HW-2 KNN - Fonts Dataset

Jose Rodriguez, Johnny Nino Ladino, Dayana Sosa; University of Houston

Background

This report will cover the questions presented by Dr. Azencott over the fonts dataset from the University of California Irvine Machine Learning Repository. The fonts dataset is made up of 153 files that are in a csv format. Each file represents a font and within each file there are 412 columns with varying number of rows depending on the file chosen. Each row represents a 'case' and each case describes numerically a digitized image of some specific character (letter or digit) type in a certain font. Each image has a 20 by 20 size and therefore 400 pixels which are represented by 400 columns. The three font files named "Courier", "Calibri" and "Times" will be used for this report. The first 12/412 columns describe the font and the other 400 columns represent the gray levels of the 400 pixels of a specific image. For this dataset, font type will be our target variable and the 400 columns describing the pixel gray levels will be our features. The aim of this report will be to classify the font type using the KNN algorithm. This report will aim to answer all questions presented in Dr. Azencott's homework instructions in a systematic way while analyzing each question and result to have an in depth understanding of this dataset. The underlying code sustaining this report will be created using python and the IDE of Jupyter notebook.

Methodology

The following methodology was used for developing the code and answering the questions in the document

1. Code will be written in python and python libraries such as pandas, numpy, scikit learn etc will be imported
2. Code and report will abide to the question numbers presented in Dr. Azencott's instructions
3. The report/code will use the KNN algorithm to classify the font from the data
4. The report/code will aim to optimize the KNN model to obtain the best model

Preliminary Treatment of Data

Each of the three chosen font files has the same column structure. To begin the analysis, data cleaning and preparation needs to be done on the dataset. Out of the 12 non-related pixel columns, we choose to keep only three of them. The three columns that will be kept are font, strength and italic. Table 1 below explains what each of these columns represent. Any row that contained any missing numerical data was discarded from each of the three files.

FEATURE	DESCRIPTION
FONT	Same as file name. Describes what font the dataset belongs to.
STRENGTH	Column lists values either equal to 0.4 or to 0.7: in each row, strength = 0.4 for normal character; strength = 0.7 for bold character
ITALIC	Column lists values either equal to 0 or to 1; in each row, italic = 0 for normal character; italic = 1 for italic character;

Table 1: 3 Column Descriptions

The three files (referred to as classes) were then filtered down to only include rows containing a strength value of 0.4 and italic value of 0. This filtered the dataset to only images of "normal" characters. The "Calibri", "Courier" and "Times" classes will be referred to as CL1, CL2 and CL3 respectively. Their respective sizes are shown in Table 2 below.

CLASS	ROWS	COLUMNS
CL1 (CALIBRI)	4,768	403
CL2 (COURIER)	4,262	403
CL3 (TIMES)	4,805	403
TOTAL	**13,835**	

Table 2: Class Size

A full data set denoted as DATA was created through the union of the three class CL1, CL2 and CL3 and has a size of 13,835 rows. DATA will contain only the 400 features related to the gray level of the pixels to create a feature matrix. The basic machine learning tool covered in Dr. Azencott's class known as KNN will be used to attempt a rough automatic classification of DATA into the 3 classes CL1, CL2 and CL3.

Part 0

The means and standard deviations of DATA are computed to be used to standardize the dataset DATA. We standardize the dataset so values in certain features are not over weighed or under weighed when we are applying our classifier. We compute the mean and standard deviation of each column leaving us with $\hat{\mu}(mean) = \mu_{X1}, \dots, \mu_{X400}$ and $\hat{\sigma}(standard\ deviation) = \sigma_{X1}, \dots, \sigma_{X400}$. We then perform the following to rescale DATA:

$$\boldsymbol{SDATA(X_i, X_j) = \frac{DATA(X_i, X_j) - \mu_j}{\sigma_j}}$$

Equation 1: Standardized Matrix Equation

We store the standardized dataset DATA into a separate data frame named SDATA. From SDATA, we compute the correlation matrix, $CORR(X_i, X_j)$, and extract the 10 pairs X_i, X_j that have the highest absolute values in $|CORR(X_i, X_j)|$. Here, X_i and X_j, refer to the position of the pixel. Table 3 below highlights the 10 highest correlated pairs by pixel position. From our pixel positions X_i, X_j, we see that the highest correlated pairs tend to share the same row and column position.

