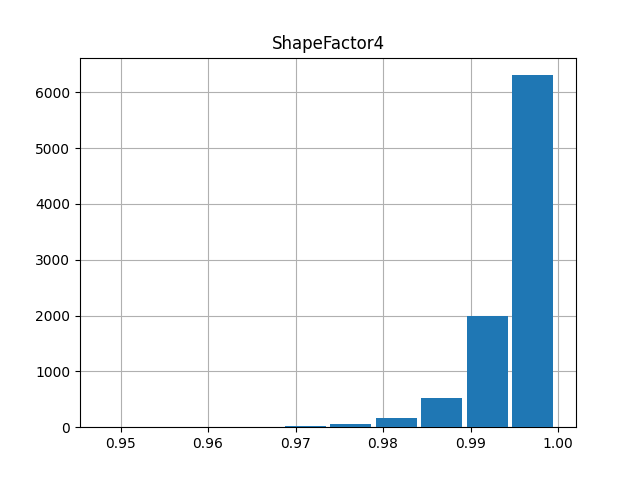
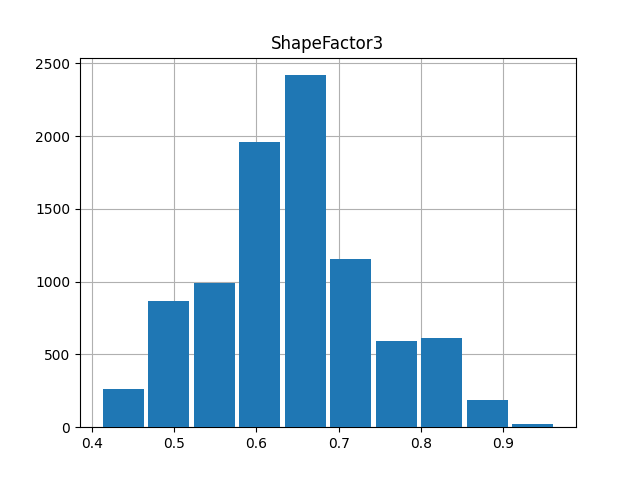
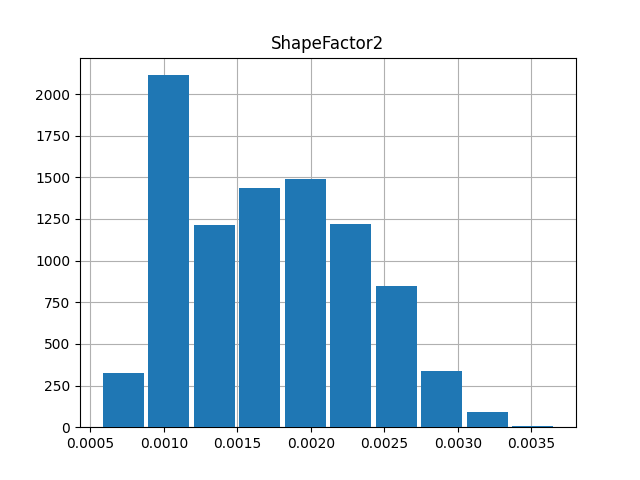
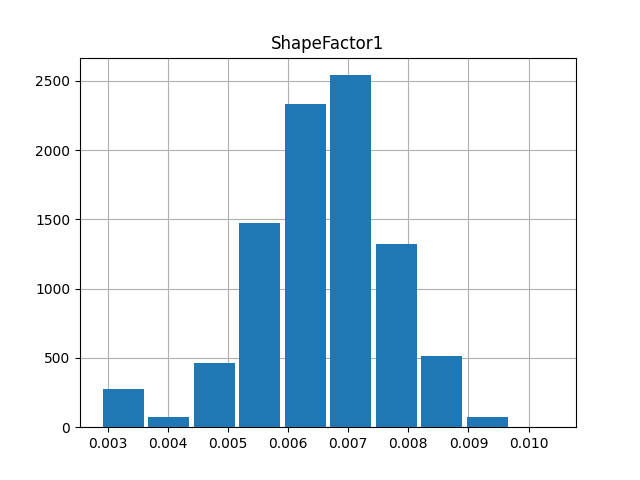
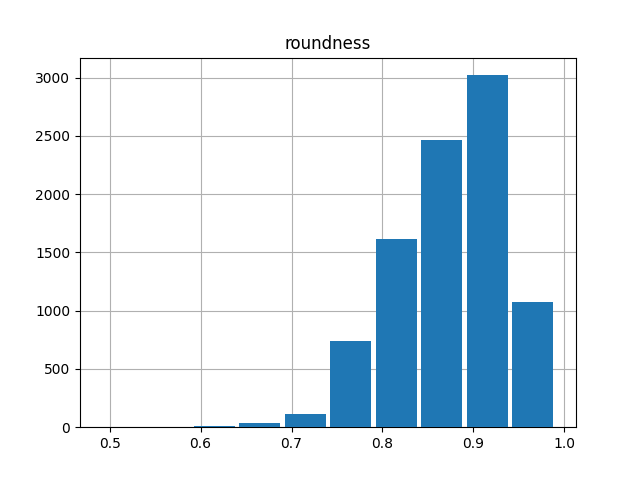
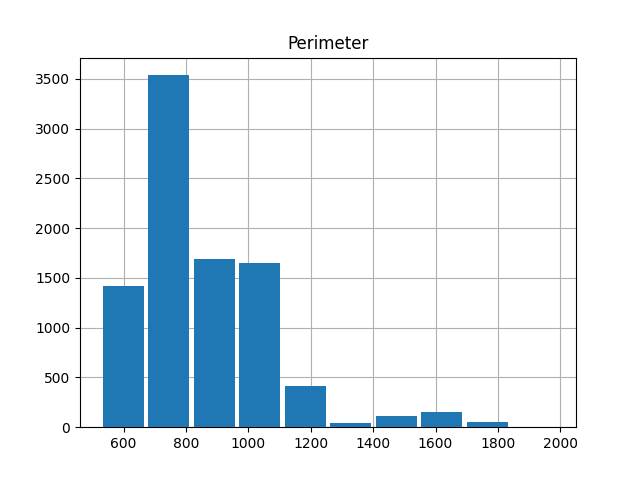
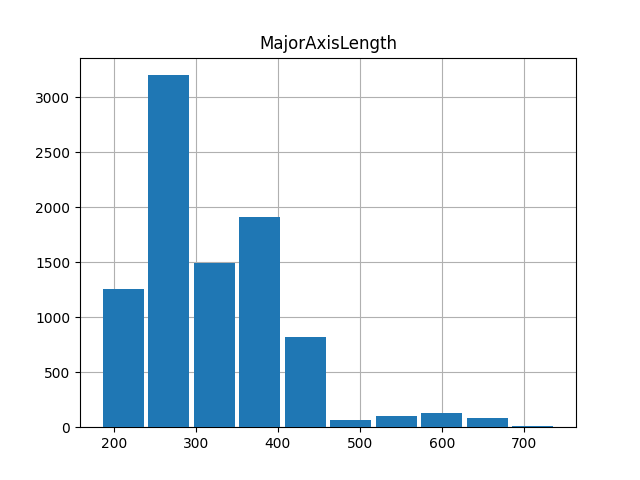
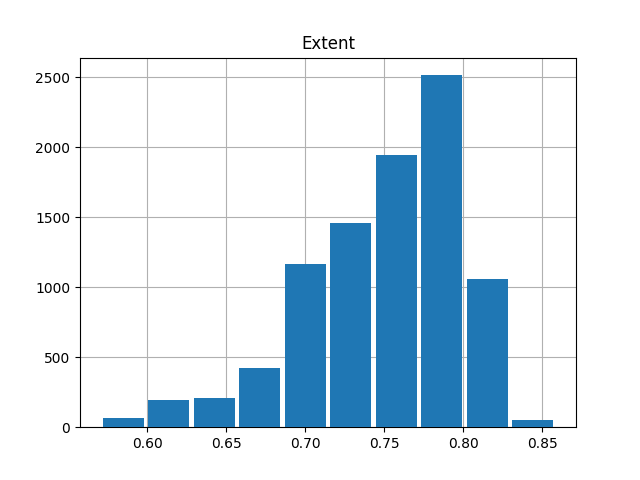
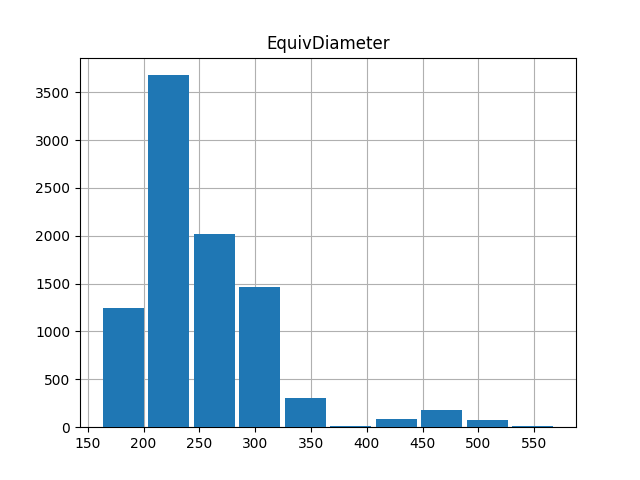
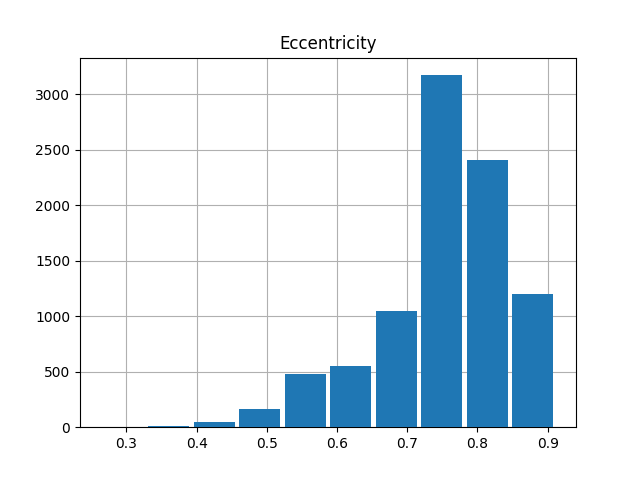
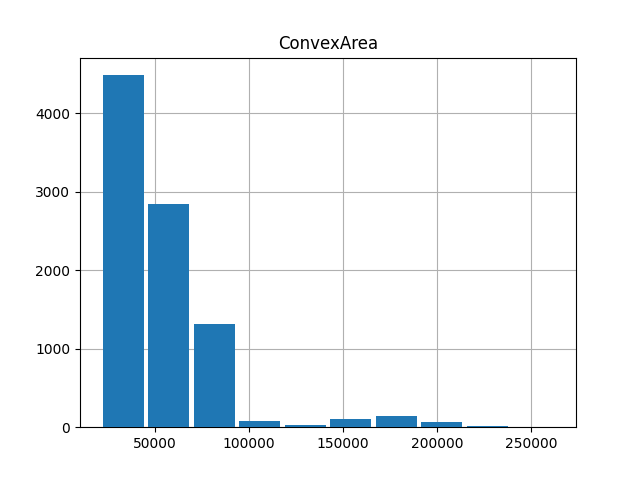
columns = df[['Eccentricity', 'Solidity', 'Compactness', 'roundness', 'ShapeFactor1']]

