# Comp5318_Assignment_2_Group43

November 13, 2020

# 1 COMP5318 - Machine Learning and Data Mining: Assignment 2

## 1.1 Group - 43

Name: Alejandro Diaz SID: 500229318 Nicholas Stollmann SID: 312062796 Name: Shashwati Dutta SID: 500693287

## 1.2 Introduction

The aim of this assignment is to find the best method to undertake character recognition. The dataset we are analysing is made up of 62992 synthesised characters from computer fonts. The data is split into 62 classes of handwritten images made up of all letters in lower and upper case as well as the ten digits.We propose three classification methods, K- Nearest Neighbor, Random Forest and Convolutional Neural Network and based on series of experiments and evaluation metric we aim to identify the best suited model for the data.

## 1.3 Libraries

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter, OrderedDict
import os
import gc
import string
from random import sample
from PIL import Image, ImageOps
import seaborn as sns
import pandas as pd

import h5py
import plotly.graph_objects as go

from sklearn.preprocessing import scale, LabelEncoder
from sklearn.decomposition import PCA
from sklearn.ensemble import AdaBoostClassifier,RandomForestClassifier
from sklearn.model_selection import␣
↪train_test_split,cross_val_score,GridSearchCV
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix,␣
↪plot_confusion_matrix,confusion_matrix,classification_report,accuracy_score
from sklearn.tree import plot_tree

from keras.regularizers import l2
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Dropout, Flatten
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.layers.advanced_activations import LeakyReLU, PReLU
from keras.utils import np_utils, generic_utils
from keras.optimizers import SGD
from keras.wrappers.scikit_learn import KerasClassifier
from keras.utils.vis_utils import plot_model

from xgboost import XGBClassifier
```

### 1.4 Data Loading

**Below code was run initially to load the data from the downloaded folders. It was then saved in .h5 file and used, due to high data loading time.** The dataset used in our experiments is the The Char47 dataset (English language) which consists of characters in natural images. In this case, we selected the synthesized set of characters. This dataset contains 62992 images of characters from computer fonts. http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/

[11]:
```
######### Raw Data loading #######

n_dim = 28

DATAPATH = "./English/Fnt"
CLASSES = [[str(el) for el in np.arange(0, 10)], list(string.ascii_uppercase),␣
↪list(string.ascii_lowercase)]
CLASSES = [item for sublist in CLASSES for item in sublist]

list_imgs = []
labels = []
i = 0
 # Get all images
for folder in sorted(os.listdir(DATAPATH)):
    folder_content = os.path.join(DATAPATH,folder)
    for img in sorted(os.listdir(folder_content)):
        list_imgs.append(os.path.join(folder_content,img))
        labels.append(CLASSES[i])
    i += 1

print(f"Number of images: {len(list_imgs)}")
```

```python
# # Convert labels list to array
labels = np.array(labels)

# Iterate over each img
for i, im in enumerate(list_imgs):

     # Open image
    image = Image.open(im)
     # Resize it
    image = image.resize((n_dim, n_dim))
     # If first one - create array
    if i == 0:
        images = np.expand_dims(
             (np.array(image, dtype= float)/255).reshape(-1), axis= 0)
     # Else append images matrix
    else:
        image = np.expand_dims(
             (np.array(image, dtype= float)/255).reshape(-1), axis= 0)
        images = np.append(images, image, axis= 0)
    if i % 1000 == 0:
        print(f"Number of loaded images: {i}")

# ######### Converting to .h5 file #######
labels2 = labels.astype(h5py.special_dtype(vlen=str))

with h5py.File('./data/images.h5','w') as H:
     H.create_dataset('Images', data=images)


with h5py.File('./data/labels.h5','w') as H:
     H.create_dataset('labels', data=labels2)


###### Loading data from .h5 file ######
#with h5py.File('./data/images.h5','r') as H:
#    images = np.copy(H['images'])
#with h5py.File('./data/labels.h5','r') as H:
#     labels = np.copy(H['labels'])

print(f"Data shape: {images.shape}")
print(f"Labels shape: {labels.shape}")
```

```
Number of images: 62992
Number of loaded images: 0
Number of loaded images: 1000
Number of loaded images: 2000
```

```
Number of loaded images: 3000
Number of loaded images: 4000
Number of loaded images: 5000
Number of loaded images: 6000
Number of loaded images: 7000
Number of loaded images: 8000
Number of loaded images: 9000
Number of loaded images: 10000
Number of loaded images: 11000
Number of loaded images: 12000
Number of loaded images: 13000
Number of loaded images: 14000
Number of loaded images: 15000
Number of loaded images: 16000
Number of loaded images: 17000
Number of loaded images: 18000
Number of loaded images: 19000
Number of loaded images: 20000
Number of loaded images: 21000
Number of loaded images: 22000
Number of loaded images: 23000
Number of loaded images: 24000
Number of loaded images: 25000
Number of loaded images: 26000
Number of loaded images: 27000
Number of loaded images: 28000
Number of loaded images: 29000
Number of loaded images: 30000
Number of loaded images: 31000
Number of loaded images: 32000
Number of loaded images: 33000
Number of loaded images: 34000
Number of loaded images: 35000
Number of loaded images: 36000
Number of loaded images: 37000
Number of loaded images: 38000
Number of loaded images: 39000
Number of loaded images: 40000
Number of loaded images: 41000
Number of loaded images: 42000
Number of loaded images: 43000
Number of loaded images: 44000
Number of loaded images: 45000
Number of loaded images: 46000
Number of loaded images: 47000
Number of loaded images: 48000
Number of loaded images: 49000
Number of loaded images: 50000
```

```
Number of loaded images: 51000
Number of loaded images: 52000
Number of loaded images: 53000
Number of loaded images: 54000
Number of loaded images: 55000
Number of loaded images: 56000
Number of loaded images: 57000
Number of loaded images: 58000
Number of loaded images: 59000
Number of loaded images: 60000
Number of loaded images: 61000
Number of loaded images: 62000
Data shape: (62992, 784)
Labels shape: (62992,)
```

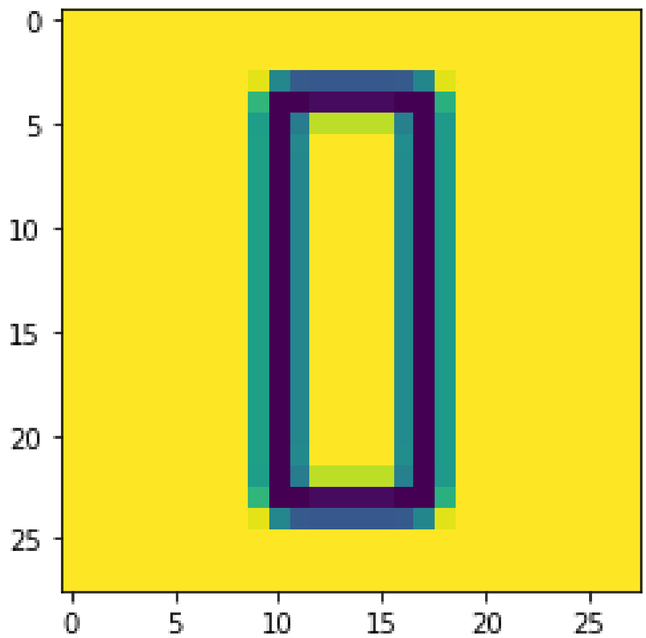
**The next function array2img allows us to convert an 1D array into a proper image using numpy().**

[12]:
```
def array2img(array, w=28, h=28, c=1):
    """
    Function to convert an 1D array image into a 2D matrix

    :param array: 1D array which we want to convert into a 2D matrix
    :param w: Width image
    :param h: Height image
    :param c: Color channel
    """
    if c==1:
      return np.asarray(array).reshape(w, h)
    else:
      return np.asarray(array).reshape(w, h, c)
```

**Showing sample image**

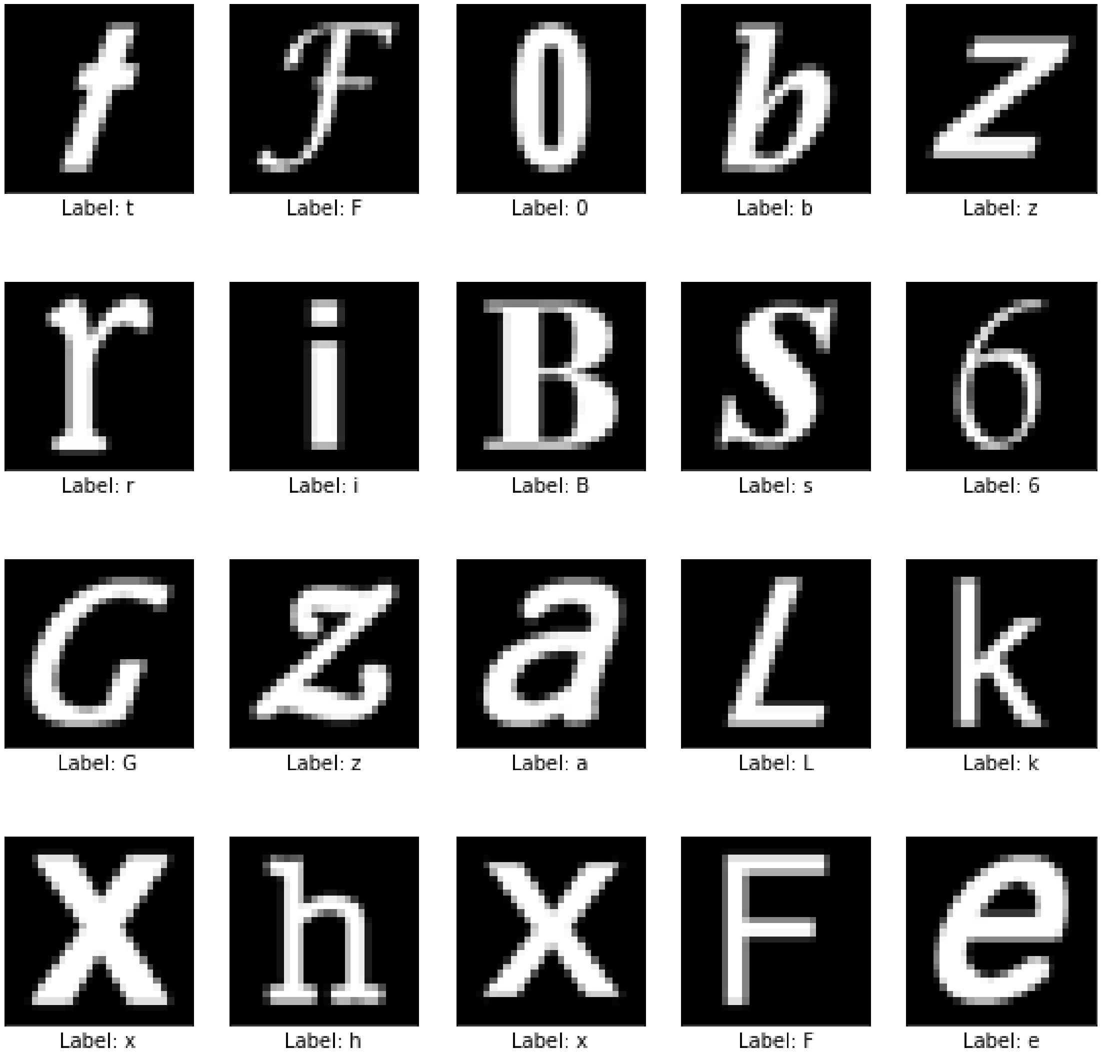
[13]:
```
plt.imshow(array2img(images[0]))
```

[13]:
```
<matplotlib.image.AxesImage at 0x1a424495ac0>
```




**We can randomly plot some images.**

[6]:
```
# Plot n images
n = 20

# Generate random index to select random images
np.random.seed(42)
img_idx = np.random.randint(1, len(images), n)
cls = [labels[i] for i in img_idx]

plt.figure(figsize=(10,10))
for i, idx in enumerate(img_idx):
    plt.subplot(4, 5, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)