$Corr(X_i, X_j)$	X_i	X_j
0.94	r19c18	r19c19
0.93	r19c1	r19c0
0.92	r0c0	r0c1
0.92	r12c1	r13c1
0.91	r12c1	r11c1
0.91	r9c1	r10c1
0.90	r12c2	r13c2
0.90	r11c1	r10c1
0.90	r11c0	r10c0
0.90	r9c0	r8c0

Table 3: Highest Correlated Pairs by Pixel Position

Part 1

1.0

The 80/20 approach of separating data will be taken for the standardized matrix, SDATA. CL1, CL2 and CL3 were split into subsets of 80% and 20% at random to be used in the KNN model. The 80% subset will be for training data and the other 20% subset will be for the test data. Once the 80/20 split for each of the three classes was created, a union of each training and test set was created to have a singular training and test dataset that is equally divided by each class and subset in order to accurately classify in the KNN model. The 80/20 approach helps prevent bias towards one specific class and helps ensure that all classes are considered for the training and test portion of the model evaluation. Table 4 below highlights how this approach splits the data evenly.

DATASET	CL1 ROW COUNT	CL2 ROW COUNT	CL3 ROW COUNT	TOTAL
TRAINSET (80%)	3.814	3,410	3,844	11,068
TESTSET (20%)	954	852	961	2,767

Table 4: Train & Test Sets

1.1

The K nearest neighbor algorithm (commonly referred to as KNN) is the unsupervised machine learning model that will be used to automatically classify arbitrary cases into one of the three classes of CL1, CL2 or CL3. The KNN algorithm will be implemented using a k value of 12 on the training and test datasets created from SDATA. The two percentages of correct classifications for the train and test set were calculated and show in Table 5 below. The percentages shown seem reasonable as the test score accuracy is slightly lower than the training set accuracy. This is expected and the 5% discrepancy in accuracy scores between the two seems to indicate that the model is not overfitting or underfitting the data as there is not a high discrepancy. There could possibly be room for improvement, however, there are no visible red flags with a k value of 12.

Dataset	Accuracy
TRAINSET (80%)	82 %
TESTSET (20%)	77 %

Table 5: K = 12 Accuracy

1.2

We used the same method from 1.1 to perform KNN using k = 5, 10, 15, 20, 30, 40, 50, 100 and computed the respective percentages of correct classification on the test set. We displayed our test accuracy scores in a plot with k values on our x-axis and accuracy percentage on the y-axis. We can see from the Figure 1 that the accuracy scores decrease while k increases. Looking at the plot, we choose the range of values for best k to be [5,10].

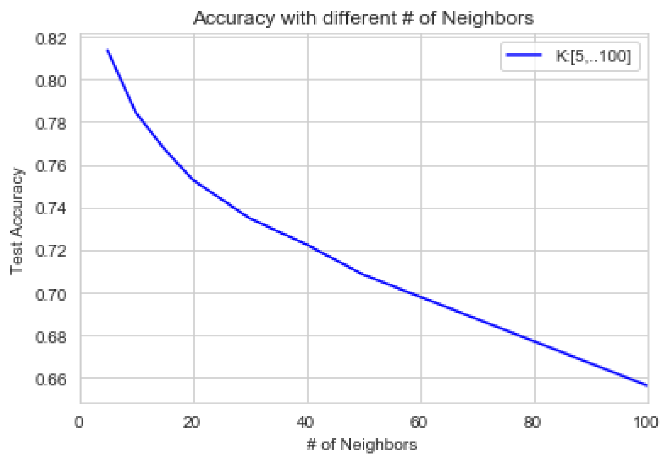

Figure 1: Line Plot of Test Accuracy Scores with Varying K Neighbors

1.3

We shall repeat the KNN algorithm for a few more values of k within the range [5, 10]. In Figure 2, we plotted the accuracy scores for k in this range on the training set and test set. In Table 6, we calculated the difference of test accuracies between the training set and test set for each k. After assessing the plot and calculations, we chose the k with the highest accuracy score and the smallest discrepancy, namely k = 7.