scatter\_matrix(columns, alpha=0.2, figsize=(6, 6), diagonal='kde')

plt.savefig(fname = "Blue\_Scatter\_Matrix.png", transparent = True)

plt.show()





***Key Attributes and How They Correlate***

******

***Analysis of Graphical Visualizations:***

* Eccentricity
  + Shows a negative correlation with 'Solidity', 'Compactness', and 'roundness', which implies that more elongated beans tend to be less solid, less compact, and less round.
* Solidity
  + Has a normal-like distribution but with a sharp peak, suggesting most beans have a similar degree of compactness.
  + Strong positive correlation with 'roundness', which is expected since more compact beans are typically rounder.
* Roundness
  + Distribution is left-skewed with most of the beans being quite round.
* Compactness
  + Shows a positive correlation with roundness. As beans are more compact, they are more round.
  + Shows no strong correlation with area or solidity. Beans remain very solid or with lots of area, regardless of compactness.
* Area
  + All 4 traits show a cluster on the low end of area with one large cluster towards the higher end. The spread of the other trait is relatively uniform. This means that all of the beans don’t show a particular correlation with area and any other trait.

# creating a pairplot by bean

import seaborn as sns

import pandas as pd

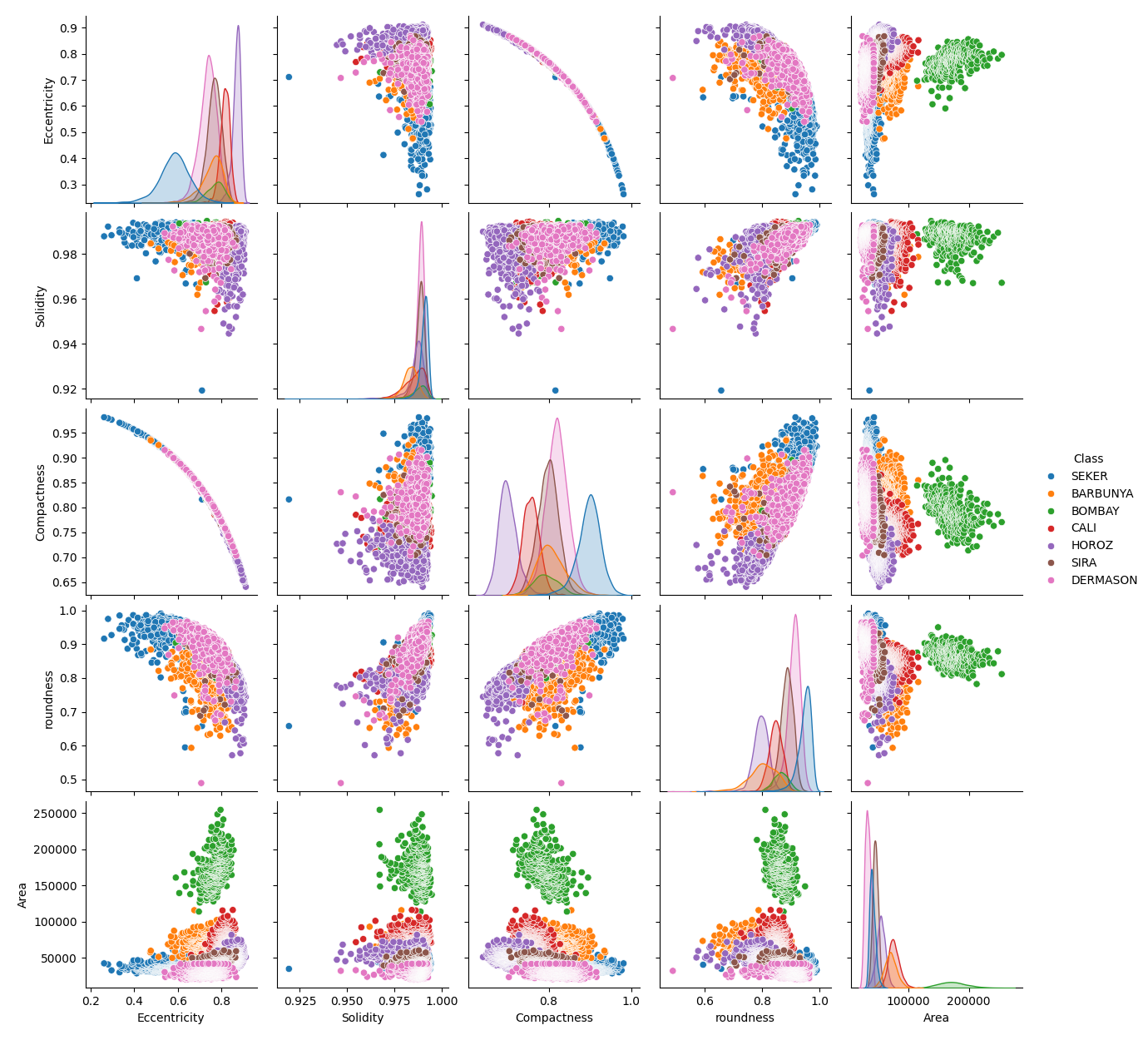
import matplotlib.pyplot as plt

df = pd.read\_csv('Adjusted Dry Bean Dataset.csv')

selected\_columns = df[[ 'Eccentricity', 'Solidity', 'Compactness', 'roundness', 'ShapeFactor1', 'Class']]sns.pairplot(selected\_columns, hue='Class')

plt.savefig(fname = "Colored\_Scatter\_Matrix.png", transparent = True)

plt.show()



In Depth Analysis of Pair Plot

* Eccentricity
  + Compared with all 4 other traits, the beans are quite eccentric.
  + Compactness forms a decreasing curve which shows that beans are less compact if they are more eccentric. This makes sense as compactness is a measure of being a circle while eccentricity is a measure of not being a circle.
  + Both solidity and roundness have strong positive correlations as well.
* Solidity
  + All 4 traits have minimal correlation but high solidity. There is one apparent outlier cluster of seker beans that are not very solid but aside from that the beans trend towards solidity.
* Compactness
  + Roundness shows a strong positive correlation as roundness and compactness are similar traits.
* Roundness
  + Roundness and eccentricity show a negative correlation due to them being opposing traits.
  + Compared with solidity, sira beans are closer to the middle of the spread as opposed to other traits where they are lower.
* Area
  + Dermason is a clear large outlier compared to other beans in size. All of the data points are in a completely removed cluster, indicating that they are much larger than the other represented beans.
  + Cali beans are consistently the smallest beans, but within the main cluster.
  + The other traits have consistent vertical spreads, but remain on the small side of area.