    # Get a random image
    img2d = array2img(images[idx])

    # Get label for this image
    plt.imshow(img2d, cmap=plt.cm.binary)
    plt.xlabel(f"Label: {cls[i]}")
plt.show()
```




### 1.5 Preprocessing

In this section, we will implement different preprocess techniques to prepare and improve the quality of our data and thus, improve the performance of our model.

#### 1.5.1 Balanced or Imbalanced

An important part to design a proper classifier is to analyze if our data contain the same number of samples for each label.

[7]:
```
# Unique labels in our data
unique_labels = list(set(labels))

# Count elements per label
count2idx = dict(zip(Counter(labels).keys(), Counter(labels).values()))
```

```python
# Order dictionary by value
sorted_count2idx = OrderedDict(sorted(count2idx.items(), key=lambda t: t[1]))

counter = {}
for idx, count in sorted_count2idx.items():
    counter[idx] = count
    print(f"There are {count} images for label {idx}")
```

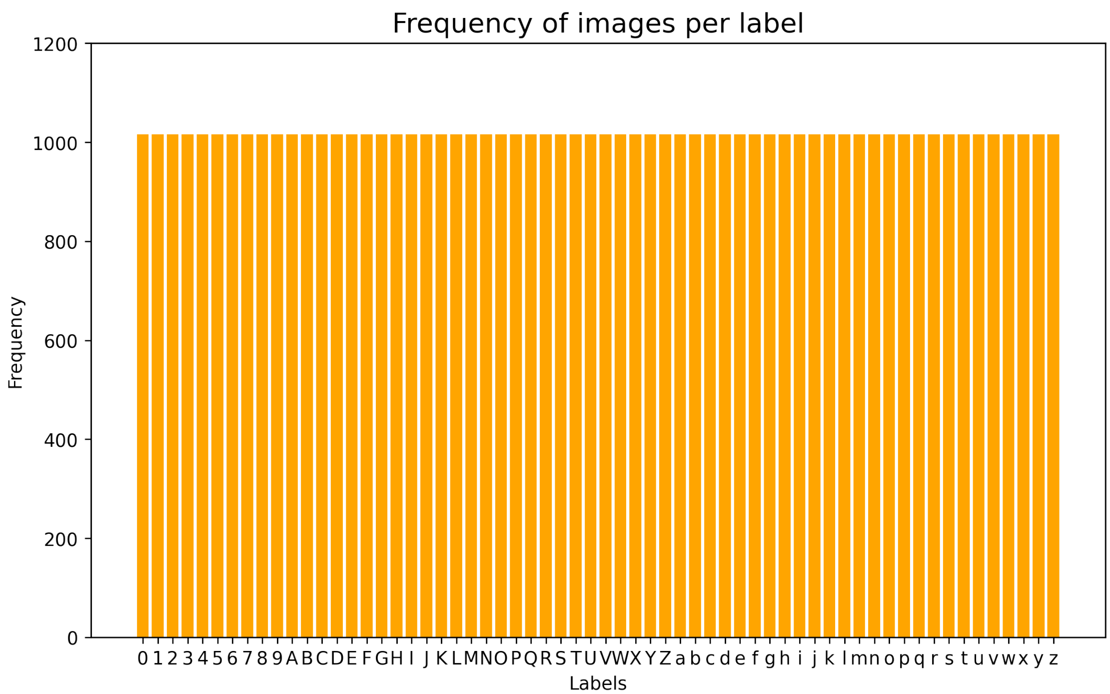
```
There are 1016 images for label 0
There are 1016 images for label 1
There are 1016 images for label 2
There are 1016 images for label 3
There are 1016 images for label 4
There are 1016 images for label 5
There are 1016 images for label 6
There are 1016 images for label 7
There are 1016 images for label 8
There are 1016 images for label 9
There are 1016 images for label A
There are 1016 images for label B
There are 1016 images for label C
There are 1016 images for label D
There are 1016 images for label E
There are 1016 images for label F
There are 1016 images for label G
There are 1016 images for label H
There are 1016 images for label I
There are 1016 images for label J
There are 1016 images for label K
There are 1016 images for label L
There are 1016 images for label M
There are 1016 images for label N
There are 1016 images for label O
There are 1016 images for label P
There are 1016 images for label Q
There are 1016 images for label R
There are 1016 images for label S
There are 1016 images for label T
There are 1016 images for label U
There are 1016 images for label V
There are 1016 images for label W
There are 1016 images for label X
There are 1016 images for label Y
There are 1016 images for label Z
There are 1016 images for label a
There are 1016 images for label b
```

```
There are 1016 images for label c
There are 1016 images for label d
There are 1016 images for label e
There are 1016 images for label f
There are 1016 images for label g
There are 1016 images for label h
There are 1016 images for label i
There are 1016 images for label j
There are 1016 images for label k
There are 1016 images for label l
There are 1016 images for label m
There are 1016 images for label n
There are 1016 images for label o
There are 1016 images for label p
There are 1016 images for label q
There are 1016 images for label r
There are 1016 images for label s
There are 1016 images for label t
There are 1016 images for label u
There are 1016 images for label v
There are 1016 images for label w
There are 1016 images for label x
There are 1016 images for label y
There are 1016 images for label z
```

**We can graph the frequency of images per label.**

[8]:
```python
# Set size of figure
fig = plt.figure(figsize=(10,6), dpi= 250)

tick_labels, values = zip(*counter.items())
plt.bar(tick_labels, values, color="orange")
plt.title('Frequency of images per label', fontsize=15)
plt.xlabel('Labels')
plt.ylabel('Frequency')
plt.ylim([0, 1200])
```

[8]:
```
(0.0, 1200.0)
```




**We can observe how our classes contain the same number of images. Hence, in this case the dataset is perfectly balanced.**

### 1.5.2 Train, Validation and Test Split

**To measure the performance of our algorithms and apply fine-tuning, we will split our data in a training 60%, validation 20% and test set 20%.**

[3]:
```
X1, X_test, y1, y_test = train_test_split(images, labels, test_size = 0.20,␣
 ↪random_state = 1) #Test set 20%

#X1, y1 are placeholders for the next split
X_train, X_val, y_train, y_val = train_test_split(X1, y1, test_size = 0.26,␣
 ↪random_state = 1) #Validation set 20% of TOTAL

print(f"X_train shape: {X_train.shape}"),
print(f"X_val shape: {X_val.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_val shape: {y_val.shape}")
print(f"y_test shape: {y_test.shape}")
```

```
X_train shape: (37290, 784)
X_val shape: (13103, 784)
X_test shape: (12599, 784)
```

```
y_train shape: (37290,)
y_val shape: (13103,)
y_test shape: (12599,)
```

### 1.5.3 Standardize Data

Additionally, an important step in order to be able to apply Principal Component Analysis (PCA) is to standardize our images so that they have mean 0 and variance =1. We can perform this step using the next formula:

$$\frac{x - \mu}{\sigma}$$

However to correctly standardize our data, we should use the training set to calculate the mean and variance, normalize the training set and then normalize the validation and test set using the same mean and variance from training set.

In a nutshell:

**scaled_train = (train - train_mean) / train_std_deviation**

**scaled_validation = (validation - train_mean) / train_std_deviation**

**scaled_test = (test - train_mean) / train_std_deviation**

[4]:
```
train_mean = np.mean(X_train)
train_std_deviation = np.std(X_train)
```

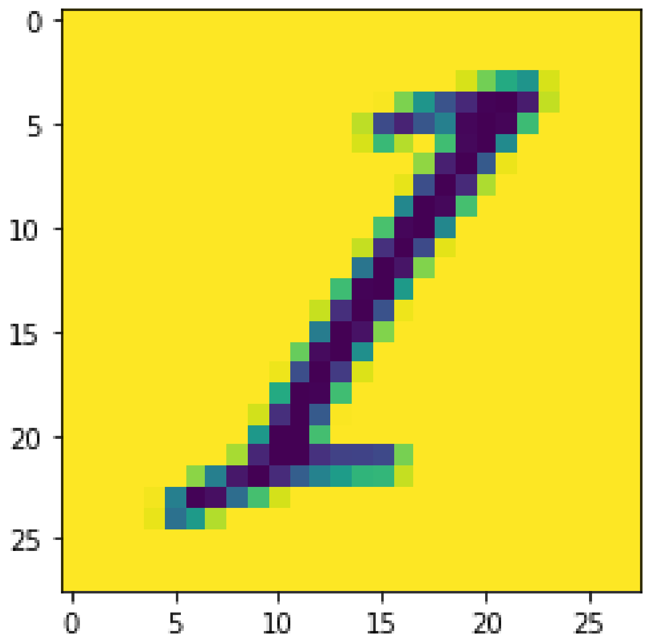
**Scaling the training, validation and test sets**

[5]:
```
scaled_X_train = (X_train - train_mean) / train_std_deviation
scaled_X_val = (X_val - train_mean) / train_std_deviation
scaled_X_test = (X_test - train_mean) / train_std_deviation
```

**Display example**

[12]:
```
plt.imshow(array2img(scaled_X_train[0]))
```

[12]:
```
<matplotlib.image.AxesImage at 0x7fb80c3d9b70>
```

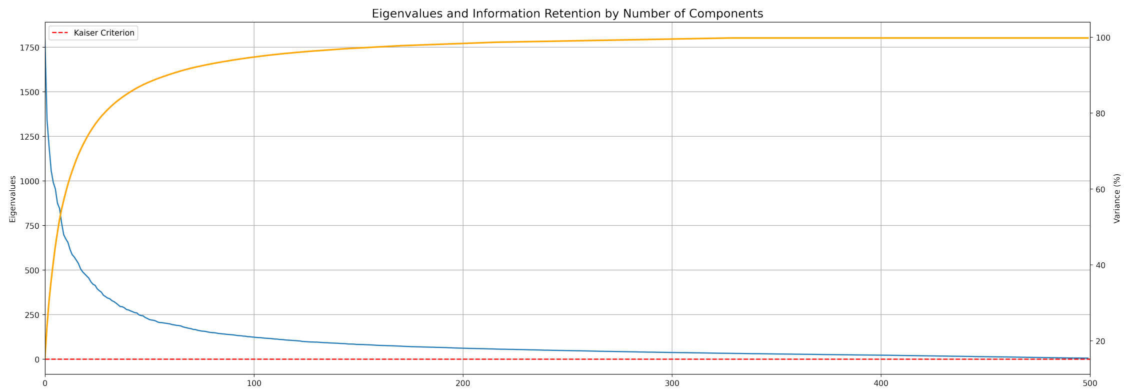


### 1.6 Dimensionality Reduction - PCA

Principal components analysis (PCA) is a popular approach for deriving principal components analysis a low-dimensional set of features from a large set of variables. We will use the PCA function from the sklearn package to perform the dimensionality reduction on the scaled dataset.

[13]:
```
n_comp = 500

pca = PCA(n_components = n_comp)

# Fit with our data
projected_data = pca.fit(scaled_X_train)

fig, ax1 = plt.subplots(figsize=(20, 7), dpi=250)

plt.title("Eigenvalues and Information Retention by Number of Components",␣
↪fontsize=15)

# Eigenvalues
ax1.set_ylabel('Eigenvalues', fontsize=10)
ax1.plot(range(0, pca.singular_values_.shape[0]), pca.singular_values_)
ax1.axhline(y=1, color='r', linestyle='--', label="Kaiser Criterion")
ax1.tick_params(axis='y')
ax1.set_xlim([0, n_comp])
ax1.legend(loc="upper left")
ax2 = ax1.twinx()
```

```
# Information Retention
ax2.set_ylabel('Variance (%)', fontsize=10)
ax2.plot(np.arange(0, 500),
         np.cumsum(np.round(pca.explained_variance_ratio_, decimals=4)*100),
         color="orange", linewidth=2)
ax2.tick_params(axis='y')
ax1.grid()

fig.tight_layout()
plt.show()
```




**How we select the number of principal components?** Researchers trying to avoid selecting the number of principal components (PCs) using subjective criteria. Many methods have been developed to solve this problem and choose the number of PCs using an objective method. To make a comparison between some of these methods are out of the scope of this assignment but 2 techniques are mentioned:

- We can use a visual criterion to select the number of main components to be taken. We must take the value where the curve starts to flatten.
- Another technique is to use the Kaiser criterion, where we discard all components that have a eigenvalue of less than 1

In this case and for simplicity, we analyzed the percentage of information retention to select the number of principal components we should take. The above figure shows that the % of information retention depends on the number of principal components we select. In this case, most of the information is retained by the first components. Furthermore, it shows that by keeping about 256 features, we can retain about 98% representation of the image data. For this reason, **we will reduce our data until 256 features.**

Now we selected the number of components to use, we can project our data in the new components.

[6]:
```
n_comp = 256
pca = PCA(n_components = n_comp)
```

