KNN

February 14, 2024

1 KNN Implementation

1.0.1 Jack DeGesero

1.0.2 Prof Shang

1.0.3 CSIT455

In this project, I manually implement the machine learning classifier known as K nearest neighbors. This model predicts characteristics of data objects based on how similar each object is to other objects. To measure similarity and decide the prediction, various distance formulas, e.g. euclidean and manhattan, are employed to find the most similar objects. Based on the value of K, which is a defined parameter, we can see what K of the 'nearest neighbors' data values are, and to take the most frequent value for the predicted value. After predicting the data, we can assess how accurate it is to the observed data.

1.1 Importing and Loading Data

Here I use a few packages to help me implement the model. I use numpy for series objects (e.g. an array with listed indicies), math (for basic functions), stats (for basic stat functions), pandas (for dataframe objects and easy CSV manipulation), and MatPlotLib and Seaborn (for graphing).

[1]:
```
import numpy as py #series
import math #for sqrt
import statistics #for mode
import pandas as pd #dataframes
import matplotlib.pyplot as plt #graphing
import seaborn as sns #graphing
```

[2]:
```
df = pd.read_csv('letter-recognition.csv', header=None) #load csv as dataframe
```

[3]:
```
df #20,000 rows by 17 columns, 0 or the column which indicates letter will be␣
 ↪our class attribute
```

[3]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	T	2	8	3	5	1	8	13	0	6	6	10	8	0	8	0	8
1	I	5	12	3	7	2	10	5	5	4	13	3	9	2	8	4	10
2	D	4	11	6	8	6	10	6	2	6	10	3	7	3	7	3	9
3	N	7	11	6	6	3	5	9	4	6	4	4	10	6	10	2	8

```
4      G   2   1   3   1   1   8   6   6   6   6   5   9   1   7   5  10
…    ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..
19995  D   2   2   3   3   2   7   7   7   6   6   6   4   2   8   3   7
19996  C   7  10   8   8   4   4   8   6   9  12   9  13   2   9   3   7
19997  T   6   9   6   7   5   6  11   3   7  11   9   5   2  12   2   4
19998  S   2   3   4   2   1   8   7   2   6  10   6   8   1   9   5   8
19999  A   4   9   6   6   2   9   5   3   1   8   1   8   2   7   2   8

[20000 rows x 17 columns]
```

1.2 Data Cleaning & Preprocessing

Here we check for any missing values. Then we make the partition of our data into a training set and testing set. We will use the training values to build the model, then use the test set to evaluate the efficacy.

[4]:
```
df.isna().sum()
```

[4]:
```
0     0
1     0
2     0
3     0
4     0
5     0
6     0
7     0
8     0
9     0
10    0
11    0
12    0
13    0
14    0
15    0
16    0
dtype: int64
```

[5]:
```
#70/30 test-train split

trainRows = int(df.shape[0] * .7) #value of where to split data (index 70%)

#slice dataframe
xtrain, xtest, ytrain, ytest = df.iloc[0:trainRows].iloc[:,1:].
  ↪reset_index(drop=True), df.iloc[trainRows:].iloc[:,1:].
  ↪reset_index(drop=True),df.iloc[0:trainRows].iloc[:,0].
  ↪reset_index(drop=True),df.iloc[trainRows:].iloc[:,0].reset_index(drop=True)
```

[6]:
```
xtrain
```

[6]:
```
       1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
0       2   8   3   5   1   8  13   0   6   6  10   8   0   8   0   8
1       5  12   3   7   2  10   5   5   4  13   3   9   2   8   4  10
2       4  11   6   8   6  10   6   2   6  10   3   7   3   7   3   9
3       7  11   6   6   3   5   9   4   6   4   4  10   6  10   2   8
4       2   1   3   1   1   8   6   6   6   6   5   9   1   7   5  10
…      ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..
13995   2   3   2   2   1   5  10   1   6  10   9   6   1  11   2   5
13996   5   7   5   5   3   6  11   2   6  11   9   5   2  11   2   5
13997   3   8   5   6   1   6   8   4   3   8  14   8   3  10   0   8
13998   4   8   4   6   4   5   7   9   6   6   5   5   2   8   3   8
13999   3   1   5   1   2   6  11   3   2   7   9   9   6  11   0   8

[14000 rows x 16 columns]
```

[7]:
```
xtest
```

[7]:
```
       1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
0       5   9   7   6   6   4   7   3   4  10  10  10   7   5   2   7
1       3   6   5   4   3   8   7   4   9   6   6   8   3   8   6   8
2       5  11   9   8  12   7   5   3   2   7   5   8  14   6   4   7
3       6  10   5   6   3  10   6   4   6  14   3   6   2   9   4  11
4       5   8   7   6   6   9   9   5   4   7   5   7   3   8   9   5
…      ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..
5995    2   2   3   3   2   7   7   7   6   6   6   4   2   8   3   7
5996    7  10   8   8   4   4   8   6   9  12   9  13   2   9   3   7
5997    6   9   6   7   5   6  11   3   7  11   9   5   2  12   2   4
5998    2   3   4   2   1   8   7   2   6  10   6   8   1   9   5   8
5999    4   9   6   6   2   9   5   3   1   8   1   8   2   7   2   8

[6000 rows x 16 columns]
```