Overall, barbunya beans are very average and tend to cluster in the middle of the graph. Compactness vs Eccentricity provided a very interesting shape to explore further. Our traits provide good data to compare beans for classification.

STEP 2 - Preparation of the Data for kNN Implementation

Preparation of the Data for kNN Implementation Using Both Distances

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from sklearn import metrics

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

import warnings

warnings.filterwarnings('ignore')

df = pd.read\_csv('Adjusted Dry Bean Dataset.csv')

X = df[['Eccentricity', 'Solidity', 'Compactness', 'roundness', 'Area']]

y = df[['Class']]

print(X.isna().sum())

*Eccentricity 0*

*Solidity 0*

*Compactness 0*

*roundness 0*

*Area 0*

Part A - Standardize the Data

from sklearn.neighbors import KNeighborsClassifier

# Setting the number of neighbors

classifier = KNeighborsClassifier(n\_neighbors=10, metric = "manhattan

")

# Loading the training set

classifier.fit(X\_train, np.ravel(y\_train,order='C'))

# Predicting the test labels

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

print(y\_pred)

*X\_train shape: (6805, 5)*

*y\_train shape: (6805, 1)*

*X\_test shape: (2269, 5)*

*y\_test shape: (2269, 1)*

**Using the Manhattan Distance Algorithm**

import numpy as np

from sklearn.neighbors import KNeighborsClassifier

# Setting the number of neighbors

classifier = KNeighborsClassifier(n\_neighbors=10, metric = "manhattan")

# Loading the training set

classifier.fit(X\_train, np.ravel(y\_train,order='C'))

# Predicting the test labels

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

print(y\_pred)

*[[198 0 15 0 1 4 7]*

*[ 0 92 0 0 0 0 0]*

*[ 8 0 248 0 8 0 2]*

*[ 9 0 0 536 4 12 35]*

*[ 0 0 4 1 298 0 8]*

*[ 6 0 0 11 0 278 14]*

*[ 2 0 1 62 6 11 388]]*

**Evaluate the Manhattan algorithm and its model performance.**

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

# creating confusion matrix and printing the classification report

print(confusion\_matrix(y\_test, y\_pred))

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

accuracy = accuracy\_score(y\_test,y\_pred)\*100

print(accuracy)

*[[198 0 15 0 1 4 7]*

*[ 0 92 0 0 0 0 0]*

*[ 8 0 248 0 8 0 2]*

*[ 9 0 0 536 4 12 35]*

*[ 0 0 4 1 298 0 8]*

*[ 6 0 0 11 0 278 14]*

*[ 2 0 1 62 6 11 388]]*

*precision recall f1-score support*

*BARBUNYA 0.89 0.88 0.88 225*

*BOMBAY 1.00 1.00 1.00 92*

*CALI 0.93 0.93 0.93 266*

*DERMASON 0.88 0.90 0.89 596*

*HOROZ 0.94 0.96 0.95 311*

*SEKER 0.91 0.90 0.91 309*

*SIRA 0.85 0.83 0.84 470*

*accuracy 0.90 2269*

*macro avg 0.91 0.91 0.91 2269*

*weighted avg 0.90 0.90 0.90 2269*

*89.81930365799911*

Performance Improvement techniques:k values

k\_range = range(1, 40)

# Creating a Python dictionary by [] and then appending the accuracy scores

scores = []

# looping through the k range 1 to 40

for k in k\_range:

knn = KNeighborsClassifier(n\_neighbors=k)

knn.fit(X\_train,np.ravel(y\_train,order='C'))

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

# appending the accuracy scores in the dictionary named scores.

scores.append(metrics.accuracy\_score(y\_test, y\_pred))

print(scores)

# Printing the K number of neighbors and Testing Accuracy.

import matplotlib.pyplot as plt

# This command allow plots to appear within the notebook

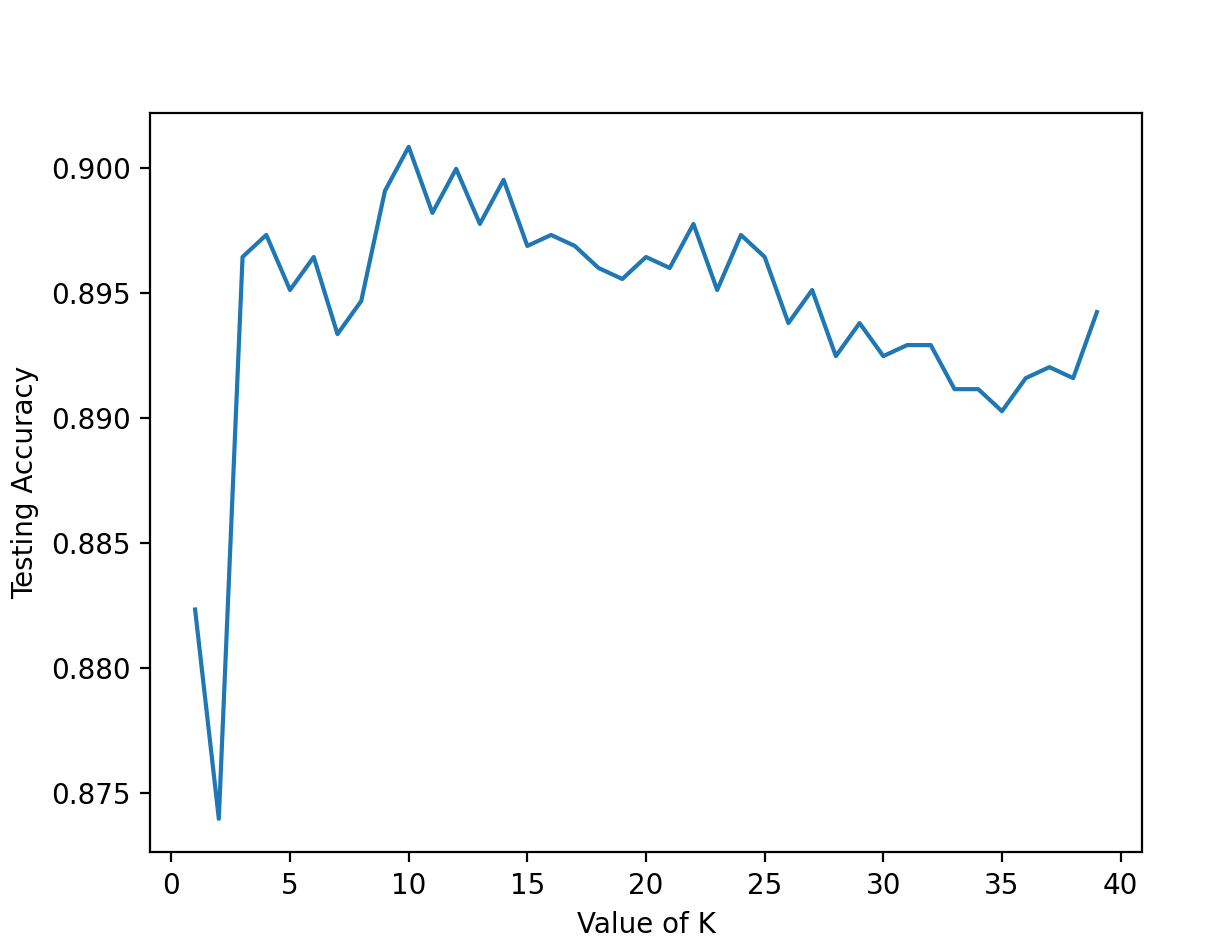
plt.plot(k\_range, scores)

plt.xlabel('Value of K')

plt.ylabel('Testing Accuracy')

plt.show()

*[0.8898193036579991, 0.8876156897311591, 0.9070074922873512, 0.9056853239312472, 0.9167033935654474, 0.9149405024239753, 0.9167033935654474, 0.9158219479947113, 0.9162626707800793, 0.9167033935654474, 0.9171441163508154, 0.9153812252093433, 0.9171441163508154, 0.9171441163508154, 0.9153812252093433, 0.9140590568532393, 0.9144997796386073, 0.9144997796386073, 0.9162626707800793, 0.9127368884971353, 0.9140590568532393, 0.9140590568532393, 0.9153812252093433, 0.9153812252093433, 0.9158219479947113, 0.9144997796386073, 0.9118554429263993, 0.9136183340678713, 0.9118554429263993, 0.9122961657117673, 0.9140590568532393, 0.9122961657117673, 0.9109739973556633, 0.9109739973556633, 0.9092111062141913, 0.9114147201410313, 0.9118554429263993, 0.9105332745702953, 0.9087703834288233]*