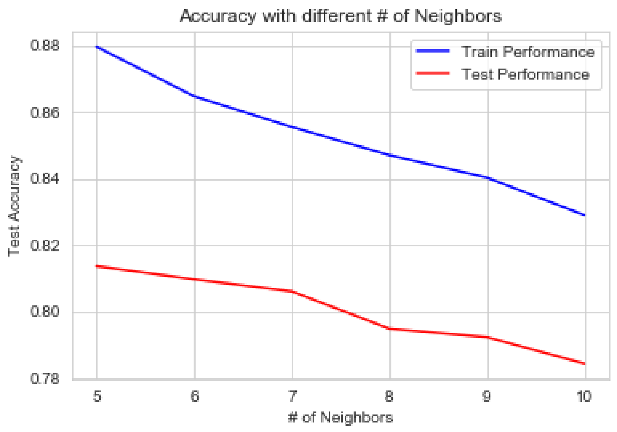

Figure 2: Line Plot of Test Accuracy Scores with Varying K Neighbors

K	TRAINING ACCURACY – TEST ACCURACY
5	0.066
6	0.055
7	0.049
8	0.052
9	0.047
10	0.044

Table 6: Calculations of Discrepancy between Training and Test Accuracy for Varying K Neighbors

1.4

From 1.3, we selected our best k as k = 7 and obtained two confusion matrices after performing KNN, one for the training set and one for the test set. In Figure 3, we see that the training set had higher percentages of correct classifications for each class than the test set, as indicated by the main diagonal. In both the test set and training set, the classes that the KNN model performs best on are Calibri, Times, and Courier, in that order. The largest misclassification that occurred in both the training and test set were when the model should have classified cases as Courier, but instead classified it as Calibri.

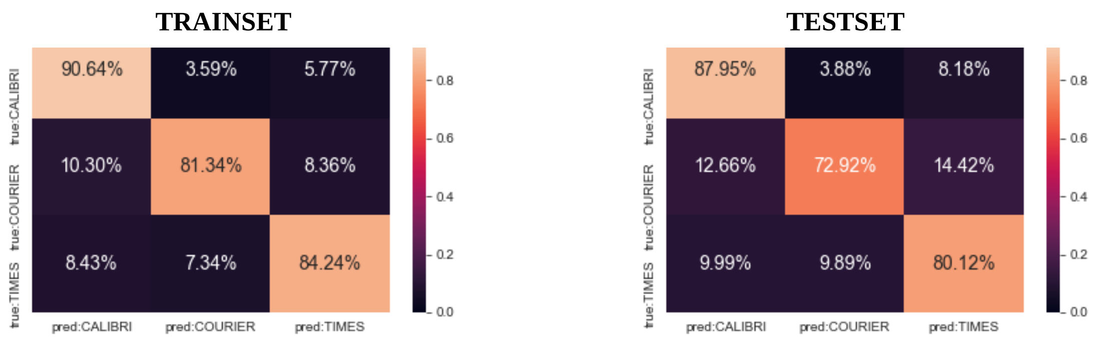

Figure 3: Confusion Matrix on TRAINSET and TESTSET

1.5

From our test confusion matrix, we extract the diagonals to create three confidence intervals for the fonts that were classified correctly. Calculating a 90% interval allows us to get an idea of the range in accuracy scores we might expect to get when running different iterations in our classifier. We can see from the confidence intervals in Table 7 that the range varies between 3% - 5%, thus indicating that the results from our classifier are relatively reliable. From the diagonals and confidence intervals, we see that our classifier does best in classifying the font "CALIBRI" and the range for our 90% confidence interval is smaller compared to the other intervals.

Class	Test Confusion Matrix Diagonals	Confidence Interval for Test Confusion Matrix Diagonals	Train Confusion Matrix Diagonals	Confidence Intervals for Train Confusion Matrix Diagonals
CALIBRI	87.95%	(86.21%, 89.68%)	90.64%	(89.86%, 91.42%)
COURIER	72.92%	(70.42%, 75.42%)	81.34%	(80.24%, 82.44%)
TIMES	80.12%	(78.01%, 82.24%)	84.24%	(83.27%, 85.20%)

Table 7: Confidence Intervals for Diagonals in Test Confusion Matrix and Train Confusion Matrix

1.6 & 1.7

Four subsets of SDATA were created and are referred to as PACK1, PACK2, PACK3 and PACK4. Each of these subsets represent 100 features from SDATA that correspond to the 100-pixel intensities displayed in a specific 10 by 10 windows of the 20 by 20 pixel image. Each "PACK" contain specific 'rLcM' names that correspond to a 10 by 10 window in the image. The 'rLcM' names corresponding to each pack can be found below in Table 8. The corresponding training and test sets of these four "PACKS" were also created.