[8]:
```
ytrain
```

[8]:
```
0        T
1        I
2        D
3        N
4        G
        ..
13995    Y
13996    T
13997    V
13998    D
13999    W
Name: 0, Length: 14000, dtype: object
```

[9]:
```
ytest
```

[9]:
```
0       M
1       X
2       M
3       I
4       Z
       ..
5995    D
5996    C
5997    T
5998    S
5999    A
Name: 0, Length: 6000, dtype: object
```

1.3 Model Construction and Training

Here we build the model to measure each row of the testing data's distance from the training data. After measuring distance, it will order the distances and take the K nearest neighbors values to vote. The value with the most votes (mode) will be the predicted value for that instance. After doing this for the entire test set, we get an array of predicted values for our testing set.

[10]:
```
def kNN(xtr, xt, ytr, k=3, method="Euclidean"):

    predictions = []

    #iterate through xt
    for index, row in xt.iterrows():

        #calculate the distances from each feature in xt to xtr (based on the␣
↪desired method), save in distances
        if method=="Euclidean":
            distances = py.sqrt(((xtr - row) ** 2).sum(axis=1))
        elif method=="Manhattan":
            distances = py.abs((xtr - row).sum(axis=1))
        else:
            print('Method Undefined')

        kNearest = ytr.iloc[distances.argsort()[:k]] #based on distances, sort␣
↪the array and find K nearest neighbors

        predictions.append(kNearest.mode()[0]) #take the mode (most frequent␣
↪value) of the k nearest, add to the prediction array

    return(pd.Series(predictions))
```

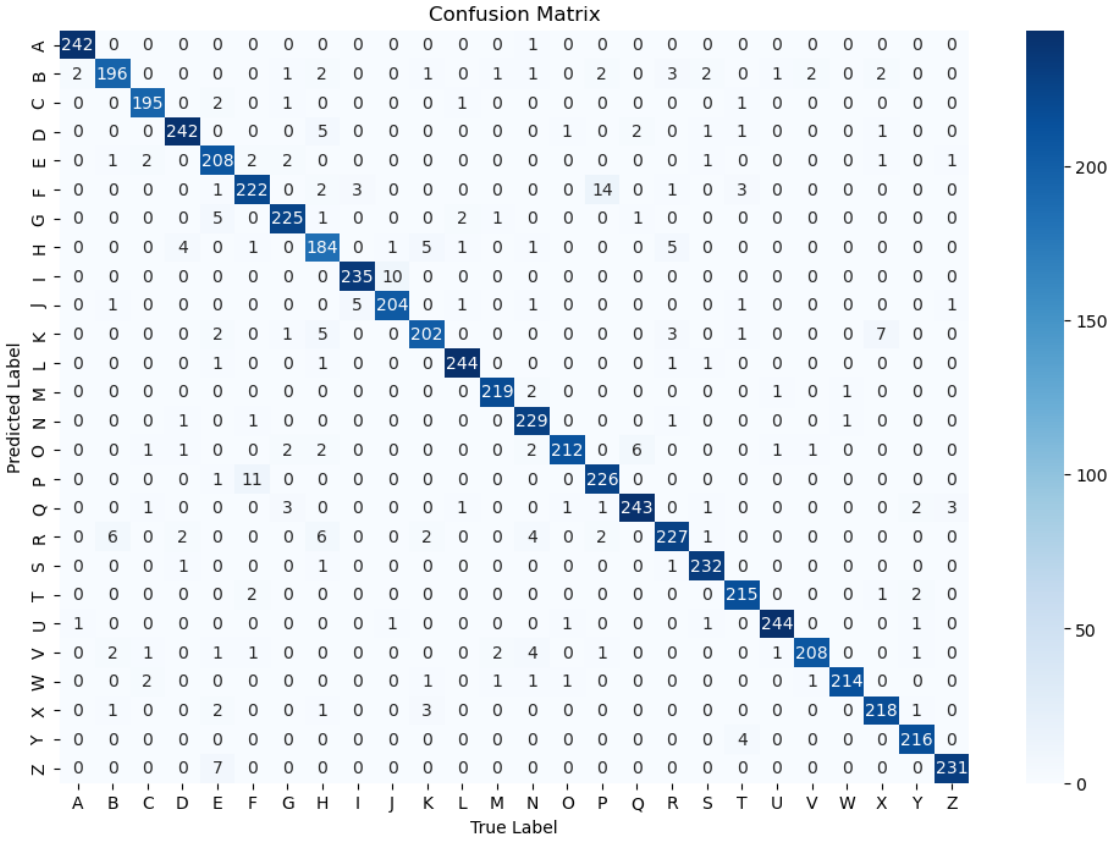
[11]:
```
pr = kNN(xtrain, xtest, ytrain, k=1, method="Euclidean")
```

1.4 Model Testing

Here we evaluate the model from the predicted values and the observed values. In the definition for results, it will make a confusion matrix which will give us insight into how the model performed, show the total correct & incorrect classifications for each instance, and the overall accuracy of the model.