# Optimizing the k-nn by using Cross validation

from sklearn.model\_selection import cross\_val\_score

import numpy as np

#create a new KNN model

knn\_cv = KNeighborsClassifier(n\_neighbors=15)

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

scaler.fit(X)

X = scaler.transform(X)

#train model with cv of 10

cv\_scores = cross\_val\_score(knn\_cv, X, np.ravel(y,order='C'), cv=10)

#print each cv score (accuracy) and average them

print(cv\_scores)

print(np.mean(cv\_scores))

*[0.882327016306743, 0.873953283384751, 0.8964301454385192, 0.8973115910092552, 0.8951079770824152, 0.8964301454385192, 0.8933450859409432, 0.8946672542970472, 0.8990744821507272, 0.9008373732921993, 0.8981930365799912, 0.8999559277214632, 0.8977523137946232, 0.8995152049360952, 0.8968708682238872, 0.8973115910092552, 0.8968708682238872, 0.8959894226531512, 0.8955486998677832, 0.8964301454385192, 0.8959894226531512, 0.8977523137946232, 0.8951079770824152, 0.8973115910092552, 0.8964301454385192, 0.8937858087263112, 0.8951079770824152, 0.8924636403702071, 0.8937858087263112, 0.8924636403702071, 0.8929043631555752, 0.8929043631555752, 0.8911414720141031, 0.8911414720141031, 0.8902600264433671, 0.8915821947994711, 0.8920229175848391, 0.8915821947994711, 0.8942265315116792]*

*0.890463915022172*

from sklearn.model\_selection import cross\_val\_score

import numpy as np

#create a new KNN model

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

scaler.fit(X)

# Train with 10 fold cross validation by an outer k value ranges and nested cross validation scores.

X = scaler.transform(X)

scores = []

k\_range = range(1, 40)

for k in k\_range:

#train model with cv of 10

knn\_cv = KNeighborsClassifier(n\_neighbors=k)

cv\_scores = cross\_val\_score(knn\_cv, X, np.ravel(y,order='C'), cv=10)

#print each cv score (accuracy) and average them

print(k)

print(cv\_scores)

print(np.mean(cv\_scores))

# Prediction

knn = KNeighborsClassifier(n\_neighbors=14)

knn.fit(X\_train,np.ravel(y\_train,order='C'))

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

accuracy\_scores = metrics.accuracy\_score(y\_test, y\_pred)

print(accuracy\_scores)