Pack	L	M
PACK1	0, 1, 2, ... , 9	0, 1, 2, ... , 9
PACK2	0, 1, 2, ... , 9	10, 11, 12, ... , 19
PACK3	10, 11, 12, ... , 19	0, 1, 2, ... , 9
PACK4	10, 11, 12, ... , 19	10, 11, 12, ... , 19

Table 8: rLcM Pack Grouping

The KNN algorithm was applied to each of these packs. The KNN classification using the 'kbest' value found earlier in 1.3 was used for k. The test accuracy scores for each of the packs can be found below in Table 9. The accuracy scores all seem to be close to each other with an average of 75%. However, the 'pack' accuracy scores are lower than the accuracy score found in question 1.4 even though the same value of k (kbest) was used. This is most likely due to the fact that there are less amount of values in the dataset and they are more closely related.

Pack	Test Accuracy
PACK1	77 %
PACK2	72 %
PACK3	75 %
PACK4	76 %

Table 9: Pack KNN Test Accuracy Scores

1.8

The test accuracy scores for each pack shown in Table 9 will be used as weights for each respective feature in each pack. The four weights w1, w2, w3 and w4 each correspond with PACK1, PACK2, PACK3 and PACK4 test accuracy scores respectively. Each respective weight was applied to each feature in each pack via matrix multiplication and each weight was normalized prior to being applied. Each weight was divided by the sum of all the weights so that the sum of the normalized weights was one. After all the weights were applied to the appropriate packs, all the packs were merged to create a dataset containing all 400 features. KNN was applied with k = kbest using all 400 features but with the weighted distance defined by these 400 weights. This makes this KNN algorithm the 'weighted' distance version of KNN and its a variation of the algorithm that may be able to produce better results than the original. The global performance and confusion matrix were computed on the test set in order to compare 'weighted' distance to 'ordinary' distance version of KNN. The test accuracy scores for the two different versions are displayed in Table 10 below. They are very similar, however, it seems like using the weighted distance does negligibly better compared to the ordinary distance version. It is good practice to try different variations of algorithms with different hyper parameters to try and tune the models.

KNN Version	Test Accuracy
Ordinary Distance Version	80.6 %
Weighted Distance Version	80.9 %

Table 10: KNN Version Test Accuracy

Confusion matrices were calculated for both versions to help drill down further into the models and see if there were any changes between the two. Figure 4 below shows the two confusion matrices and helps confirm that using weighted distance version has little to no effect on this model. As shown, the numbers in the matrices are very similar and there are no big changes in any of the values that would indicate that one version is superior to the other in terms of calculating true positive/negatives or false positives/negatives. Depending on the task at hand, confusion matrices can help decide which model to use since the true positives/negatives and false positives/negatives can be important differentiators in which model to go forward with. Since the weighted distance model seems to have little to no effect, the ordinary distance KNN model would be the simpler model to use.