```
# Fit with our data
X_train = pca.fit_transform(scaled_X_train)
y_train = y_train
```

**The shape of our projected data is:**

[15]:
```
print(f"Shape of projected data - X_train: {X_train.shape}")
print(f"Shape of y_train: {y_train.shape}")
```

```
Shape of projected data - X_train: (37290, 256)
Shape of y_train: (37290,)
```

**Finally, we need to project our validation and test set into these new components.**

[7]:
```
X_val = pca.transform(scaled_X_val)
y_val = y_val

X_test = pca.transform(scaled_X_test)
y_test = y_test

print(f"Shape of projected data - X_validation: {X_val.shape}")
print(f"Shape of y_train: {y_val.shape}")
print(f"Shape of projected data - X_test: {X_test.shape}")
print(f"Shape of y_train: {y_test.shape}")
```

```
Shape of projected data - X_validation: (13103, 256)
Shape of y_train: (13103,)
Shape of projected data - X_test: (12599, 256)
Shape of y_train: (12599,)
```

**Additionally, we can generate a graph to visualize our projected data. As we can not generate a graph to visualize the projection of 256 features, we we will project our data in 3 principal components to plot a 3D graph.**

[8]:
```
n_comp = 3

# Initialize sklearn pca with 3 components
pca_3comp = PCA(n_components = 3)

indices = sample(list(range(X_train.shape[0])), 5000)
images_3d  = pca_3comp.fit_transform(X_train[indices])

print(f"Projection shape= {images_3d.shape}")

# Get label to set color in 3D graph
y_label = []
y_label_color = []
for el in indices:
```

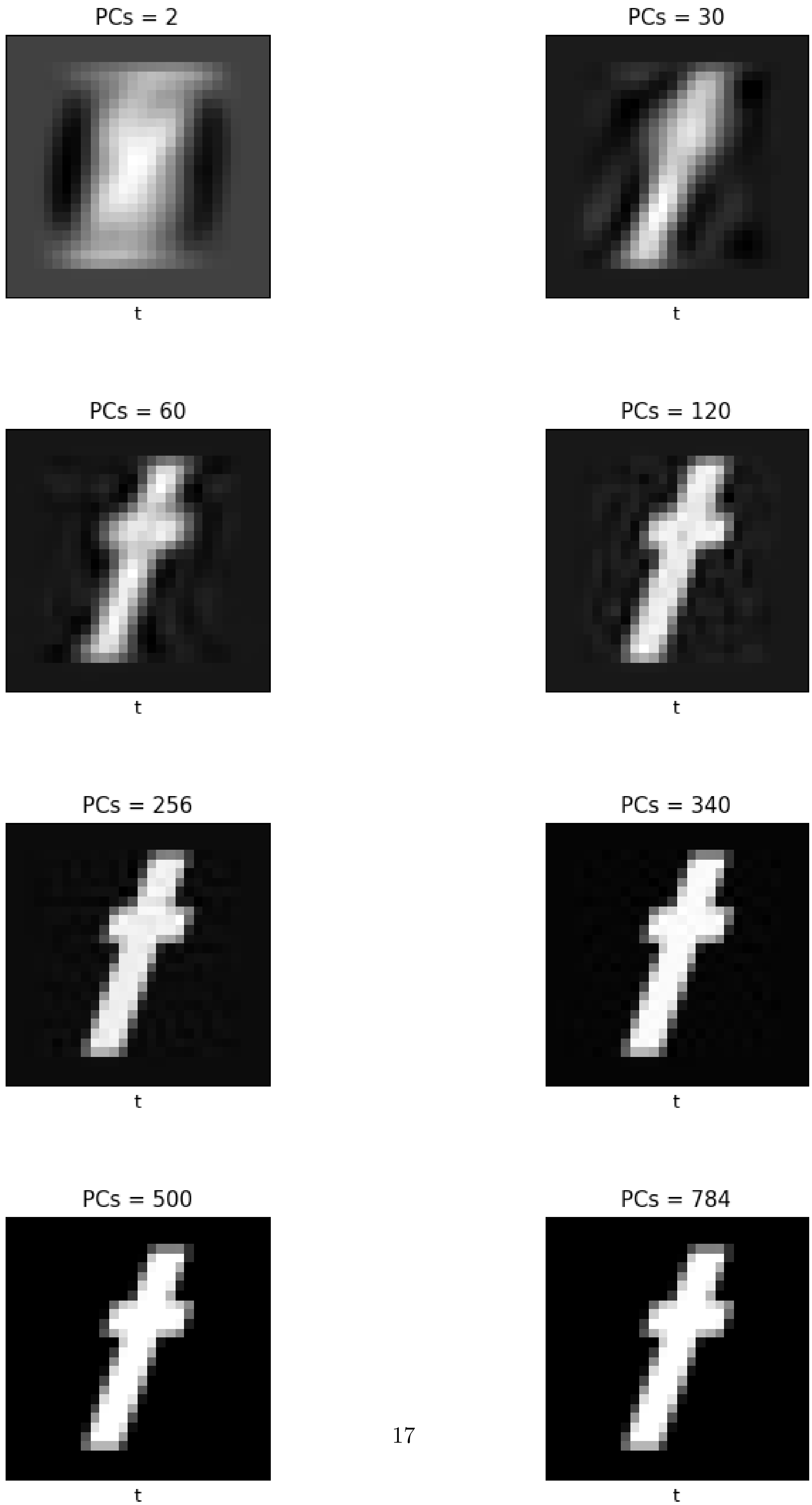
```
    y_label_color.append(el)
    y_label.append(y_train[el])

# Plot
layout = go.Layout(
    autosize=False,
    width=1000,
    height=1000,
    scene = dict(
        xaxis = dict(nticks=4, range=[-10,10],),
                     yaxis = dict(nticks=4, range=[-10,10],),
                     zaxis = dict(nticks=4, range=[-10,10],),))

fig = go.Figure(data=[go.Scatter3d(x=images_3d[:,0], y=images_3d[:,1],␣
↪z=images_3d[:,2],
                mode='markers', marker=dict(size=4, opacity=0.7,␣
↪color=y_label_color, showscale=True),
                                 text=['digit='+str(j) for j in y_label])],␣
↪layout=layout)
fig.show()
```

```
Projection shape= (5000, 3)
```

**We can also reconstruct our data from the result of the PCA using the function reconstruct(). We will generate several reconstructions for different values of principal components.**

[18]:
```
# Plot n images
n_components = [2, 30, 60, 120, 256, 340, 500, 784]

# Select random image
np.random.seed(42)
img_idx = np.random.randint(1, len(images))

plt.figure(figsize=(10, 15), dpi=90)
plt.subplots_adjust(hspace=0.5)
for i, nc in enumerate(n_components):
    plt.subplot(4, 2, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)

    pca_ncomp = PCA(n_components = nc)
    images_3d  = pca_ncomp.fit_transform(images)

    # Reconstruct data
    Xhat = pca_ncomp.inverse_transform(images_3d)
```

```
    # Convert each 1d array into 2d matrix
    img2d = array2img(Xhat[img_idx])

    # Set title
    plt.title(f'PCs = {nc}')

    # Get label for this image
    lb_img = labels[img_idx]
    plt.imshow(img2d, cmap=plt.cm.binary)
    plt.xlabel(lb_img)
plt.show()
```


PCs = 2
t
PCs = 30
t
PCs = 60
t
PCs = 120
t
PCs = 256
t
PCs = 340
t
PCs = 500
t
PCs = 784
t

It can be observed how if we increase the number of `principal components` the reconstructed image looks more like the original. Additionally, there is practically no difference between the image generated using `PCs=256` or `PCs=784`. Furthermore, it seems that the value we chose equal to 256, produces a reasonable result.

## 1.7 Classifiers

In this section we will implement different algorithms for classification:

- Random Forest
- KNN
- Convolutional Neural Network (CNN)

Few boosting algorithms (XGBoost, Gradient Boost and AdaBoost) were also tested (code available in appendix), but due to high run-time and comparative lower accuracy above 3 classifiers were finalized.

### 1.7.1 Random Forest

**Hyper Parameter Tuning**

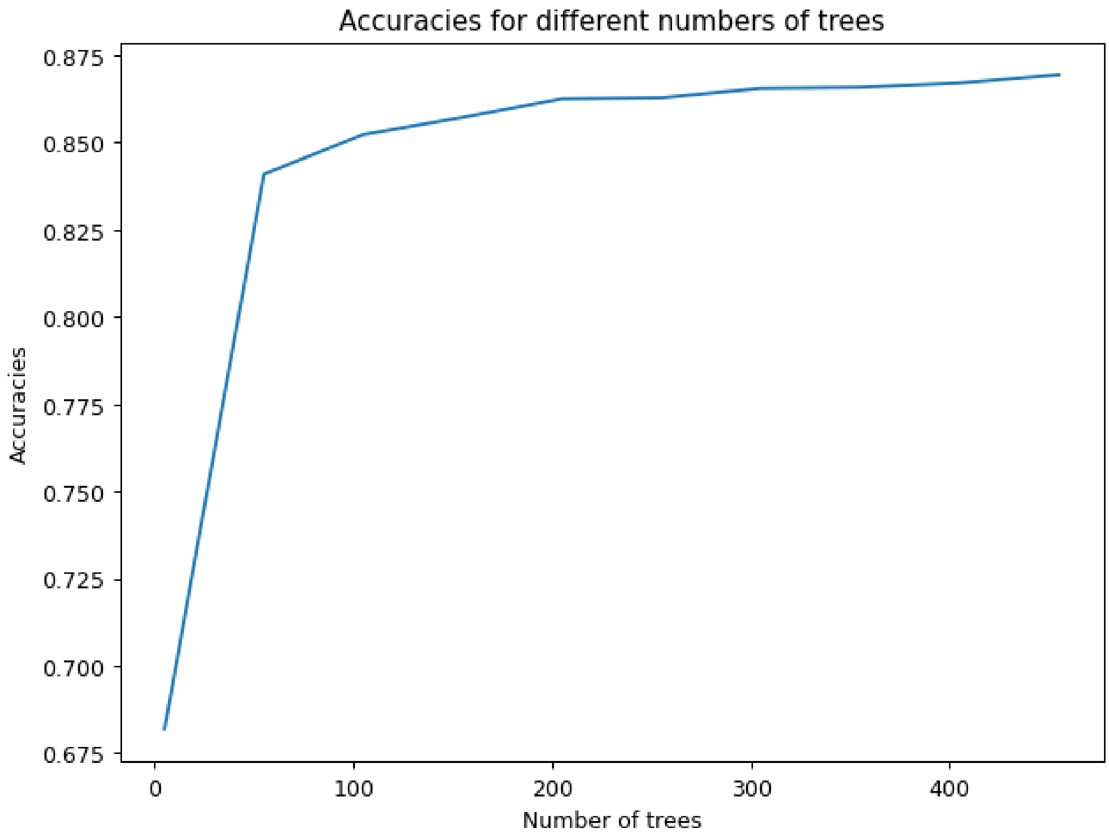
**Tunning the number of trees (n_estimators) to find the optimal value for the forest classifier**

[32]:
```
n_estimators = range(5, 500, 50)
estimators_accuracies = []

for estimator in n_estimators:
    random_forest = RandomForestClassifier(n_estimators = estimator,n_jobs = -1)
    random_forest.fit(X_train, y_train)
    estimators_accuracies.append(random_forest.score(X_val,y_val))

max_index = estimators_accuracies.index(max(estimators_accuracies))
best_number = n_estimators[max_index]
print("The best number of estimators to use for random forests is: {}.".
 ↪format(best_number))
```