*1*

*[0.70594714 0.8469163 0.90638767 0.92511013 0.92723264 0.91620728*

*0.90848953 0.88423374 0.82690187 0.69570011]*

*0.8543126393347871*

*2*

*[0.69603524 0.82599119 0.86894273 0.90638767 0.91951488 0.91951488*

*0.91951488 0.90628445 0.85777288 0.68357222]*

*0.8503531028855355*

*3*

*[0.72136564 0.8722467 0.92070485 0.93832599 0.94377067 0.94046307*

*0.9415656 0.91179713 0.87210584 0.71554576]*

*0.877789124236846*

*4*

*[0.72907489 0.86013216 0.89977974 0.9339207 0.93605292 0.94487321*

*0.94707828 0.92723264 0.88423374 0.72767365]*

*0.8790051921180829*

*5*

*[0.73568282 0.88105727 0.92070485 0.93832599 0.95259096 0.94487321*

*0.94266814 0.91951488 0.87320838 0.73428886]*

*0.8842915357304179*

*6*

*[0.73348018 0.87555066 0.90859031 0.93722467 0.94266814 0.94707828*

*0.94266814 0.93384785 0.88864388 0.74972437]*

*0.8859476465474115*

*7*

*[0.74229075 0.88325991 0.91960352 0.94493392 0.95038589 0.95148842*

*0.94597574 0.92723264 0.88092613 0.74972437]*

*0.8895821292055427*

*8*

*[0.73898678 0.87334802 0.91519824 0.93832599 0.94928335 0.95038589*

*0.94818082 0.92833517 0.89195149 0.76515987]*

*0.8899155612975924*

*9*

*[0.74449339 0.88436123 0.92290749 0.93722467 0.95148842 0.94707828*

*0.9415656 0.92061742 0.88754135 0.76185226]*

*0.8899130113799183*

*10*

*[0.74779736 0.88105727 0.9185022 0.93942731 0.94818082 0.94707828*

*0.94487321 0.92723264 0.88974642 0.77287762]*

*0.8916773115610838*

*11*

*[0.75110132 0.87885463 0.92400881 0.93832599 0.94928335 0.94818082*

*0.94377067 0.92502756 0.8831312 0.76846748]*

*0.8910151829383794*

*12*

*[0.75330396 0.87334802 0.91519824 0.93942731 0.94818082 0.95038589*

*0.94487321 0.92943771 0.88643881 0.77618523]*

*0.8916779186843398*

*13*

*[0.75881057 0.87444934 0.91299559 0.94162996 0.94818082 0.94707828*

*0.94377067 0.9261301 0.87761852 0.77618523]*

*0.8906849078872596*

*14*

*[0.75770925 0.87114537 0.91409692 0.94052863 0.94707828 0.94928335*

*0.94046307 0.92502756 0.88423374 0.7783903 ]*

*0.8907956471691056*

*15*

*[0.75991189 0.87334802 0.91740088 0.94162996 0.94928335 0.94707828*

*0.94046307 0.92502756 0.87872106 0.77177508]*

*0.890463915022172*

*16*

*[0.76651982 0.87444934 0.91629956 0.94273128 0.94597574 0.94818082*

*0.9415656 0.9261301 0.87982359 0.78059537]*

*0.8922271223814775*

*17*

*[0.76431718 0.87334802 0.91629956 0.94273128 0.94707828 0.94377067*

*0.93936053 0.92392503 0.87541345 0.77508269]*

*0.8901326685738432*

*18*

*[0.75881057 0.87004405 0.91299559 0.94162996 0.94818082 0.94597574*

*0.9415656 0.92282249 0.88423374 0.7783903 ]*

*0.8904648864193814*

*19*

*[0.76651982 0.87444934 0.91519824 0.94273128 0.95148842 0.94377067*

*0.93936053 0.92171996 0.88092613 0.77618523]*

*0.8912349615569555*

*20*

*[0.77092511 0.87334802 0.91189427 0.94273128 0.94266814 0.9415656*

*0.93715546 0.92502756 0.88643881 0.78169791]*

*0.8913452151401969*

*21*

*[0.77092511 0.87334802 0.91519824 0.9438326 0.94597574 0.94487321*

*0.93495039 0.92392503 0.87982359 0.77949283]*

*0.8912344758583508*

*22*

*[0.76321586 0.87004405 0.91409692 0.94273128 0.94266814 0.9415656*

*0.93495039 0.92171996 0.88202867 0.77949283]*

*0.8892513684558183*

*23*

*[0.77092511 0.87114537 0.91629956 0.94273128 0.94266814 0.94377067*

*0.93715546 0.92061742 0.88092613 0.7783903 ]*

*0.890462943624963*

*24*

*[0.76872247 0.87004405 0.91079295 0.93942731 0.9415656 0.94377067*

*0.93495039 0.92392503 0.88202867 0.78390298]*

*0.8899130113799181*

*25*

*[0.77092511 0.87004405 0.90969163 0.93722467 0.94046307 0.94487321*

*0.93495039 0.91730981 0.8831312 0.78280044]*

*0.8891413577218792*

*26*

*[0.76872247 0.8722467 0.91079295 0.93612335 0.93936053 0.94487321*

*0.93384785 0.92171996 0.88533627 0.7783903 ]*

*0.8891413577218794*

*27*

*[0.7654185 0.87004405 0.90969163 0.93502203 0.93936053 0.94487321*

*0.93495039 0.91730981 0.88423374 0.78280044]*

*0.8883704326117471*

*28*

*[0.76431718 0.87004405 0.90969163 0.93722467 0.93936053 0.94707828*

*0.93495039 0.92061742 0.88092613 0.7783903 ]*

*0.8882600576038545*

*29*

*[0.76211454 0.86674009 0.90969163 0.9339207 0.93936053 0.94377067*

*0.93715546 0.92061742 0.88423374 0.78169791]*

*0.8879302682513392*

*30*

*[0.76431718 0.86784141 0.90969163 0.93612335 0.94046307 0.94707828*

*0.93605292 0.91951488 0.88423374 0.78280044]*

*0.888811689794015*

*31*

*[0.76431718 0.86784141 0.91079295 0.93502203 0.93936053 0.94266814*

*0.93715546 0.92061742 0.8831312 0.78059537]*

*0.8881501682945665*

*32*

*[0.76431718 0.86674009 0.90859031 0.93612335 0.93825799 0.94377067*

*0.93715546 0.92171996 0.8831312 0.78500551]*

*0.888481171893593*

*33*

*[0.76101322 0.86784141 0.90969163 0.93612335 0.93936053 0.94266814*

*0.93715546 0.92282249 0.88423374 0.78390298]*

*0.8884812933182442*

*34*

*[0.76211454 0.86674009 0.90969163 0.93722467 0.93936053 0.94487321*

*0.93715546 0.91841235 0.88202867 0.78500551]*

*0.8882606647271102*

*35*

*[0.75991189 0.86894273 0.91079295 0.93612335 0.93936053 0.94707828*

*0.93715546 0.92061742 0.88092613 0.78500551]*

*0.8885914254768345*

*36*

*[0.76211454 0.86674009 0.91079295 0.93832599 0.94046307 0.94707828*

*0.93605292 0.91951488 0.88202867 0.78610805]*

*0.8889219433772567*

*37*

*[0.75881057 0.87004405 0.90969163 0.93722467 0.94046307 0.94707828*

*0.93825799 0.91841235 0.87872106 0.78721058]*

*0.8885914254768347*

*38*

*[0.7654185 0.86784141 0.90859031 0.93612335 0.93715546 0.94707828*

*0.93825799 0.92061742 0.87872106 0.79272326]*

*0.889252704126981*

*39*

*[0.76101322 0.87114537 0.90748899 0.93612335 0.93715546 0.94487321*

*0.93715546 0.91951488 0.87872106 0.79162073]*

*0.8884811718935932*

*0.8995152049360952*

Import numpy as np

import pandas as pd

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

from sklearn.ensemble import RandomForestClassifier

import matplotlib.pyplot as plt

# Load the dataset

df = pd.read\_csv('Adjusted Dry Bean Dataset.csv')

# Exclude the 'Class' column which contains string values and the first column if it's an index

X = df.iloc[:, 3:13] # Adjust the range if needed

y = df['Class Number'] # Make sure this column is numeric

print(X)

print(y)

# Splitting the dataset into the Training set and Test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=12)

# Random Forest to get feature importance for classification

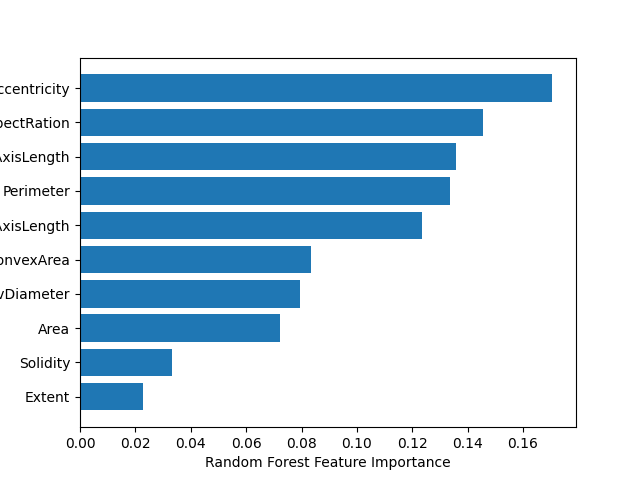
rf = RandomForestClassifier(n\_estimators=100, random\_state=12)

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

# Get the feature importances and sort them

sorted\_idx = rf.feature\_importances\_.argsort()

print("Feature importances:", rf.feature\_importances\_)



# Print sorted indices (i.e., the indices of the features in ascending order of importance)

print("Sorted indices of features:", sorted\_idx)

# Plotting feature importances

plt.barh(X\_train.columns[sorted\_idx], rf.feature\_importances\_[sorted\_idx])

plt.xlabel("Random Forest Feature Importance")

plt.show()

# Using a subset of predictor feature variables for the classification:

from sklearn.model\_selection import cross\_val\_score

import numpy as np

from sklearn.neighbors import KNeighborsClassifier

#create a new KNN model

from sklearn.preprocessing import StandardScaler

from sklearn import metrics

data\_Optimize = pd.read\_csv('Adjusted Dry Bean DataSet.csv')

# Remove spaces in the column names

data\_Optimize.columns = data\_Optimize.columns.to\_series().apply(lambda x: x.strip())

X = data\_Optimize[['Area','roundness','Compactness','Solidity','Eccentricity',]]

y = df[['Class']]

scaler = StandardScaler()

scaler.fit(X)

X = scaler.transform(X)

scores = []

k\_range = range(1, 40)

for k in k\_range:

#train model with cv of 10

knn\_cv = KNeighborsClassifier(n\_neighbors=k)

cv\_scores = cross\_val\_score(knn\_cv, X, np.ravel(y,order='C'), cv=10)

#print each cv score (accuracy) and average them

print(k)

print(cv\_scores)

print(np.mean(cv\_scores))

knn = KNeighborsClassifier(n\_neighbors=15)

knn.fit(X\_train,np.ravel(y\_train,order='C'))

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

accuracy\_scores = metrics.accuracy\_score(y\_test, y\_pred)

print(accuracy\_scores)