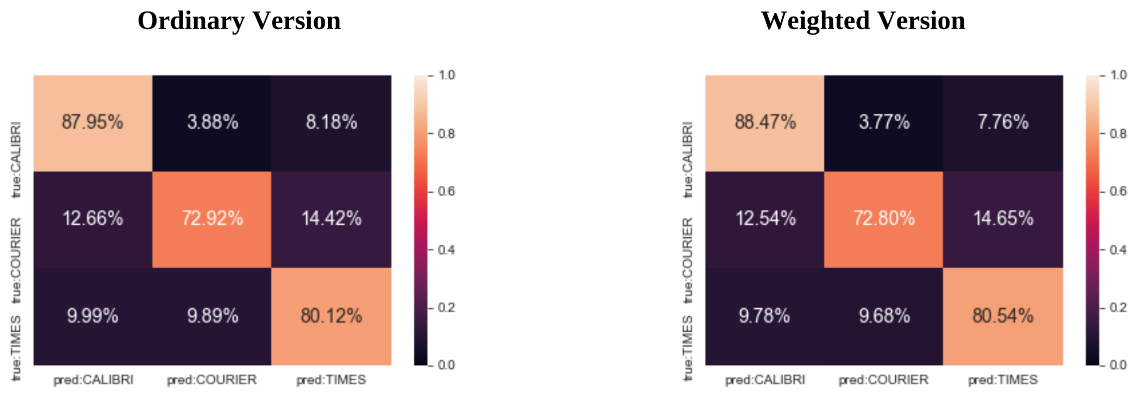

Figure 4: KNN Version Confusion Matrix

Summary

We performed pre-processing and filtering of the data to properly setup the data. Next, we normalized the features to be used in the classification of the data and created a correlation matrix to ensure that the data was correlated. The data was split using the 80/20 approach to ensure a balanced sample was taken from each class. The KNN algorithm was applied in multiple iterations to find the optimal parameter of k. Variations of both the sampling of the data and the version of KNN were ran in order to find the optimal model to be used. The best global accuracy was accomplished using a KNN value of 7 and the “ordinary” distance version of KNN. We successfully applied the KNN algorithm to automatically classify arbitrary cases into one of the three classes chosen from the fonts dataset.

Acknowledgements

The authors gratefully acknowledge the guidance of Professor Robert Azencott.

Fonts Dataset Source

This dataset was taken from the University of California Irvine repository of machine learning datasets. The dataset can be found at following link: https://archive.ics.uci.edu/ml/machine-learning-databases/00417/

References

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013) *An Introduction to Statistical Learning with applications in R*, www.StatLearning.com, Springer-Verlag, New York