```
The best number of estimators to use for random forests is: 455.
```

**Plot accuracy vs number of estimators**

[33]:
```
fig = plt.subplots(figsize=(8, 6), dpi=90)

plt.plot(n_estimators,estimators_accuracies)
plt.title("Accuracies for different numbers of trees")
plt.xlabel("Number of trees")
plt.ylabel("Accuracies")
plt.show()
```




**Tuning for best criterion to decide the split at each iteration**

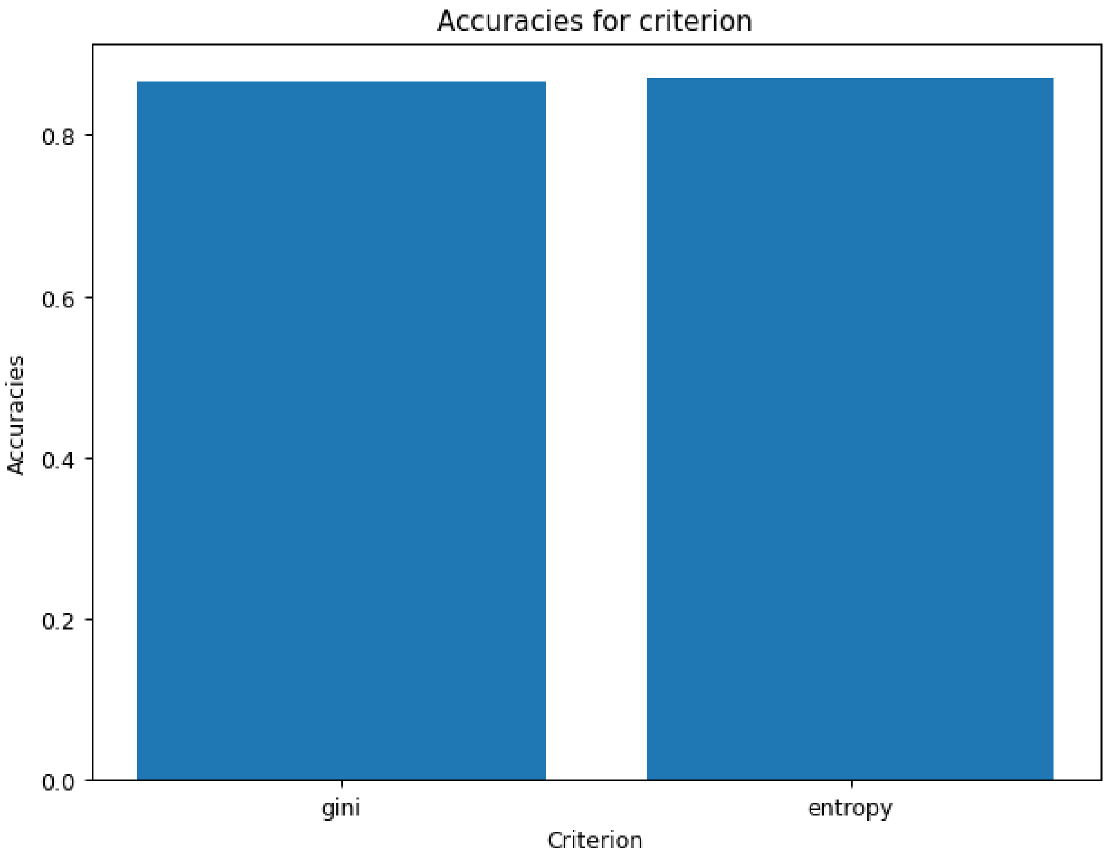
[39]:
```
criterions = ["gini", "entropy"]
criterions_accuracies = []

for criterion in criterions:
    random_forest = RandomForestClassifier(n_estimators =455,criterion =␣
↪criterion ,n_jobs = -1)
    random_forest.fit(X_train,y_train)
    criterions_accuracies.append(random_forest.score(X_val,y_val))
print("The best criterion to use for this random forest is {} and it gives an␣
↪accuracy of {:.2f}%.".format(criterions[criterions_accuracies.
↪index(max(criterions_accuracies))],max(criterions_accuracies)*100))
```

```
The best criterion to use for this random forest is entropy and it gives an
accuracy of 87.03%.
```

**Plot Accuracies for different criterion**

[40]:
```
fig = plt.subplots(figsize=(8, 6), dpi=90)
```

```
plt.bar(criterions,criterions_accuracies)
plt.title("Accuracies for criterion")
plt.xlabel("Criterion")
plt.ylabel("Accuracies")
plt.show()
```




**Tuning to establish how many features should be selected at each split**

[43]:
```
max_features = ["auto", "log2", 5,10,15,20]
max_features_accuracies = []

for feature in max_features:
    random_forest = RandomForestClassifier(n_estimators =455 ,n_jobs =␣
↪-1,max_features = feature)
    random_forest.fit(X_train,y_train)
    max_features_accuracies.append(random_forest.score(X_val,y_val))

print("The best distance measure to use for random forests is {} and it gives␣
↪an accuracy of {:.2f}%.".format(max_features[max_features_accuracies.
↪index(max(max_features_accuracies))],max(max_features_accuracies)*100))
```

```
The best distance measure to use for random forests is 10 and it gives an
accuracy of 86.81%.
```

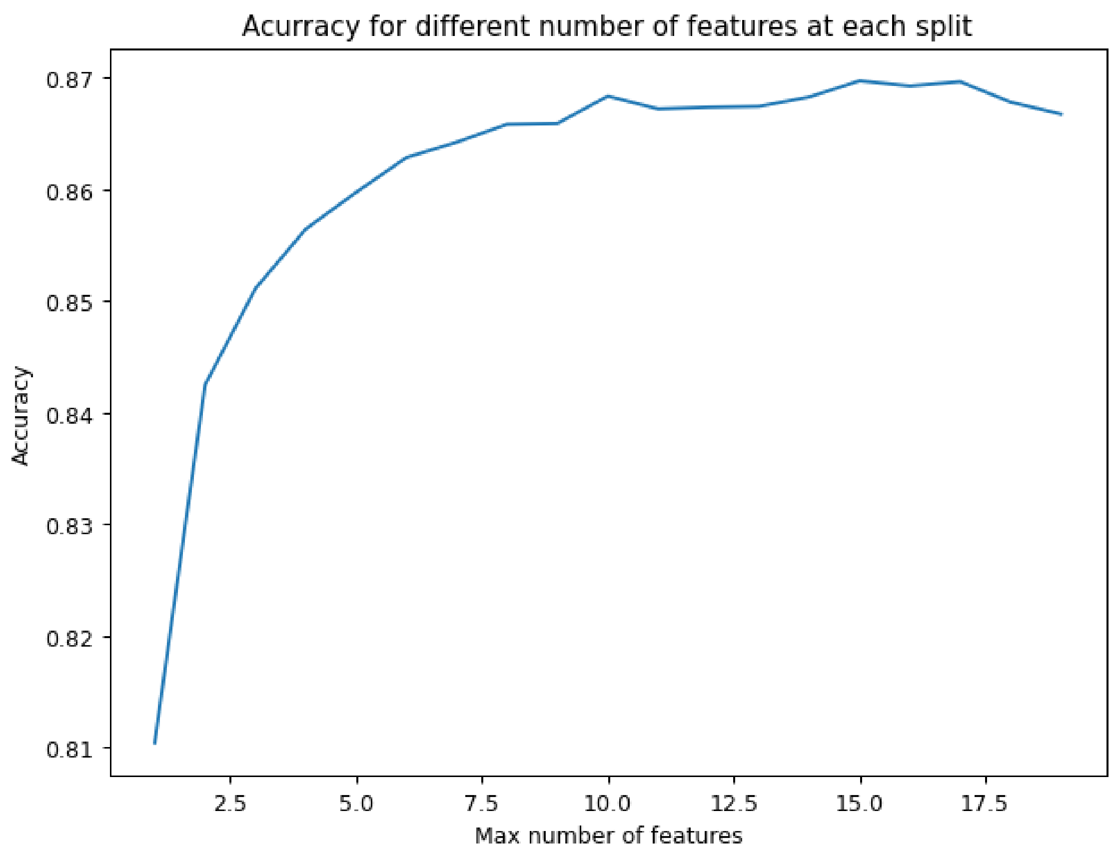
Second iteration for number of feature selection at each split

[44]:
```python
max_features_2 = range(1,20,1)
max_features_accuracies_val = []


for feature in max_features_2:
    random_forest = RandomForestClassifier(n_estimators =455 ,n_jobs =␣
↪-1,max_features = feature)
    random_forest.fit(X_train,y_train)
    max_features_accuracies_val.append(random_forest.score(X_val,y_val))

print("The best number of features to use for random forests is {} and it gives␣
↪an accuracy of {:.2f}%.".format(max_features_2[max_features_accuracies_val.
↪index(max(max_features_accuracies_val))],max(max_features_accuracies_val)*100))
```

```
The best number of features to use for random forests is 15 and it gives an
accuracy of 86.97%.
```

**Plot acurracy for different number of features at each split**

[45]:
```python
fig = plt.subplots(figsize=(8, 6), dpi=90)

plt.plot(max_features_2,max_features_accuracies_val)
plt.title("Acurracy for different number of features at each split")
plt.xlabel("Max number of features")
plt.ylabel("Accuracy")

plt.show()
```



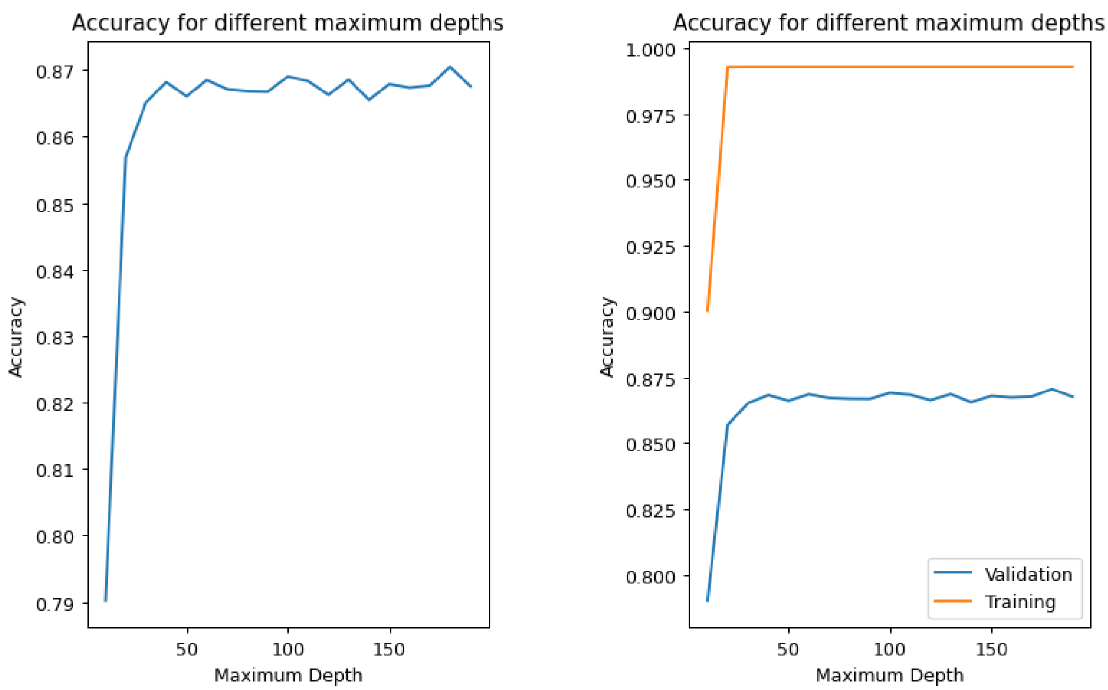

**Tuning maximum depth**

[50]:
```
max_depth = range(10, 200, 10)
depth_accuracies_train = []
depth_accuracies_val = []


for depth in max_depth:
    rf_depth = RandomForestClassifier(n_estimators =455 ,max_depth =␣
 ↪depth,n_jobs = -1)
    rf_depth.fit(X_train,y_train)
    depth_accuracies_val.append(rf_depth.score(X_val,y_val))
    depth_accuracies_train.append(rf_depth.score(X_train,y_train))

max_index = depth_accuracies_val.index(max(depth_accuracies_val))
best_depth = max_depth[max_index]
print("The best maximum depth setting to use for random forests is: {}.".
 ↪format(best_depth))
```