[12]:
```
def Results(yp, yt):

    #join results into dataframe
    df = pd.concat([yp, yt], axis=1)

    #pivot from 'long' to 'wide' format via crosstab.
    confusion_matrix = pd.crosstab(df.iloc[:, 0], df.iloc[:, 1])

    confusion_matrix.index.name=None

    #show confusion matrix
    plt.figure(figsize=(12, 8))
    sns.heatmap(confusion_matrix, annot=True, cmap="Blues", fmt="d")
    plt.xlabel("True Label")
    plt.ylabel("Predicted Label")
    plt.title("Confusion Matrix")
    plt.show()

    totCor = 0
    totInc = 0

    print("Letter, Correct Guesses, Incorrect Guesses:")

    for index,row in confusion_matrix.iterrows():

        print(index, row[index], row.sum()-row[index])

        totCor += row[index]

        totInc += row.sum()-row[index]

    print("Sum " + str(totCor) + " " + str(totInc))

    accuracy = totCor/(len(yt))*100

    print("\nThe accuracy of the model is: " + str(accuracy) + "%")
```

[13]:
```
Results(pr, ytest)
```




```
Letter, Correct Guesses, Incorrect Guesses:
A 242 1
B 196 20
C 195 5
D 242 11
E 208 10
F 222 24
G 225 10
H 184 18
I 235 10
J 204 10
K 202 19
L 244 4
M 219 4
N 229 4
O 212 16
P 226 12
Q 243 13
R 227 23
S 232 3
T 215 5
```

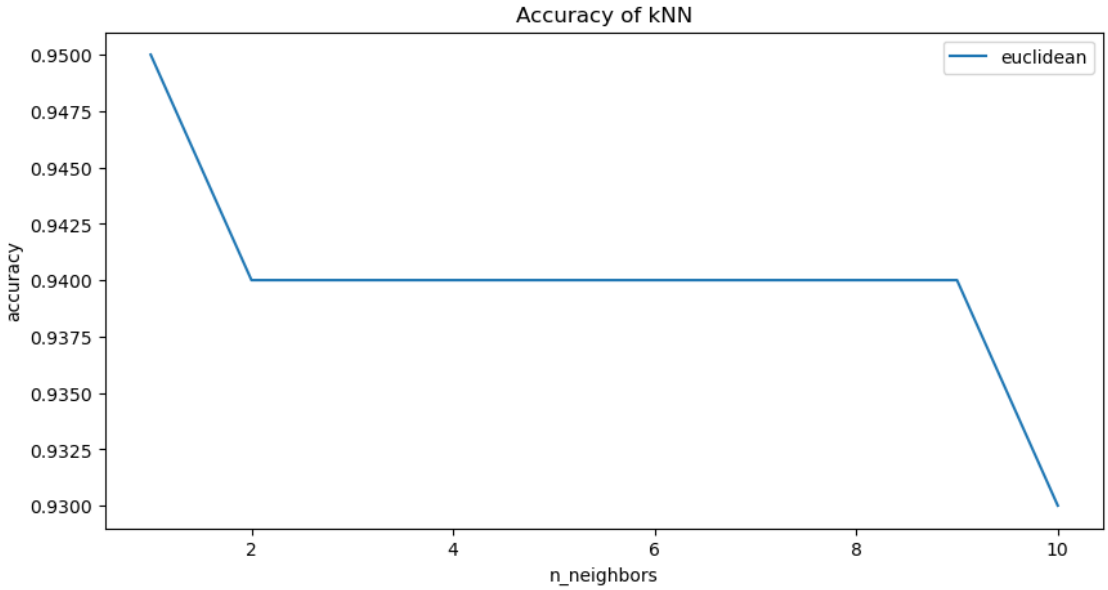
```
U 244 5
V 208 14
W 214 7
X 218 8
Y 216 4
Z 231 7
Sum 5733 267

The accuracy of the model is: 95.55%
```

1.5 Evaluation

Although this model can be applied to almost any dataset, in this case example, we try to predict the value of a letter based on recorded measurements (e.g. heigh, width, thickness, etc). After tweaking K and recording the results, we can produce a graph which will tell us the efficacy of K=1-10. After running the test, the most effective K is k=1 with an accuracy of 95%.

[14]:
```
#After running the test a few times, see the accuracy of kNN (via euclidean␣
 ↪distance) for various K values
#results
results = pd.Series({
    1: 0.95,
    2: 0.94,
    3: 0.94,
    4: 0.94,
    5: 0.94,
    6: 0.94,
    7: 0.94,
    8: 0.94,
    9: 0.94,
    10: 0.93
})
results.plot(figsize=(10,5))
plt.xlabel("n_neighbors")
plt.ylabel("accuracy")
plt.title("Accuracy of kNN")
plt.legend(['euclidean'])
```

[14]:
```
<matplotlib.legend.Legend at 0x21ae3206f10>
```


Accuracy of kNN
euclidean
0.9500
0.9475
0.9450
0.9425
0.9400
0.9375
0.9350
0.9325
0.9300
accuracy
2
4
6
8
10
n_neighbors