Work Distribution

Jose Rodriguez: 33%
Dayana Sosa: 33%
Johnny Nino Ladino: 33%

Code

```
# In[1]:

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from scipy import stats
import math
from numpy import genfromtxt
import png
from numpy import genfromtxt
from PIL import Image

# In[2]:

# Read in csv
calibri_df = pd.read_csv('CALIBRI.csv')
courier_df = pd.read_csv('COURIER.csv')
times_df = pd.read_csv('TIMES.csv')

#Display Dataframe
calibri_df.head()

# # Data Cleaning and Preparation

# In[3]:

dataframes = [calibri_df, courier_df, times_df]

discard = ['fontVariant', 'm_label', 'orientation', 'm_top', 'm_left', 'originalH', 'originalW', 'h', 'w']

calibri_df.drop(columns = discard, inplace = True)
courier_df.drop(columns = discard, inplace = True)
times_df.drop(columns = discard, inplace = True)

# In[4]:

print(calibri_df[calibri_df.isnull().any(axis=1)])
print(courier_df[courier_df.isnull().any(axis=1)])
print(times_df[times_df.isnull().any(axis=1)])

# In[5]:

cl1 = calibri_df[(calibri_df["strength"] == 0.4) & (calibri_df['italic'] == 0)]
cl2 = courier_df[(courier_df['strength'] == 0.4) & (courier_df['italic'] == 0)]
cl3 = times_df[(times_df['strength'] == 0.4) & (times_df['italic'] == 0)]

# In[6]:

print(len(cl1), len(cl2), len(cl3))

# In[7]:
```

```
df = pd.concat([cl1, cl2, cl3], ignore_index=True)

# # Part 0

# In[8]:

scaler = StandardScaler()
scaler.fit(df.iloc[:,3:])
sdf = scaler.transform(df.iloc[:,3:])

# In[9]:

sdf = pd.DataFrame(sdf, columns = df.iloc[:,3:].columns)
corr_sdf = sdf.corr().abs()

# In[10]:

pd.DataFrame(corr_sdf.unstack().sort_values(kind = 'quicksort', ascending=False).drop_duplicates()[0:11])

# In[11]:

sdf = sdf.join(df.iloc[:,0])

# # Part 1

# In[12]:

cl1_train, cl1_test, font1_train, font1_test = train_test_split(sdf[sdf['font'] == 'CALIBRI'].iloc[:,0:400], sdf[sdf['font'] == 'CA
LIBRI'].iloc[:,400], test_size=0.2, random_state=0)
cl2_train, cl2_test, font2_train, font2_test = train_test_split(sdf[sdf['font'] == 'COURIER'].iloc[:,0:400], sdf[sdf['font'] == 'C
OURIER'].iloc[:,400], test_size=0.2, random_state=0)
cl3_train, cl3_test, font3_train, font3_test = train_test_split(sdf[sdf['font'] == 'TIMES'].iloc[:,0:400], sdf[sdf['font'] == 'TIME
S'].iloc[:,400], test_size=0.2, random_state=0)

# In[13]:

X_train = pd.concat([cl1_train, cl2_train, cl3_train])
X_test = pd.concat([cl1_test, cl2_test, cl3_test])
y_train = pd.concat([font1_train, font2_train, font3_train])
y_test = pd.concat([font1_test, font2_test, font3_test])

# In[14]:

k = 12

# In[15]:

classifier = KNeighborsClassifier(n_neighbors=k)

# In[16]:

classifier.fit(X_train, y_train)

# In[17]:

y_pred = classifier.predict(X_train)

# In[18]:
```

```
y_pred_test = classifier.predict(X_test)

# In[19]:

print(accuracy_score(y_true = y_train, y_pred = y_pred))
print(accuracy_score(y_true = y_test, y_pred = y_pred_test))

# In[20]:

k = [5,10,15,20,30,40,50,100]
test_perf_k = []

# In[21]:

for i in k:
    classifier = KNeighborsClassifier(n_neighbors = i)
    classifier.fit(X_train, y_train)
    test_perf_k.append(classifier.predict(X_test))

# In[22]:

test_perf_k2 = []
for i in test_perf_k:
    test_perf_k2.append(accuracy_score(y_true = y_test, y_pred = i))
test_perf_k2

# In[23]:

# Aesthetics
sns.set_style("whitegrid")

# In[24]:

sns.lineplot(x = k, y = test_perf_k2, color = 'blue', label = "K:[5,..100]")
plt.title('Accuracy with different # of Neighbors')
plt.xlabel('# of Neighbors')
plt.ylabel('Test Accuracy')
plt.xlim(0, 100)

# In[81]:

k_N = [5, 6, 7 , 8, 9, 10]
trainperf_N = []
testperf_N = []

for i in k_N:
    # Fitting classifier to the Training set
    classifier = KNeighborsClassifier(n_neighbors = i)
    classifier.fit(X_train, y_train)
    # Predicting the Test set results
    y_pred_test = classifier.predict(X_test)
    # computing the % of correct classification on test set to a list & printing results
    testperf_N.append(accuracy_score(y_test, y_pred_test))