```
The best maximum depth setting to use for random forests is: 180.
```

**Plot Accuracy for different maximum depths**

[51]:
```
fig = plt.subplots(figsize=(10, 6), dpi=90)


plt.subplot(121)
plt.plot(max_depth,depth_accuracies_val)
plt.title("Accuracy for different maximum depths")
plt.xlabel("Maximum Depth")
plt.ylabel("Accuracy")


plt.subplot(122)
plt.plot(max_depth,depth_accuracies_val)
plt.plot(max_depth,depth_accuracies_train)
plt.title("Accuracy for different maximum depths")
plt.xlabel("Maximum Depth")
plt.ylabel("Accuracy")
plt.legend(["Validation","Training"])

plt.subplots_adjust(wspace = 0.5)
plt.show()
```

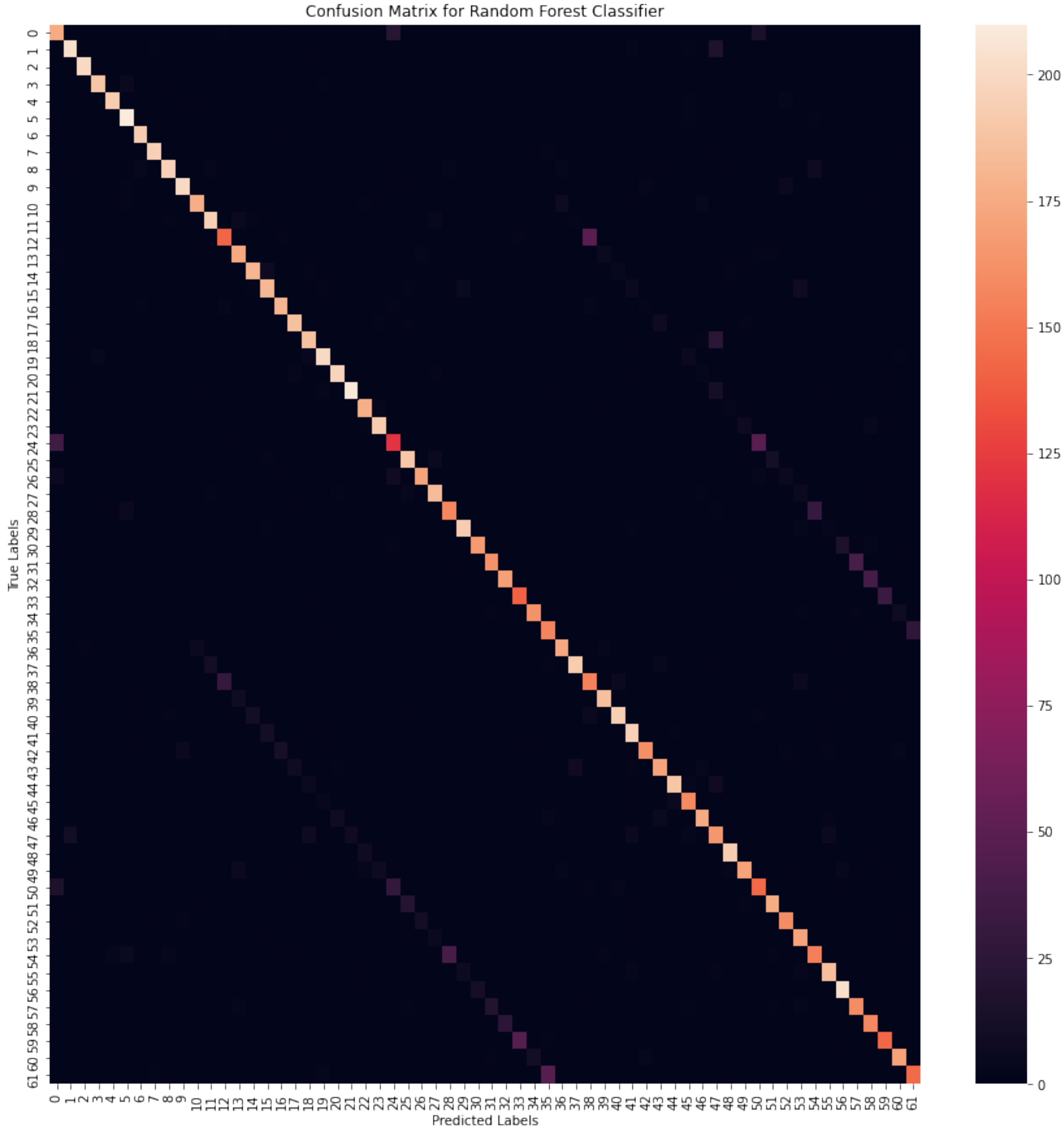



This gives the best parameters to use for this random forest: number of estimators = 455 Best Criterion to decide on tree splits = Entropy Maximum number of features = 15 Maximum depth = 180

**Random Forest confusion matrix on test set**

[16]:
```
random_forest_best = RandomForestClassifier(n_estimators = 455,criterion =␣
 ↪"entropy",max_features = 15,max_depth=180, n_jobs = -1)
random_forest_best.fit(X_train,y_train)
print("Best accuracy from Random Forest classifier is {:.2f}%.".
 ↪format((random_forest_best.score(X_test,y_test)*100)))

forest_predictions = random_forest_best.predict(X_test)
cm = confusion_matrix(y_test,forest_predictions)

plt.figure(figsize = (15,15))
ax = plt.subplot()
sns.heatmap(cm, ax=ax)
ax.set_ylabel("True Labels")
ax.set_xlabel("Predicted Labels")
ax.set_title("Confusion Matrix for Random Forest Classifier")
plt.show()
```

```
Best accuracy from Random Forest classifier is 87.02%.
```




**Random Forest metrics**

[24]:
```
print(classification_report(y_test,forest_predictions))
```

```
              precision    recall  f1-score   support

           0       0.73      0.82      0.77       214
           1       0.91      0.88      0.89       232
           2       0.97      0.97      0.97       206
           3       0.94      0.95      0.95       199
           4       0.95      0.95      0.95       202
           5       0.89      0.97      0.93       217
```

| | | | | |
|---|---|---|---|---|
| 6 | 0.94 | 0.98 | 0.96 | 197 |
| 7 | 0.96 | 0.97 | 0.96 | 201 |
| 8 | 0.95 | 0.88 | 0.91 | 224 |
| 9 | 0.95 | 0.94 | 0.95 | 213 |
| A | 0.95 | 0.89 | 0.92 | 199 |
| B | 0.91 | 0.91 | 0.91 | 214 |
| C | 0.78 | 0.71 | 0.75 | 199 |
| D | 0.88 | 0.90 | 0.89 | 193 |
| E | 0.91 | 0.88 | 0.90 | 206 |
| F | 0.88 | 0.87 | 0.88 | 210 |
| G | 0.91 | 0.91 | 0.91 | 199 |
| H | 0.92 | 0.92 | 0.92 | 204 |
| I | 0.89 | 0.85 | 0.87 | 220 |
| J | 0.88 | 0.91 | 0.90 | 219 |
| K | 0.92 | 0.95 | 0.94 | 207 |
| L | 0.94 | 0.90 | 0.92 | 231 |
| M | 0.88 | 0.95 | 0.91 | 187 |
| N | 0.92 | 0.92 | 0.92 | 208 |
| O | 0.65 | 0.58 | 0.61 | 208 |
| P | 0.83 | 0.89 | 0.86 | 213 |
| Q | 0.90 | 0.86 | 0.88 | 199 |
| R | 0.90 | 0.91 | 0.91 | 202 |
| S | 0.77 | 0.77 | 0.77 | 206 |
| T | 0.93 | 0.95 | 0.94 | 204 |
| U | 0.91 | 0.85 | 0.88 | 196 |
| V | 0.86 | 0.78 | 0.81 | 210 |
| W | 0.86 | 0.80 | 0.83 | 210 |
| X | 0.74 | 0.80 | 0.77 | 177 |
| Y | 0.93 | 0.90 | 0.91 | 182 |
| Z | 0.72 | 0.83 | 0.77 | 187 |
| a | 0.88 | 0.92 | 0.90 | 190 |
| b | 0.93 | 0.91 | 0.92 | 210 |
| c | 0.70 | 0.79 | 0.74 | 197 |
| d | 0.94 | 0.93 | 0.94 | 199 |
| e | 0.93 | 0.88 | 0.91 | 220 |
| f | 0.86 | 0.92 | 0.89 | 213 |
| g | 0.94 | 0.83 | 0.88 | 195 |
| h | 0.90 | 0.86 | 0.88 | 201 |
| i | 0.95 | 0.90 | 0.93 | 210 |
| j | 0.85 | 0.93 | 0.89 | 171 |
| k | 0.91 | 0.89 | 0.90 | 196 |
| l | 0.71 | 0.79 | 0.74 | 210 |
| m | 0.92 | 0.94 | 0.93 | 206 |
| n | 0.92 | 0.83 | 0.87 | 206 |
| o | 0.67 | 0.73 | 0.70 | 198 |
| p | 0.90 | 0.88 | 0.89 | 200 |
| q | 0.88 | 0.89 | 0.89 | 179 |
| r | 0.84 | 0.93 | 0.88 | 184 |

```
           s       0.79      0.72      0.75       214
           t       0.89      0.92      0.90       202
           u       0.89      0.92      0.90       221
           v       0.78      0.82      0.80       196
           w       0.75      0.84      0.79       189
           x       0.79      0.72      0.76       198
           y       0.89      0.89      0.89       193
           z       0.84      0.70      0.77       206

    accuracy                           0.87     12599
   macro avg       0.87      0.87      0.87     12599
weighted avg       0.87      0.87      0.87     12599
```

**10-fold cross validation**

**Combining training and validation set to complete**

[20]:
```
print(f"X_train shape: {X_train.shape}")
print(f"X_val shape: {X_val.shape}")

print(f"y_train shape: {y_train.shape}")
print(f"y_val shape: {y_val.shape}")

#Combine training and validation set into 1
X_train_val = np.vstack((X_train,X_val))

#Combine labels
y_train_val = np.concatenate((y_train,y_val))

print(f"X_train_val shape: {X_train_val.shape}")
print(f"y_train_val shape: {y_train_val.shape}")
```

```
X_train shape: (37290, 256)
X_val shape: (13103, 256)
y_train shape: (37290,)
y_val shape: (13103,)
X_train_val shape: (50393, 256)
y_train_val shape: (50393,)
```

[11]:
```
cross_val_k_fold_rf = RandomForestClassifier(n_estimators = 455,criterion =␣
 ↪"entropy",max_features = 15,max_depth=180, n_jobs = -1)
cross_val_scores_rf =␣
 ↪cross_val_score(cross_val_k_fold_rf,X_train_val,y_train_val,cv = 10)
print(cross_val_scores_rf)

print(f"The average accuracy of the random forest classifier using 10-fold␣
 ↪cross validation is: {cross_val_scores_rf.mean()}.")
```