*1*

*0.08329431 0.07934374 0.02285139 0.03326447]*

*Sorted indices of features: [8 9 0 7 6 2 1 3 4 5]*

*1*

*[0.70594714 0.8469163 0.90638767 0.92511013 0.92723264 0.91620728*

*0.90848953 0.88423374 0.82690187 0.69570011]*

*0.8543126393347871*

*2*

*[0.69603524 0.82599119 0.86894273 0.90638767 0.91951488 0.91951488*

*0.91951488 0.90628445 0.85777288 0.68357222]*

*0.8503531028855355*

*3*

*[0.72136564 0.8722467 0.92070485 0.93832599 0.94377067 0.94046307*

*0.9415656 0.91179713 0.87210584 0.71554576]*

*0.877789124236846*

*4*

*[0.72907489 0.86013216 0.89977974 0.9339207 0.93605292 0.94487321*

*0.94707828 0.92723264 0.88423374 0.72767365]*

*0.8790051921180829*

*5*

*[0.73568282 0.88105727 0.92070485 0.93832599 0.95259096 0.94487321*

*0.94266814 0.91951488 0.87320838 0.73428886]*

*0.8842915357304179*

*6*

*[0.73348018 0.87555066 0.90859031 0.93722467 0.94266814 0.94707828*

*0.94266814 0.93384785 0.88864388 0.74972437]*

*0.8859476465474115*

*7*

*[0.74229075 0.88325991 0.91960352 0.94493392 0.95038589 0.95148842*

*0.94597574 0.92723264 0.88092613 0.74972437]*

*0.8895821292055427*

*8*

*[0.73898678 0.87334802 0.91519824 0.93832599 0.94928335 0.95038589*

*0.94818082 0.92833517 0.89195149 0.76515987]*

*0.8899155612975924*

*9*

*[0.74449339 0.88436123 0.92290749 0.93722467 0.95148842 0.94707828*

*0.9415656 0.92061742 0.88754135 0.76185226]*

*0.8899130113799183*

*10*

*[0.74779736 0.88105727 0.9185022 0.93942731 0.94818082 0.94707828*

*0.94487321 0.92723264 0.88974642 0.77287762]*

*0.8916773115610838*

*11*

*[0.75110132 0.87885463 0.92400881 0.93832599 0.94928335 0.94818082*

*0.94377067 0.92502756 0.8831312 0.76846748]*

*0.8910151829383794*

*12*

*[0.75330396 0.87334802 0.91519824 0.93942731 0.94818082 0.95038589*

*0.94487321 0.92943771 0.88643881 0.77618523]*

*0.8916779186843398*

*13*

*[0.75881057 0.87444934 0.91299559 0.94162996 0.94818082 0.94707828*

*0.94377067 0.9261301 0.87761852 0.77618523]*

*0.8906849078872596*

*14*

*[0.75770925 0.87114537 0.91409692 0.94052863 0.94707828 0.94928335*

*0.94046307 0.92502756 0.88423374 0.7783903 ]*

*0.8907956471691056*

*15*

*[0.75991189 0.87334802 0.91740088 0.94162996 0.94928335 0.94707828*

*0.94046307 0.92502756 0.87872106 0.77177508]*

*0.890463915022172*

*16*

*[0.76651982 0.87444934 0.91629956 0.94273128 0.94597574 0.94818082*

*0.9415656 0.9261301 0.87982359 0.78059537]*

*0.8922271223814775*

*17*

*[0.76431718 0.87334802 0.91629956 0.94273128 0.94707828 0.94377067*

*0.93936053 0.92392503 0.87541345 0.77508269]*

*0.8901326685738432*

*18*

*[0.75881057 0.87004405 0.91299559 0.94162996 0.94818082 0.94597574*

*0.9415656 0.92282249 0.88423374 0.7783903 ]*

*0.8904648864193814*

*19*

*[0.76651982 0.87444934 0.91519824 0.94273128 0.95148842 0.94377067*

*0.93936053 0.92171996 0.88092613 0.77618523]*

*0.8912349615569555*

*20*

*[0.77092511 0.87334802 0.91189427 0.94273128 0.94266814 0.9415656*

*0.93715546 0.92502756 0.88643881 0.78169791]*

*0.8913452151401969*

*21*

*[0.77092511 0.87334802 0.91519824 0.9438326 0.94597574 0.94487321*

*0.93495039 0.92392503 0.87982359 0.77949283]*

*0.8912344758583508*

*22*

*[0.76321586 0.87004405 0.91409692 0.94273128 0.94266814 0.9415656*

*0.93495039 0.92171996 0.88202867 0.77949283]*

*0.8892513684558183*

*23*

*[0.77092511 0.87114537 0.91629956 0.94273128 0.94266814 0.94377067*

*0.93715546 0.92061742 0.88092613 0.7783903 ]*

*0.890462943624963*

*24*

*[0.76872247 0.87004405 0.91079295 0.93942731 0.9415656 0.94377067*

*0.93495039 0.92392503 0.88202867 0.78390298]*

*0.8899130113799181*

*25*

*[0.77092511 0.87004405 0.90969163 0.93722467 0.94046307 0.94487321*

*0.93495039 0.91730981 0.8831312 0.78280044]*

*0.8891413577218792*

*26*

*[0.76872247 0.8722467 0.91079295 0.93612335 0.93936053 0.94487321*

*0.93384785 0.92171996 0.88533627 0.7783903 ]*

*0.8891413577218794*

*27*

*[0.7654185 0.87004405 0.90969163 0.93502203 0.93936053 0.94487321*

*0.93495039 0.91730981 0.88423374 0.78280044]*

*0.8883704326117471*

*28*

*[0.76431718 0.87004405 0.90969163 0.93722467 0.93936053 0.94707828*

*0.93495039 0.92061742 0.88092613 0.7783903 ]*

*0.8882600576038545*

*29*

*[0.76211454 0.86674009 0.90969163 0.9339207 0.93936053 0.94377067*

*0.93715546 0.92061742 0.88423374 0.78169791]*

*0.8879302682513392*

*30*

*[0.76431718 0.86784141 0.90969163 0.93612335 0.94046307 0.94707828*

*0.93605292 0.91951488 0.88423374 0.78280044]*

*0.888811689794015*

*31*

*[0.76431718 0.86784141 0.91079295 0.93502203 0.93936053 0.94266814*

*0.93715546 0.92061742 0.8831312 0.78059537]*

*0.8881501682945665*

*32*

*[0.76431718 0.86674009 0.90859031 0.93612335 0.93825799 0.94377067*

*0.93715546 0.92171996 0.8831312 0.78500551]*

*0.888481171893593*

*33*

*[0.76101322 0.86784141 0.90969163 0.93612335 0.93936053 0.94266814*

*0.93715546 0.92282249 0.88423374 0.78390298]*

*0.8884812933182442*

*34*

*[0.76211454 0.86674009 0.90969163 0.93722467 0.93936053 0.94487321*

*0.93715546 0.91841235 0.88202867 0.78500551]*

*0.8882606647271102*

*35*

*[0.75991189 0.86894273 0.91079295 0.93612335 0.93936053 0.94707828*

*0.93715546 0.92061742 0.88092613 0.78500551]*

*0.8885914254768345*

*36*

*[0.76211454 0.86674009 0.91079295 0.93832599 0.94046307 0.94707828*

*0.93605292 0.91951488 0.88202867 0.78610805]*

*0.8889219433772567*

*37*

*[0.75881057 0.87004405 0.90969163 0.93722467 0.94046307 0.94707828*

*0.93825799 0.91841235 0.87872106 0.78721058]*

*0.8885914254768347*

*38*

*[0.7654185 0.86784141 0.90859031 0.93612335 0.93715546 0.94707828*

*0.93825799 0.92061742 0.87872106 0.79272326]*

*0.889252704126981*

*39*

*[0.76101322 0.87114537 0.90748899 0.93612335 0.93715546 0.94487321*

*0.93715546 0.91951488 0.87872106 0.79162073]*

*0.8884811718935932*

*0.6637285147642134*

from sklearn.neighbors import KNeighborsClassifier

test\_scores = []

train\_scores = []

for i in range(1,15):

knn = KNeighborsClassifier(i)

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

train\_scores.append(knn.score(X\_train,y\_train))

test\_scores.append(knn.score(X\_test,y\_test))

## Training Evaluation

max\_train\_score = max(train\_scores)

# # Store the max train test score index by enumerating through all the scores.

train\_scores\_ind = [i for i, v in enumerate(train\_scores) if v == max\_train\_score]

# Store the max score in the first curly parenthesis and the indices in the second.

# The lambda function takes the index starting at zero therefore one is added to get the k value.

print('Max train score {} % and k = {}'.format(max\_train\_score\*100,list(map(lambda x: x+1, train\_scores\_ind))))

## Testing Evaluation

max\_test\_score = max(test\_scores)

test\_scores\_ind = [i for i, v in enumerate(test\_scores) if v == max\_test\_score]

print('Max test score {} % and k = {}'.format(max\_test\_score\*100,list(map(lambda x: x+1, test\_scores\_ind))))

## Train Test Evaluation by comparative graph.

plt.figure(figsize=(12,5))

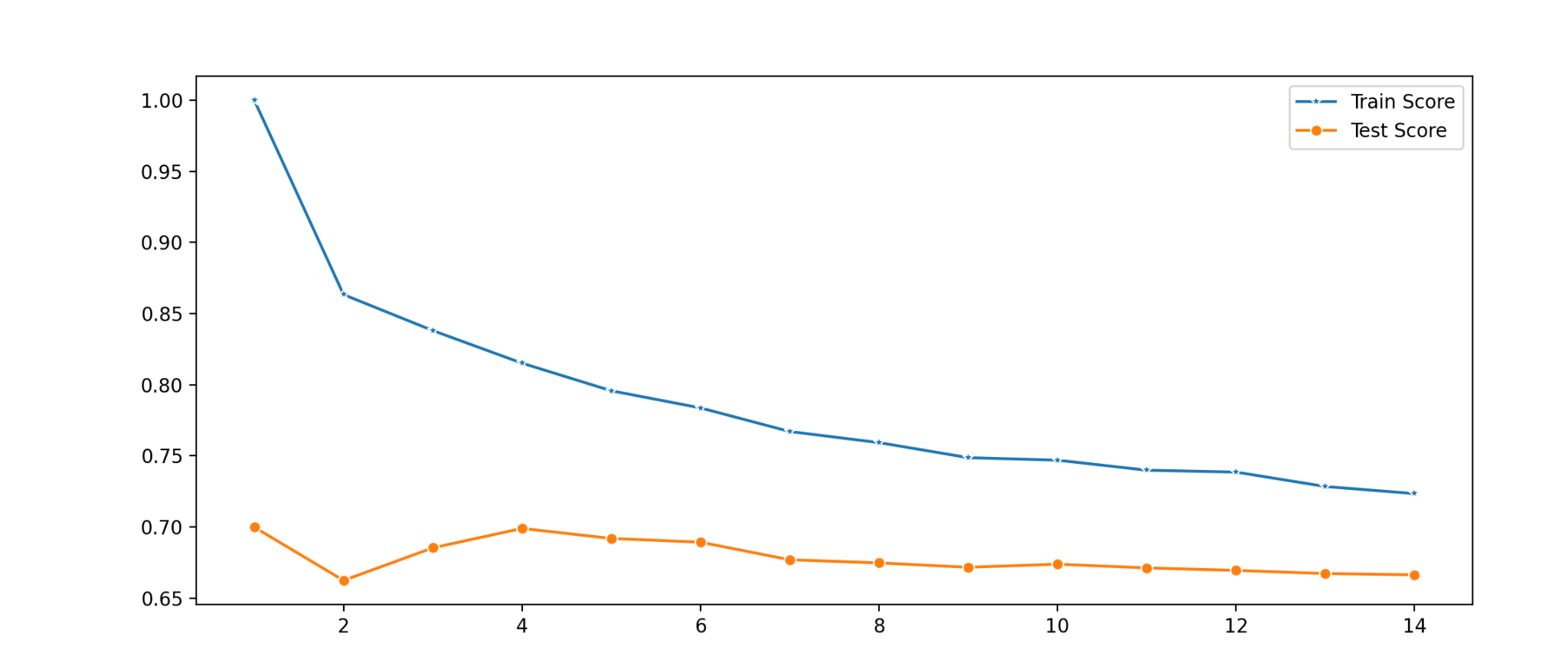
p = sns.lineplot(x=range(1,15), y=train\_scores, marker='\*', label='Train Score')

p = sns.lineplot(x=range(1,15), y=test\_scores, marker='o', label='Test Score')

plt.show()