    y_pred_train = classifier.predict(X_train)
    trainperf_N.append(accuracy_score(y_train, y_pred_train))

# In[26]:

testperf_N
```

```
# In[82]:

testperf_N

# In[88]:

for i in range(0, len(testperf_N)):
    print(k_N[i], trainperf_N[i] - testperf_N[i])

# In[89]:

f, ax = plt.subplots(1, 1)

sns.lineplot(x = k_N, y = trainperf_N, color="blue", label="Train Performance")
sns.lineplot(x = k_N, y = testperf_N, color="red", label="Test Performance")

plt.title('Accuracy with different # of Neighbors')
plt.xlabel('# of Neighbors')
plt.ylabel('Test Accuracy')
ax.legend()
plt.savefig('Accuracy_with_K_ab.png')

plt.show()

# In[27]:

f, ax = plt.subplots(1, 1)

sns.lineplot(x = k, y = test_perf_k2, color="blue", label="K:[5,..100]")
sns.lineplot(x = k_N, y = testperf_N, color="red", label="K:[5,..,10]")

plt.title('Accuracy with different # of Neighbors')
plt.xlabel('# of Neighbors')
plt.ylabel('Test Accuracy')
plt.xlim(0, 100)
ax.legend()
plt.savefig('Accuracy_with_K_ab.png')

plt.show()

# In[28]:

testperf_df_N = pd.DataFrame({'k': k_N,
                              'test_perf': testperf_N})
testperf_df_N = testperf_df_N.sort_values(by = ['test_perf'], ascending = False)

# In[91]:

kbest = 7

# In[92]:

classifier = KNeighborsClassifier(n_neighbors = kbest)
classifier.fit(X_train, y_train)
y_pred_train = classifier.predict(X_train)
y_pred_test = classifier.predict(X_test)

# In[93]:
```

```
cmtx_a_train = pd.DataFrame(
    confusion_matrix(y_true=y_train,y_pred = y_pred_train, labels = ['CALIBRI', 'COURIER', 'TIMES'], normalize = 'true'),
    index=['true:CALIBRI', 'true:COURIER', 'true:TIMES'],
    columns=['pred:CALIBRI', 'pred:COURIER', 'pred:TIMES'])

# In[94]:

cmtx_a_train

# In[95]:

sns.heatmap(cmtx_a_train, annot=True, fmt = '.02%', annot_kws={"size": 14}, vmin = 0, vmax = 1)
plt.savefig('cmtx_train.png')

# In[96]:

N = len(y_train[y_train == 'CALIBRI'])
p_N = np.diagonal(confusion_matrix(y_true = y_train, y_pred = y_pred_train))[0] / N
sigma = math.sqrt(((p_N * (1 - p_N)) / N))
stats.norm.interval(alpha = 0.90, loc=p_N, scale=sigma)

# In[97]:

N = len(y_train[y_train == 'COURIER'])
p_N = np.diagonal(confusion_matrix(y_true = y_train, y_pred = y_pred_train))[1] / N
sigma = math.sqrt(((p_N * (1 - p_N)) / N))
stats.norm.interval(alpha = 0.90, loc=p_N, scale=sigma)

# In[98]:

N = len(y_train[y_train == 'TIMES'])
p_N = np.diagonal(confusion_matrix(y_true = y_train, y_pred = y_pred_train))[2] / N
sigma = math.sqrt(((p_N * (1 - p_N)) / N))
stats.norm.interval(alpha = 0.90, loc=p_N, scale=sigma)

# In[99]:

confusion_matrix(y_true = y_test, y_pred = y_pred_test)

# In[100]:

cmtx_a_test = pd.DataFrame(
    confusion_matrix(y_true=y_test,y_pred = y_pred_test, labels = ['CALIBRI', 'COURIER', 'TIMES'], normalize = 'true'),
    index=['true:CALIBRI', 'true:COURIER', 'true:TIMES'],
    columns=['pred:CALIBRI', 'pred:COURIER', 'pred:TIMES'])

# In[101]:

cmtx_a_test

# In[102]:

sns.heatmap(cmtx_a_test, annot=True, fmt = '.02%', annot_kws={"size": 14}, vmin = 0, vmax = 1)
plt.savefig('cmtx_test.png')

# In[103]:

p_N = sum(np.diagonal(confusion_matrix(y_true = y_test, y_pred = y_pred_test))) / len(y_test)
N = len(y_test)
sigma = math.sqrt(((p_N * (1 - p_N)) / N))
```

```
stats.norm.interval(alpha = 0.95, loc=p_N, scale=sigma)

# In[104]:

p_N = sum(np.diagonal(confusion_matrix(y_true = y_test, y_pred = y_pred_test))) / len(y_test)
N = len(y_test)
sigma = math.sqrt(((p_N * (1 - p_N)) / N))
stats.norm.interval(alpha = 0.90, loc=p_N, scale=sigma)

# In[105]:

N = len(y_test[y_test == 'CALIBRI'])
p_N = np.diagonal(confusion_matrix(y_true = y_test, y_pred = y_pred_test))[0] / N
sigma = math.sqrt(((p_N * (1 - p_N)) / N))
stats.norm.interval(alpha = 0.90, loc=p_N, scale=sigma)

# In[106]:

N = len(y_test[y_test == 'COURIER'])
p_N = np.diagonal(confusion_matrix(y_true = y_test, y_pred = y_pred_test))[1] / N
sigma = math.sqrt(((p_N * (1 - p_N)) / N))
stats.norm.interval(alpha = 0.9, loc=p_N, scale=sigma)