```
[0.89047619 0.89027778 0.88511905 0.87874578 0.89422504 0.88648541
 0.87914269 0.88450089 0.88053185 0.88589006]
The average accuracy of the random forest classifier using 10-fold cross
validation is: 0.8855394746375099.
```

**Example plot of a tree from the forest This graph has a reduced maximum depth for graphing purposes**

[12]:
```python
ex_rf = RandomForestClassifier(n_estimators = 5,criterion = "entropy",max_depth = 4, n_jobs = -1)
ex_rf.fit(X_train,y_train)
fig = plt.subplots(figsize=(10, 10), dpi=90)

plot_tree(ex_rf.estimators_[0], filled = True, fontsize = 5)
plt.show()
```

### 1.7.2 K-nearest Neighbours

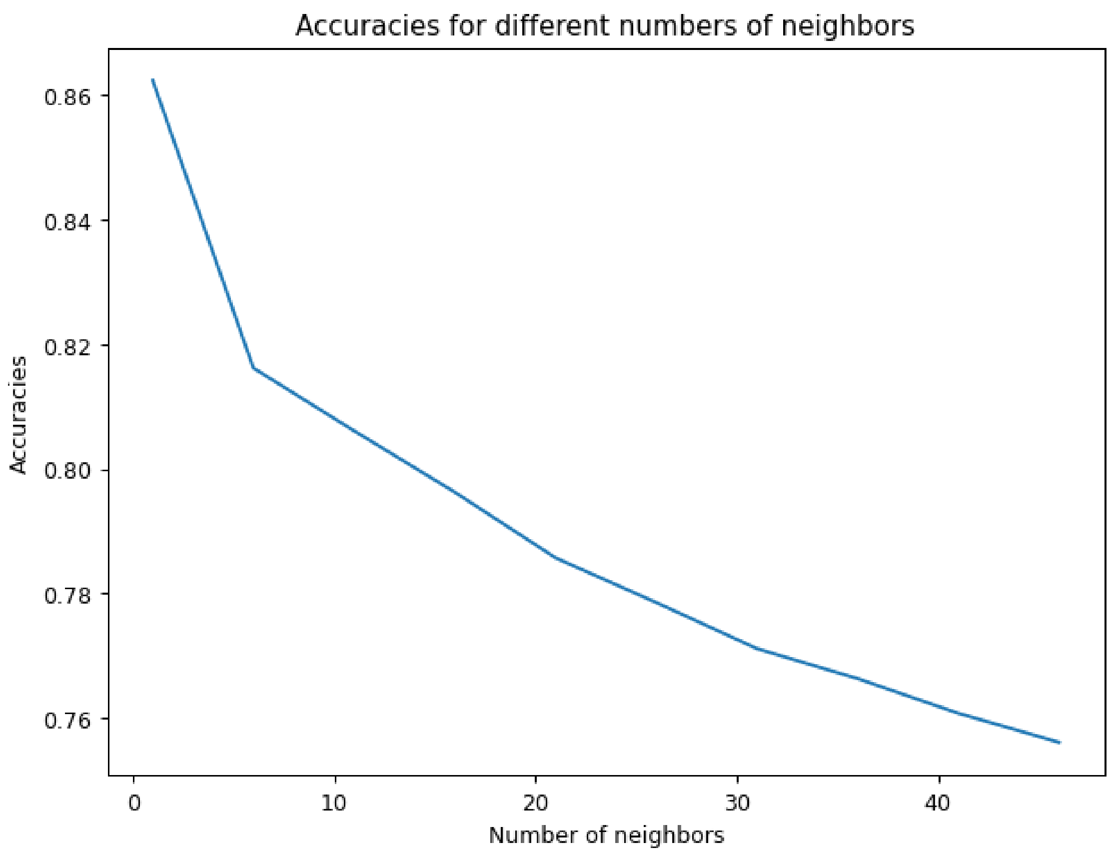
**Hyper Parameter Tuning**

**Finding the best k-value for this algorithm**

[19]:
```
numbers_k = range(1,50,5)
accuracies_k = []

for k in numbers_k:
    knn_model = KNeighborsClassifier(n_neighbors = k,n_jobs = -1)
    knn_model.fit(X_train,y_train)
    accuracies_k.append(knn_model.score(X_val,y_val))
```

**Plot Accuracies for different numbers of neighbors**

[20]:
```
fig = plt.subplots(figsize=(8, 6), dpi=90)

plt.plot(numbers_k,accuracies_k)
plt.title("Accuracies for different numbers of neighbors")
plt.xlabel("Number of neighbors")
plt.ylabel("Accuracies")
plt.show()
```

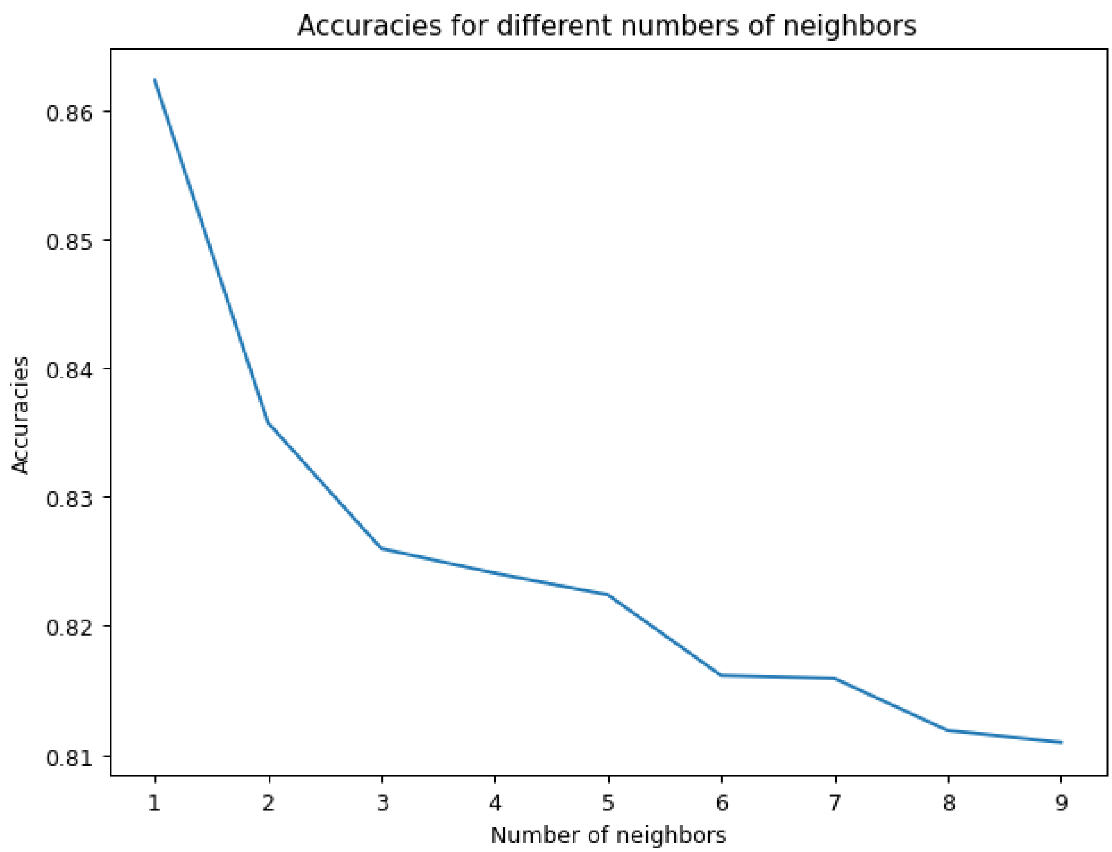


**Testing for smaller values as the graph above shows the accuracy drops drastically for larger values of k.**

[22]:
```
numbers_k = range(1,10,1)
accuracies_k = []

for k in numbers_k:
    knn_model = KNeighborsClassifier(n_neighbors = k)
    knn_model.fit(X_train,y_train)
    accuracies_k.append(knn_model.score(X_val,y_val))
```

**Plot Accuracies for different numbers of neighbors¶**

[23]:
```
fig = plt.subplots(figsize=(8, 6), dpi=90)

plt.plot(numbers_k,accuracies_k)
plt.title("Accuracies for different numbers of neighbors")
plt.xlabel("Number of neighbors")
plt.ylabel("Accuracies")
plt.show()
```



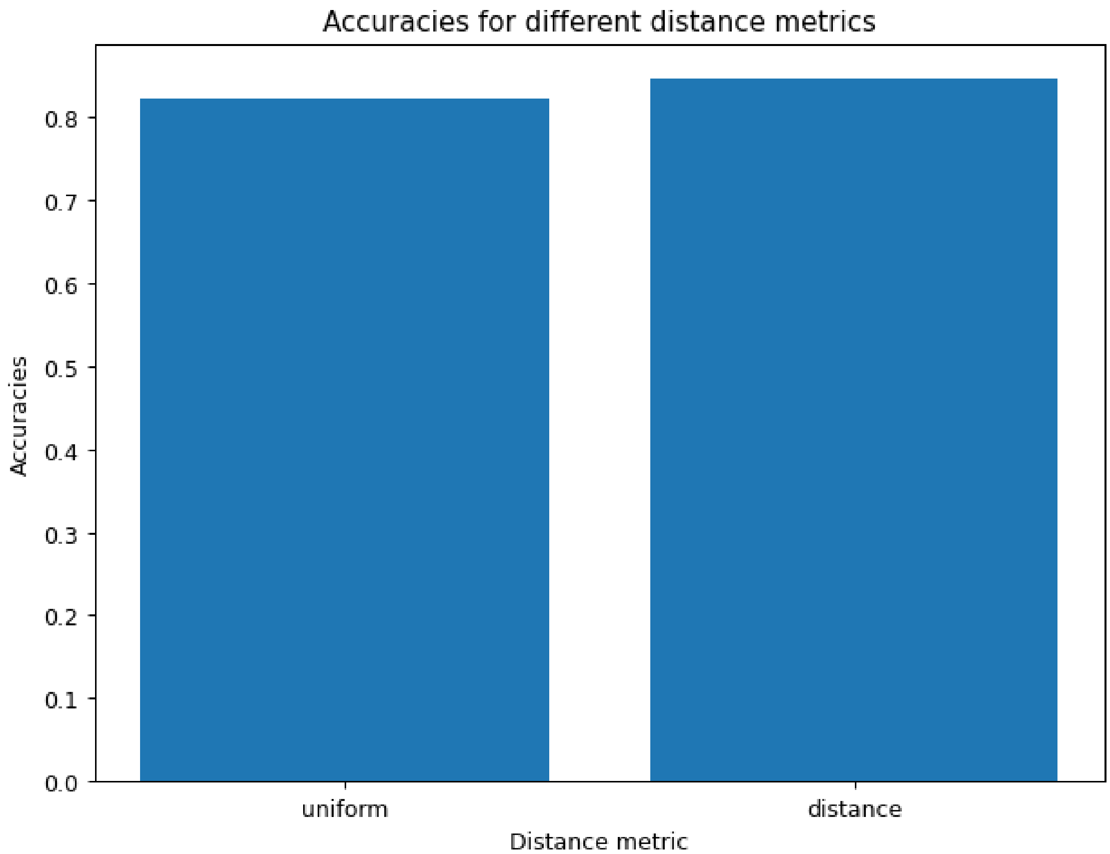
**Tunning weight metrics**

[24]:
```
weight_metrics = ["uniform","distance"]
accuracies_w = []

for weight in weight_metrics:
    knn_model = KNeighborsClassifier(weights = weight, n_jobs = -1)
    knn_model.fit(X_train,y_train)
    accuracies_w.append(knn_model.score(X_val,y_val))

print("The best distance measure to use for knn is {} and it gives an accuracy␣
↪of {:.2f}%.\nDistance refers to the weighted distance.".
↪format(weight_metrics[accuracies_w.
↪index(max(accuracies_w))],max(accuracies_w)*100))
```

```
The best distance measure to use for knn is distance and it gives an accuracy of
84.61%.
Distance refers to the weighted distance.
```

**Plot Accuracies for different distance metrics**

[25]:
```
fig = plt.subplots(figsize=(8, 6), dpi=90)

plt.bar(weight_metrics,accuracies_w)
plt.title("Accuracies for different distance metrics")
plt.xlabel("Distance metric")
plt.ylabel("Accuracies")
plt.show()
```



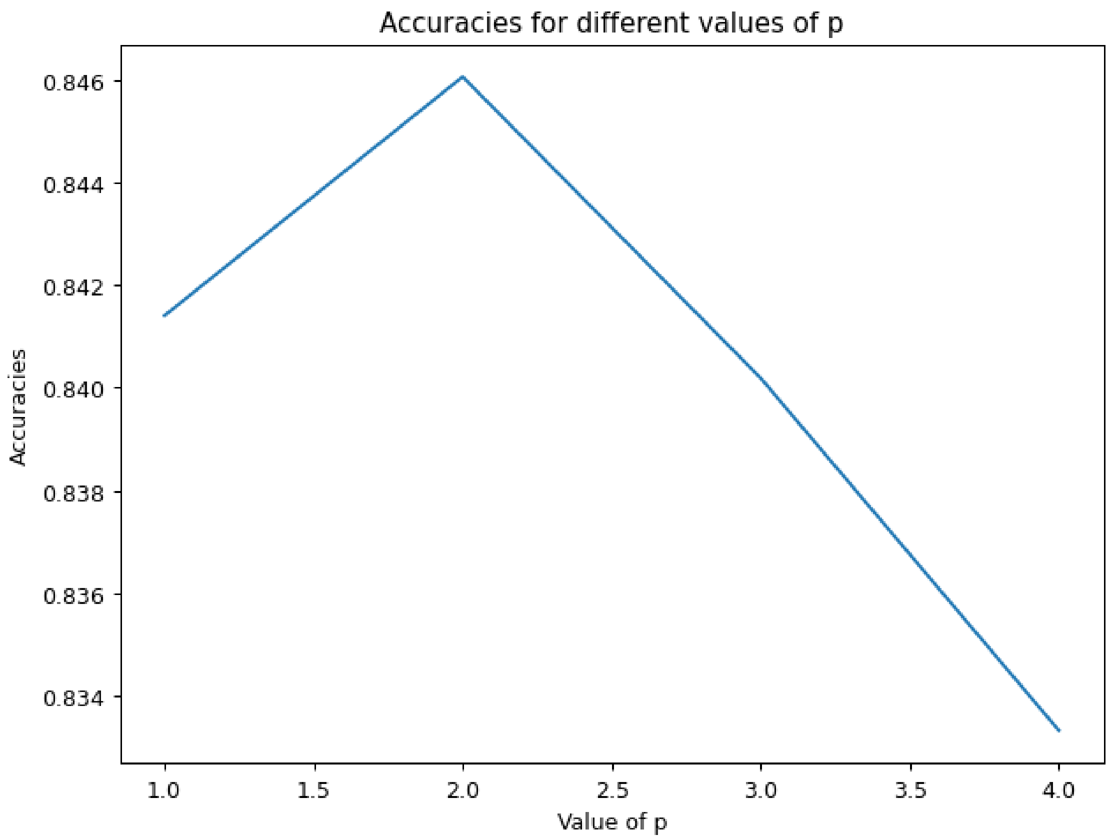

**Tunning for best power parameter, this is the parameter that is responsible for the distance metric used. ie p = 1 means "Minkowski" distance and p = 2 means "Euclidean" distance.**

[26]:
```
power_param = range(1,5)
accuracies_p = []