*Max train score 100.0 % and k = [1]*

*Max test score 69.98677831643896 % and k = [1]*



## Error Rate Graph

# Create an empty dictionary to collect errors across the different k-values

error = []

# Iterate through k=1 to 40 and run the classifier.Predict and append the error for each iteration.

for i in range(1, 40):

knn = KNeighborsClassifier(n\_neighbors=i)

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

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

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

# Create a plot of Mean error versus kvalue.

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

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

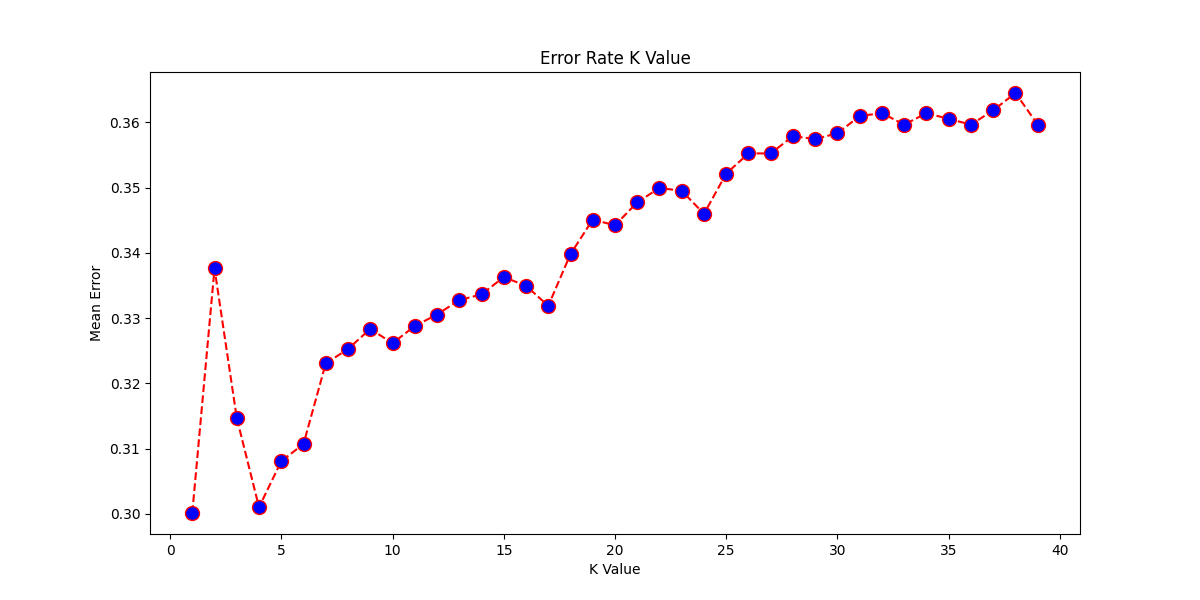
markerfacecolor='blue', markersize=10)

plt.title('Error Rate K Value')

plt.xlabel('K Value')

plt.ylabel('Mean Error')

plt.show()



**Comparing Manhattan and Cosine Distance Algorithm**

Code remained mostly the same apart from the adjusted line:

classifier = KNeighborsClassifier(n\_neighbors=10, metric = "manhattan")]

**vs**

classifier = KNeighborsClassifier(n\_neighbors=10, metric = "cosine")

***Manhattan Cosine***

| *[[180 0 17 0 2 2 5]*  *[ 0 75 0 0 0 0 0]*  *[ 7 0 263 0 4 1 1]*  *[ 0 0 0 547 3 10 33]*  *[ 0 0 5 3 302 0 9]*  *[ 6 0 0 11 1 314 10]*  *[ 3 0 1 64 3 6 381]]* | *[[203 0 10 0 0 2 4]*  *[ 0 105 0 0 0 0 0]*  *[ 9 0 258 0 5 0 5]*  *[ 3 0 0 556 2 11 38]*  *[ 2 0 8 1 265 0 7]*  *[ 6 0 0 10 1 316 15]*  *[ 3 0 0 57 4 6 357]]* |
| --- | --- |
| *class precision recall f1-score support*  *BARBUNYA 0.92 0.87 0.90 206*  *BOMBAY 1.00 1.00 1.00 75*  *CALI 0.92 0.95 0.94 276*  *DERMASON 0.88 0.92 0.90 593*  *HOROZ 0.96 0.95 0.95 319*  *SEKER 0.94 0.92 0.93 342*  *SIRA 0.87 0.83 0.85 458*  *accuracy 0.91 2269*  *macro avg 0.93 0.92 0.92 2269*  *weighted avg 0.91 0.91 0.91 2269* | *class precision recall f1-score support*  *BARBUNYA 0.90 0.93 0.91 219*  *BOMBAY 1.00 1.00 1.00 105*  *CALI 0.93 0.93 0.93 277*  *DERMASON 0.89 0.91 0.90 610*  *HOROZ 0.96 0.94 0.95 283*  *SEKER 0.94 0.91 0.93 348*  *SIRA 0.84 0.84 0.84 427*  *accuracy 0.91 2269*  *macro avg 0.92 0.92 0.92 2269*  *weighted avg 0.91 0.91 0.91 2269* |
| ***Manhattan Average: 90.87703834288233*** | ***Cosine Average: 90.78889378580872*** |
|  |  |

Analysis of the Accuracy:

The Manhattan Algorithm turned out to be slightly more accurate, however the difference is a mere 0.088%. We assume that the large size of the dataset helped prevent outliers that would affect the more complicated algorithm.

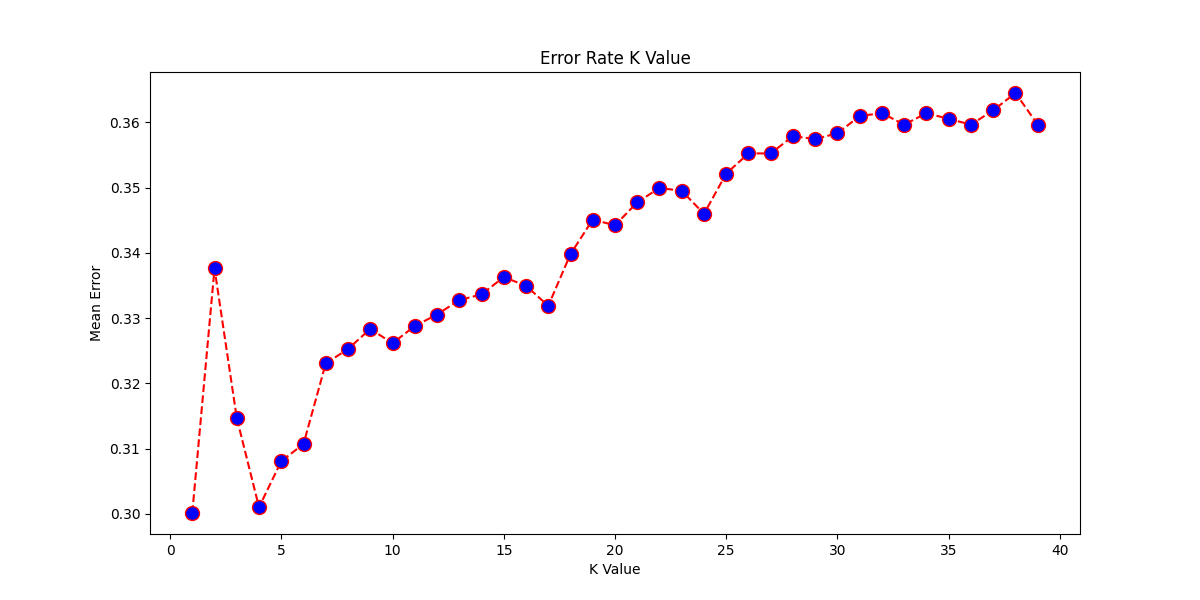
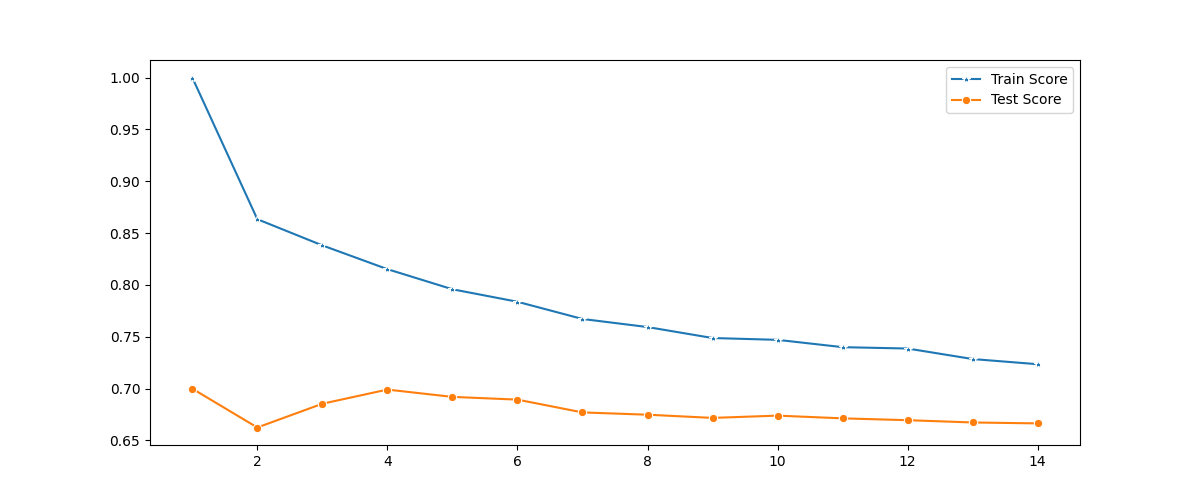
Looking at Multiple K-Values:

Both graphs have their highest accuracies between k=5 and k=20. However, Cosine accuracy drops off much more steeply with less increasing jumps than Manhattan. Manhattan stays more accurate longer but dips lower as k increases long-range than Cosine.

Manhattan

Max train score 100.0 % and k = [1]

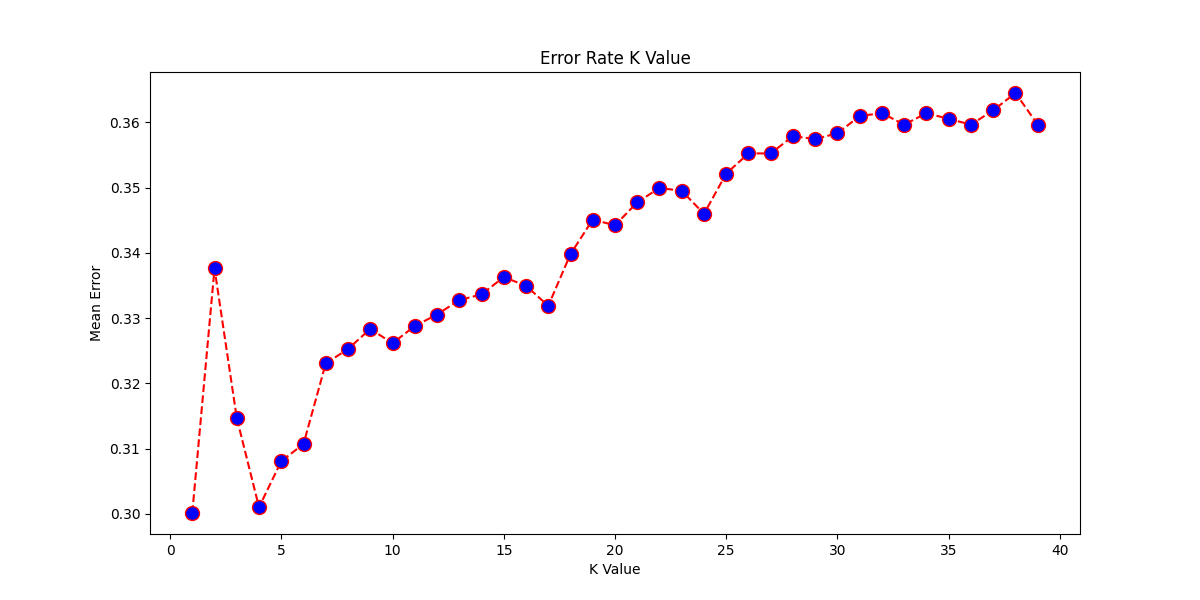
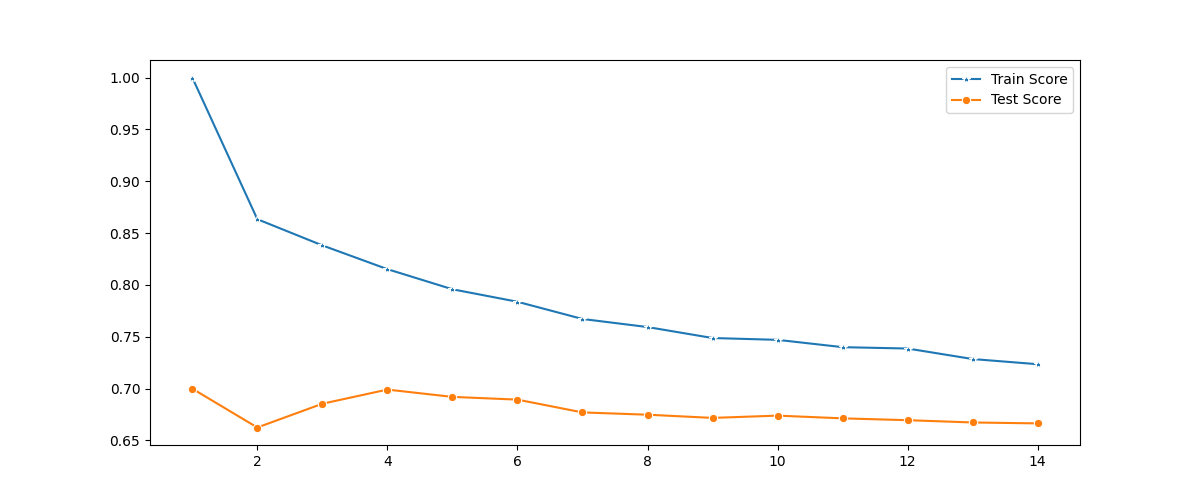
Max test score 69.98677831643896 % and k = [1]



Cosine

Max train score 100.0 % and k = [1]

Max test score 69.98677831643896 % and k = [1]



Oddly, the graphs have the same train and test score lines. They show very similar, if not identical results. The error rate by k-value graph is almost identical as well. Given the graphs are functionally identical, we cannot recommend the use of one algorithm over the other.

Conclusion

In this project we explored seven types of turkish beans and how different attributes allow us to classify them. We specifically looked at area, solidity, eccentricity, compactness, and roundness to categorize the beans into seven types: seker, sira, bombay, dermason, horoz, barbunya, and cali.

Our research objective is to use a kNN model trained on a sample dataset to classify which type of bean is being studied and if there was any correlation between these attributes to the type.

The kNN models will measure similarity to classify the observed beans in the test dataset. As we had a wealth of traits to compare, we chose to use ones that told us things about the shape and size of the bean. The easiest way to classify a food is by shape and size, so we focused on those traits. We had distinct profiles of the beans in area, as well as eccentricity vs compactness. The easiest trait to classify beans was by eccentricity as some beans are more circular than others. We would have though that area was a strong predictor of type, but it was not. The two different kNN algorithms had functionally identical results so we would recommend the Manhattan Algorithm to classify beans due to its lower cost and speed of calculation, as well as having a higher accuracy rate than the more complicated and expensive Cosine Distance.

Research Questions:

We sought to answer the following questions:

1. Can we use a limited set of quantitative attributes: area, roundness, compactness, eccentricity, and solidity to categorize beans accurately?
2. Can we optimize the classification of a dried bean by comparing alternative distance algorithms? - Manhattan and Cosine

To answer these questions, we needed to change the bean type variable from categorical to numerical by assigning a numeric value to each type. This was used as a target variable for the kNN model. We had a 2:1 split of training to testing data. The model was run in sequence to optimize for k, finding that k=1 was the best accuracy score for Manhattan and for Cosine distance, without a risk of overfitting. K=4 was very comparable to k=1, but still had a lower accuracy score.

The classification report measured the quality of predictions from the k-model and showed a precision (% of correct predictions) of 0.91 for both Manhattan and Cosine distance.

Research Application:

We could implement the findings from this study by sampling “volunteer” beans (plants that grow without being planted) from home gardeners around Turkey to help them identify what plant has sprouted. This would allow them to avoid any beans they don’t like or use the beans for their best culinary purpose.

Critique:

An improvement to the model would be to use more attributes as the foundation of the comparisons, or to use different traits. Our method of choosing traits was somewhat subjective as we chose traits that had interesting distributions and seemed like they would be well suited for categorizing beans. A further study could use all of the different combinations of traits to find the most accurate one.