# In[107]:

N = len(y_test[y_test == 'TIMES'])
p_N = np.diagonal(confusion_matrix(y_true = y_test, y_pred = y_pred_test))[2] / N
sigma = math.sqrt(((p_N * (1 - p_N)) / N))
stats.norm.interval(alpha = 0.9, loc=p_N, scale=sigma)

# In[108]:

cols = list(X_train.iloc[:,:].columns)
cols_split = np.array_split(cols, 40)
cols_split_part1 = cols_split[0:20]
cols_split_part2 = cols_split[20:40]

# In[109]:

PACK1 = np.concatenate(cols_split_part1[0::2]).tolist()
PACK2 = np.concatenate(cols_split_part1[1::2]).tolist()
PACK3 = np.concatenate(cols_split_part2[0::2]).tolist()
PACK4 = np.concatenate(cols_split_part2[1::2]).tolist()

# In[110]:

Packs = [PACK1, PACK2, PACK3, PACK4]
W = []

# In[111]:

for i in range(len(Packs)):
    # Fitting classifier to the Training set
    classifier = KNeighborsClassifier(n_neighbors = kbest)
    classifier.fit(X_train[Packs[i]], y_train)
    # Predicting the Test set results
    Y_Pred = classifier.predict(X_test[Packs[i]])
    # computing the % of correct classification on test set to a list
    W.append(accuracy_score(y_test, Y_Pred)) #performance
```

```
# In[112]:

Pack_Names = ['PACK1', 'PACK2', 'PACK3', 'PACK4']
W_dict = {'Pack': Pack_Names, 'W': W}
W_df = pd.DataFrame(W_dict)
W_df

# In[113]:

Weight_Sum_Check = (W[0]/sum(W)) + (W[1]/sum(W)) + (W[2]/sum(W)) + (W[3]/sum(W))

PACK1_Train = X_train[Packs[0]] * (W[0]/sum(W))
PACK2_Train = X_train[Packs[1]] * (W[1]/sum(W))
PACK3_Train = X_train[Packs[2]] * (W[2]/sum(W))
PACK4_Train = X_train[Packs[3]] * (W[3]/sum(W))

PACK1_Test = X_test[Packs[0]] * (W[0]/sum(W))
PACK2_Test = X_test[Packs[1]] * (W[1]/sum(W))
PACK3_Test = X_test[Packs[2]] * (W[2]/sum(W))
PACK4_Test = X_test[Packs[3]] * (W[3]/sum(W))

X_PACK_Train = pd.concat([PACK1_Train, PACK2_Train, PACK3_Train, PACK4_Train], axis=1)
X_PACK_Test = pd.concat([PACK1_Test, PACK2_Test, PACK3_Test, PACK4_Test], axis=1)

# In[114]:

classifier = KNeighborsClassifier(n_neighbors = kbest)
classifier.fit(X_PACK_Train, y_train)
# Predicting the Test set results
Y_Pred = classifier.predict(X_PACK_Test)
# Making the Confusion Matrix for Test data
Weighted_Distance_testconf = confusion_matrix(y_test, Y_Pred)
Weighted_Distance_perf = accuracy_score(y_test, Y_Pred)

# In[115]:

Weighted_Distance_perf

# In[116]:

Weighted_Distance_testconf = confusion_matrix(y_test, Y_Pred, normalize = 'true')

# In[117]:

Weighted_Distance_testconf = pd.DataFrame(
    confusion_matrix(y_true=y_test,y_pred = Y_Pred, labels = ['CALIBRI', 'COURIER', 'TIMES'], normalize = 'true'),
    index=['true:CALIBRI', 'true:COURIER', 'true:TIMES'],
    columns=['pred:CALIBRI', 'pred:COURIER', 'pred:TIMES'])

# In[118]:

sns.heatmap(Weighted_Distance_testconf, annot=True, fmt = '.02%', annot_kws={"size": 14}, vmin = 0, vmax = 1)
plt.savefig('Weighted_Distance_testconf.png')
plt.title('Weighted Version')

# In[119]:

my_data = genfromtxt('calibri2.csv', delimiter=',')

# In[121]:
```

```
image = Image.fromarray(my_data.transpose())

# In[122]:

i = 3
j = i + 20

images = []
for z in range(0, 100):
    image_z = []
    for k in range(0,20):
        image_z.append(calibri_df.iloc[z,i:j])
        i += 20
        j += 20
    images.append(np.array(image_z))
    i = 3
    j = 23

# In[123]:

image = Image.fromarray((images[11]).astype(np.uint8))

image.convert('RGB')

# In[124]:

i = 3
j = i + 20

images = []
for z in range(0, 100):
    image_z = []
    for k in range(0,20):
        image_z.append(courier_df.iloc[z,i:j])
        i += 20
        j += 20
    images.append(np.array(image_z))
    i = 3
    j = 23

# In[125]:

image = Image.fromarray((images[11]).astype(np.uint8))

image.convert('RGB')

# In[126]:

i = 3
j = i + 20

images = []
for z in range(0, 100):
    image_z = []
    for k in range(0,20):
        image_z.append(times_df.iloc[z,i:j])
        i += 20
        j += 20
    images.append(np.array(image_z))
    i = 3
    j = 23


image = Image.fromarray((images[11]).astype(np.uint8))

image.convert('RGB')
```