for power in power_param:
    knn_model = KNeighborsClassifier(weights = "distance",p = power, n_jobs =␣
 ↪-1)
    knn_model.fit(X_train,y_train)
    accuracies_p.append(knn_model.score(X_val,y_val))

print("The best power parameter to use for knn is {} and it gives an accuracy␣
 ↪of {:.2f}%.\nTherefore Minkowski distance is the best metric to use.".
 ↪format(power_param[accuracies_p.
 ↪index(max(accuracies_p))],max(accuracies_p)*100))
```

```
The best power parameter to use for knn is 2 and it gives an accuracy of 84.61%.
Therefore Minkowski distance is the best metric to use.
```

**Plot Accuracies for different values of p**

[27]:
```
fig = plt.subplots(figsize=(8, 6), dpi=90)

plt.plot(power_param,accuracies_p)
plt.title("Accuracies for different values of p")
plt.xlabel("Value of p")
plt.ylabel("Accuracies")
plt.show()
```



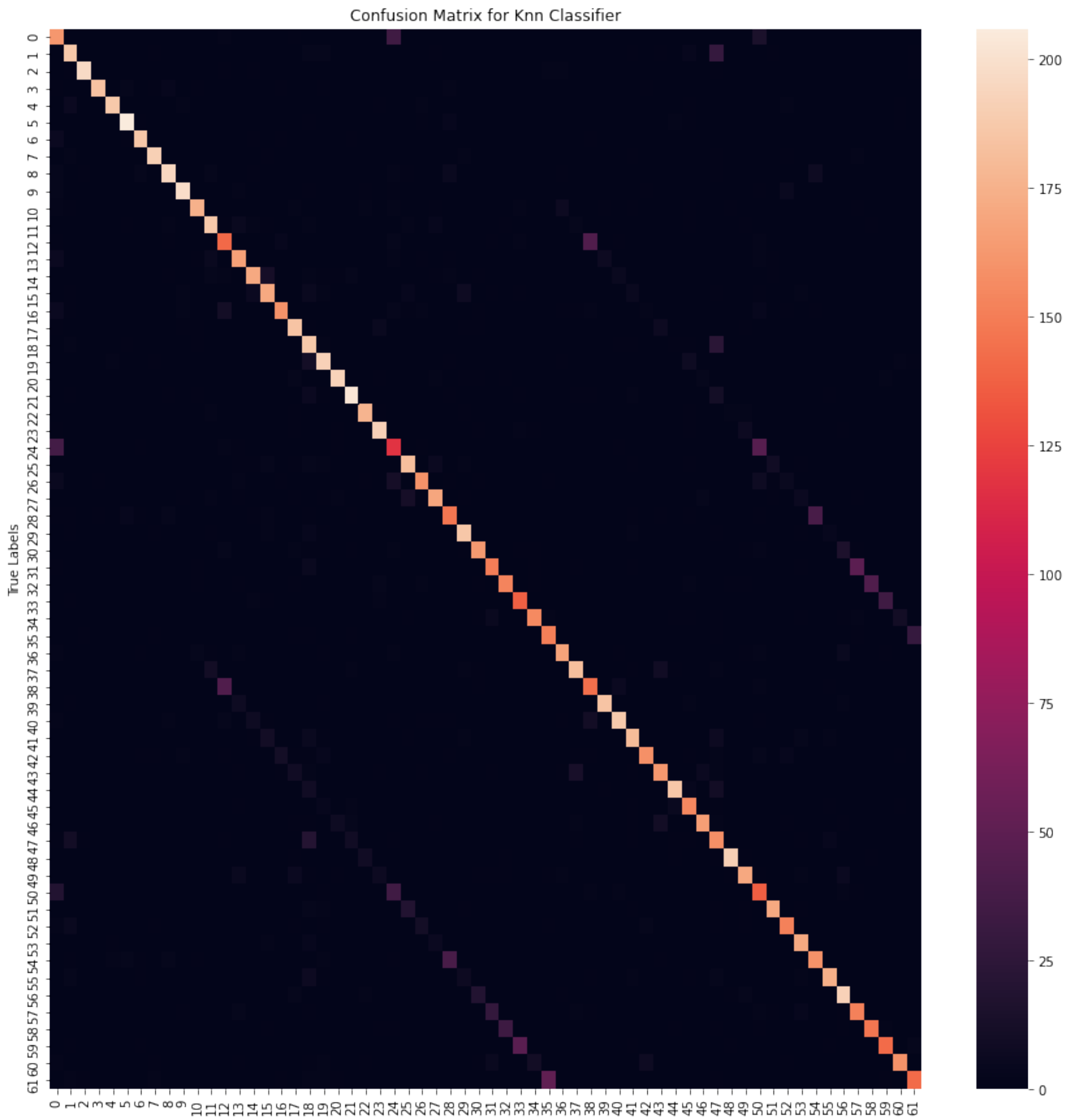

**K-Nearest Neighbor confusion matrix on test set**

**Using the best Knn classifier with parameters: k = 5 weights = "distance" (using weighted distance to establish the closest neighbors) p = 1 (Minkowski distance)**

[18]:
```
knn_best = KNeighborsClassifier(weights = "distance",p = 1, n_jobs = -1)
knn_best.fit(X_train,y_train)
knn_predictions = knn_best.predict(X_test)
print("Best accuracy from KNN classifier is {:.2f}%.".format(knn_best.
 ↪score(X_test,y_test)*100))
cm2 = confusion_matrix(y_test,knn_predictions)
```

```python
plt.figure(figsize = (15,15))
ax = plt.subplot()
sns.heatmap(cm2, ax=ax)
ax.set_ylabel("True Labels")
ax.set_xlabel("Predicted Labels")
ax.set_title("Confusion Matrix for Knn Classifier")
plt.show()
```

Best accuracy from KNN classifier is 83.81%.



**K-nearest neighbours metrics**

```
[25]: print(classification_report(y_test,knn_predictions))
```

```
              precision    recall  f1-score   support

           0       0.61      0.76      0.68       214
           1       0.82      0.81      0.81       232
           2       0.98      0.96      0.97       206
           3       0.99      0.92      0.96       199
           4       0.97      0.93      0.95       202
           5       0.94      0.95      0.94       217
           6       0.94      0.94      0.94       197
           7       0.95      0.95      0.95       201
           8       0.92      0.87      0.89       224
           9       0.95      0.93      0.94       213
           A       0.94      0.88      0.91       199
           B       0.87      0.87      0.87       214
           C       0.67      0.71      0.69       199
           D       0.83      0.87      0.85       193
           E       0.88      0.83      0.85       206
           F       0.83      0.81      0.82       210
           G       0.89      0.80      0.84       199
           H       0.84      0.91      0.87       204
           I       0.64      0.85      0.73       220
           J       0.84      0.88      0.86       219
           K       0.93      0.93      0.93       207
           L       0.90      0.88      0.89       231
           M       0.92      0.95      0.93       187
           N       0.90      0.92      0.91       208
           O       0.54      0.57      0.55       208
           P       0.82      0.85      0.83       213
           Q       0.87      0.80      0.84       199
           R       0.88      0.84      0.86       202
           S       0.72      0.71      0.72       206
           T       0.87      0.91      0.89       204
           U       0.84      0.83      0.84       196
           V       0.78      0.71      0.75       210
           W       0.81      0.73      0.76       210
           X       0.71      0.77      0.74       177
           Y       0.92      0.86      0.89       182
           Z       0.72      0.81      0.76       187
           a       0.91      0.88      0.90       190
           b       0.88      0.87      0.87       210
           c       0.70      0.73      0.71       197
           d       0.95      0.93      0.94       199
           e       0.92      0.85      0.88       220
           f       0.88      0.85      0.86       213
```

```
           g       0.91      0.82      0.86       195
           h       0.85      0.81      0.83       201
           i       0.93      0.89      0.91       210
           j       0.82      0.91      0.86       171
           k       0.93      0.84      0.88       196
           l       0.60      0.75      0.67       210
           m       0.96      0.93      0.95       206
           n       0.89      0.83      0.86       206
           o       0.59      0.69      0.64       198
           p       0.92      0.85      0.89       200
           q       0.88      0.85      0.87       179
           r       0.87      0.93      0.90       184
           s       0.77      0.75      0.76       214
           t       0.93      0.86      0.89       202
           u       0.84      0.87      0.86       221
           v       0.72      0.78      0.75       196
           w       0.77      0.78      0.77       189
           x       0.78      0.72      0.75       198
           y       0.91      0.83      0.87       193
           z       0.80      0.69      0.74       206

    accuracy                           0.84     12599
   macro avg       0.84      0.84      0.84     12599
weighted avg       0.84      0.84      0.84     12599
```

**10-fold cross validation**

[21]:
```
cross_val_k_fold_knn = KNeighborsClassifier(weights = "distance",p = 1, n_jobs␣
 ↪= -1)
cross_val_scores_knn =␣
 ↪cross_val_score(cross_val_k_fold_knn,X_train_val,y_train_val,cv = 10)

print(cross_val_scores_knn)

print(f"The average accuracy of the K-nearest neighbours classifier using␣
 ↪10-fold cross validation is: {cross_val_scores_knn.mean()}.")
```

```
[0.85952381 0.86011905 0.86071429 0.84977178 0.861282   0.85096249
 0.84659655 0.84600119 0.85155785 0.85552689]
The average accuracy of the K-nearest neighbours classifier using 10-fold cross
validation is: 0.8542055892609077.
```

### 1.7.3 Convolutional Neural Network

**In this section, we will implement a Convolutional Neural Network (CNN) using Keras. Firstly, we need to reshape X__train, X__val and X__test to create the proper inputs for our CNN.**

**Preparing input data**

[9]:
```
# Reshaping
X_train_input = X_train.reshape(X_train.shape[0], 16, 16)
X_val_input = X_val.reshape(X_val.shape[0], 16, 16)
X_test_input = X_test.reshape(X_test.shape[0], 16, 16)

X_train_input = X_train_input[..., np.newaxis]
X_val_input = X_val_input[..., np.newaxis]
X_test_input = X_test_input[..., np.newaxis]
```

**Additionally, we need to create an one-hot vector for our labels.**

[10]:
```
# Create one-hot array
le = LabelEncoder()
y_train_input = le.fit_transform(y_train)
y_val_input = le.fit_transform(y_val)
y_test_input = le.fit_transform(y_test)

y_train_input = to_categorical(y_train_input)
y_val_input = to_categorical(y_val_input)
y_test_input = to_categorical(y_test_input)
```

**The new dimensions are:**

[11]:
```
print(f"X_train shape: {X_train_input.shape}"),
print(f"X_val shape: {X_val_input.shape}")
print(f"X_test shape: {X_test_input.shape}")
print(f"\ny_train shape: {y_train_input.shape}")
print(f"y_val shape: {y_val_input.shape}")
print(f"y_test shape: {y_test_input.shape}")
```

```
X_train shape: (37290, 16, 16, 1)
X_val shape: (13103, 16, 16, 1)
X_test shape: (12599, 16, 16, 1)

y_train shape: (37290, 62)
y_val shape: (13103, 62)
y_test shape: (12599, 62)
```

### 1.7.4 Defining the architecture of our CNN

**We define our model and train it using the GPU.**

[12]:
```
def create_model(n_unq_labels=62,
                 filters=(32, 64, 128),
                 kernel_size=(3, 3, 3),
                 input_shape=(16, 16, 1)):
  """
    Function to initialize a CNN model.
```

```
    :param filters: Number of filters for the CNN layers
    :param kernel_size: Kernel sizes
    :param input_shape: Input shape - i.e (16, 16, 1)
    :param n_unq_label: Number of unique labels
    :return model: Instance model
    """

    # Init model
    model = Sequential()

    # Add layers
    model.add(Conv2D(filters[0], kernel_size=kernel_size[0], activation="relu",␣
↪input_shape=input_shape))
    model.add(Conv2D(filters[1], kernel_size=kernel_size[1], activation="relu"))
    model.add(Conv2D(filters[2], kernel_size=kernel_size[2], activation="relu"))

    # Flatten and generate output
    model.add(Flatten())
    model.add(Dense(n_unq_labels, activation="softmax"))

    # Compile using adam as optimizer and categorical crossentropy as loss␣
↪function
    model.compile(optimizer='adam', loss='categorical_crossentropy',␣
↪metrics=['accuracy'])
    return model
```

**Fine-Tuning** Once we have defined the architecture of our neural network we can perform the fine-tuning of the CNN model. We will try to find the optimal values for: - Number of filters - Number of kernel

As we did before, we will use `grid_search()`:

[13]:
```
model = KerasClassifier(build_fn=create_model, epochs=15, verbose=1)

param_grid = dict(filters=[(32, 64, 128), (32, 128, 356)],
                  kernel_size=[(3, 3, 3), (3, 4, 5)])
grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=3)
grid_result = grid.fit(X_train_input, y_train_input,verbose=0)
```

```
389/389 [==============================] - 1s 3ms/step - loss: 0.7179 -
accuracy: 0.8389
389/389 [==============================] - 1s 3ms/step - loss: 0.7950 -
accuracy: 0.8433
389/389 [==============================] - 1s 3ms/step - loss: 0.7868 -
accuracy: 0.8397
389/389 [==============================] - 1s 3ms/step - loss: 0.7706 -
accuracy: 0.8360
```

```
389/389 [==============================] - 1s 3ms/step - loss: 0.7601 -
accuracy: 0.8400
389/389 [==============================] - 1s 3ms/step - loss: 0.7625 -
accuracy: 0.8364
389/389 [==============================] - 3s 8ms/step - loss: 0.6471 -
accuracy: 0.8460
389/389 [==============================] - 3s 8ms/step - loss: 0.6488 -
accuracy: 0.8467
389/389 [==============================] - 3s 8ms/step - loss: 0.6765 -
accuracy: 0.8412
389/389 [==============================] - 4s 9ms/step - loss: 0.7280 -
accuracy: 0.8447
389/389 [==============================] - 4s 9ms/step - loss: 0.7413 -
accuracy: 0.8456
389/389 [==============================] - 4s 9ms/step - loss: 0.6832 -
accuracy: 0.8423
```

[13]:

```
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.846205 using {'filters': (32, 128, 356), 'kernel_size': (3, 3, 3)}
0.841566 (0.002405) with: {'filters': (32, 64, 128), 'kernel_size': (3, 3, 3)}
0.836149 (0.004191) with: {'filters': (32, 64, 128), 'kernel_size': (3, 4, 5)}
0.846205 (0.001388) with: {'filters': (32, 128, 356), 'kernel_size': (3, 3, 3)}
0.844409 (0.002219) with: {'filters': (32, 128, 356), 'kernel_size': (3, 4, 5)}
```

The optimal values for filters are `(32, 128, 356)` and for kernel_size `(3, 3, 3)`. Now, we will train the model with these values and evaluate it on the test set.

[30]:

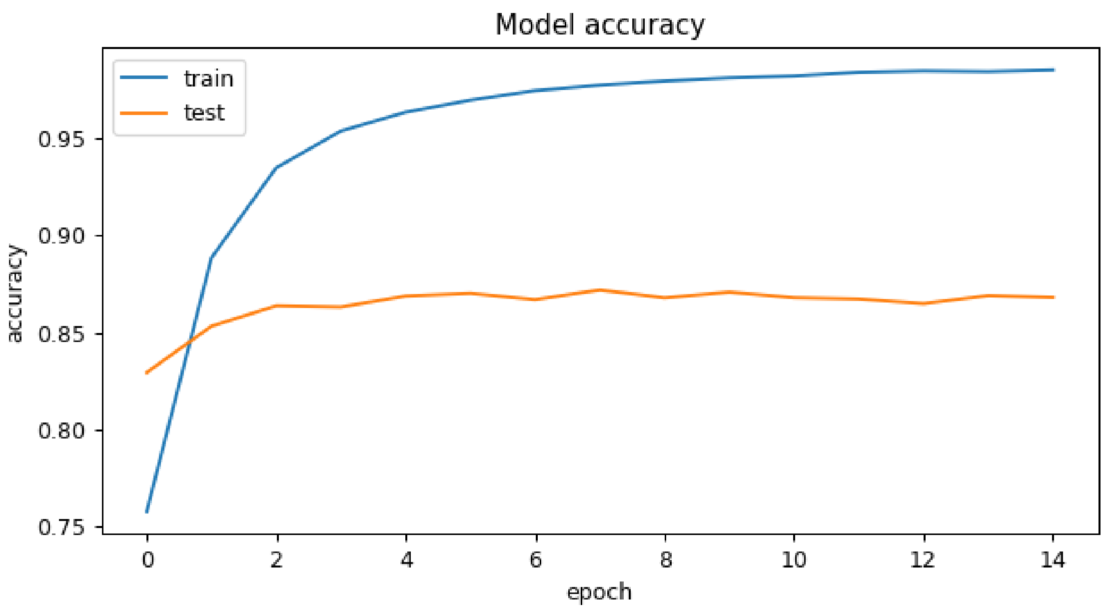
```
filters = (32, 128,356)
kernel_size = (3, 3, 3)
input_shape = (16, 16, 1)
n_unq_labels = len(set(labels))

model = create_model(filters=filters,
                     kernel_size=kernel_size,
                     input_shape=input_shape,
                     n_unq_labels=n_unq_labels)

hist = model.fit(X_train_input,
                 y_train_input,
                 validation_data=(X_val_input, y_val_input),
                 epochs=15)
```

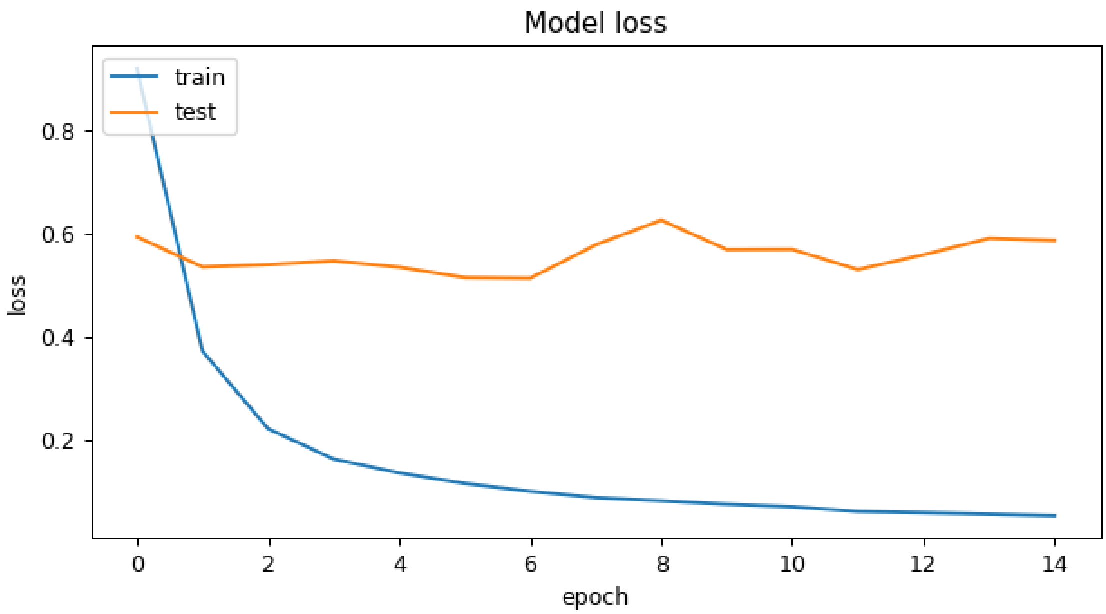
```
Epoch 1/15
1166/1166 [==============================] - 44s 38ms/step - loss: 0.9143 -
accuracy: 0.7580 - val_loss: 0.5843 - val_accuracy: 0.8287
Epoch 2/15
1166/1166 [==============================] - 43s 37ms/step - loss: 0.3705 -
accuracy: 0.8880 - val_loss: 0.5173 - val_accuracy: 0.8561
Epoch 3/15
1166/1166 [==============================] - 43s 37ms/step - loss: 0.2218 -
accuracy: 0.9357 - val_loss: 0.5105 - val_accuracy: 0.8645
Epoch 4/15
1166/1166 [==============================] - 43s 37ms/step - loss: 0.1641 -
accuracy: 0.9532 - val_loss: 0.5400 - val_accuracy: 0.8657
Epoch 5/15
1166/1166 [==============================] - 45s 39ms/step - loss: 0.1336 -
accuracy: 0.9639 - val_loss: 0.5593 - val_accuracy: 0.8641
Epoch 6/15
1166/1166 [==============================] - 46s 39ms/step - loss: 0.1164 -
accuracy: 0.9703 - val_loss: 0.5470 - val_accuracy: 0.8684
Epoch 7/15
1166/1166 [==============================] - 44s 38ms/step - loss: 0.1001 -
accuracy: 0.9742 - val_loss: 0.5432 - val_accuracy: 0.8673
Epoch 8/15
1166/1166 [==============================] - 45s 39ms/step - loss: 0.0890 -
accuracy: 0.9777 - val_loss: 0.5312 - val_accuracy: 0.8681
Epoch 9/15
1166/1166 [==============================] - 43s 37ms/step - loss: 0.0824 -
accuracy: 0.9791 - val_loss: 0.5553 - val_accuracy: 0.8689
Epoch 10/15
1166/1166 [==============================] - 45s 39ms/step - loss: 0.0751 -
accuracy: 0.9809 - val_loss: 0.5527 - val_accuracy: 0.8704
Epoch 11/15
1166/1166 [==============================] - 45s 39ms/step - loss: 0.0664 -
accuracy: 0.9833 - val_loss: 0.5384 - val_accuracy: 0.8683
Epoch 12/15
1166/1166 [==============================] - 46s 39ms/step - loss: 0.0620 -
accuracy: 0.9842 - val_loss: 0.5491 - val_accuracy: 0.8683
Epoch 13/15
1166/1166 [==============================] - 45s 38ms/step - loss: 0.0600 -
accuracy: 0.9845 - val_loss: 0.5714 - val_accuracy: 0.8709
Epoch 14/15
1166/1166 [==============================] - 45s 39ms/step - loss: 0.0552 -
accuracy: 0.9851 - val_loss: 0.5877 - val_accuracy: 0.8667
Epoch 15/15
1166/1166 [==============================] - 45s 39ms/step - loss: 0.0545 -
accuracy: 0.9852 - val_loss: 0.5886 - val_accuracy: 0.8678
```

**Plot Model accuracy and Model Loss**

[15]:

```
fig = plt.subplots(figsize=(8, 4), dpi=90)

# Display Accuracy
plt.plot(hist.history['accuracy'])
plt.plot(hist.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# Display Loss
fig = plt.subplots(figsize=(8, 4), dpi=90)
plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```





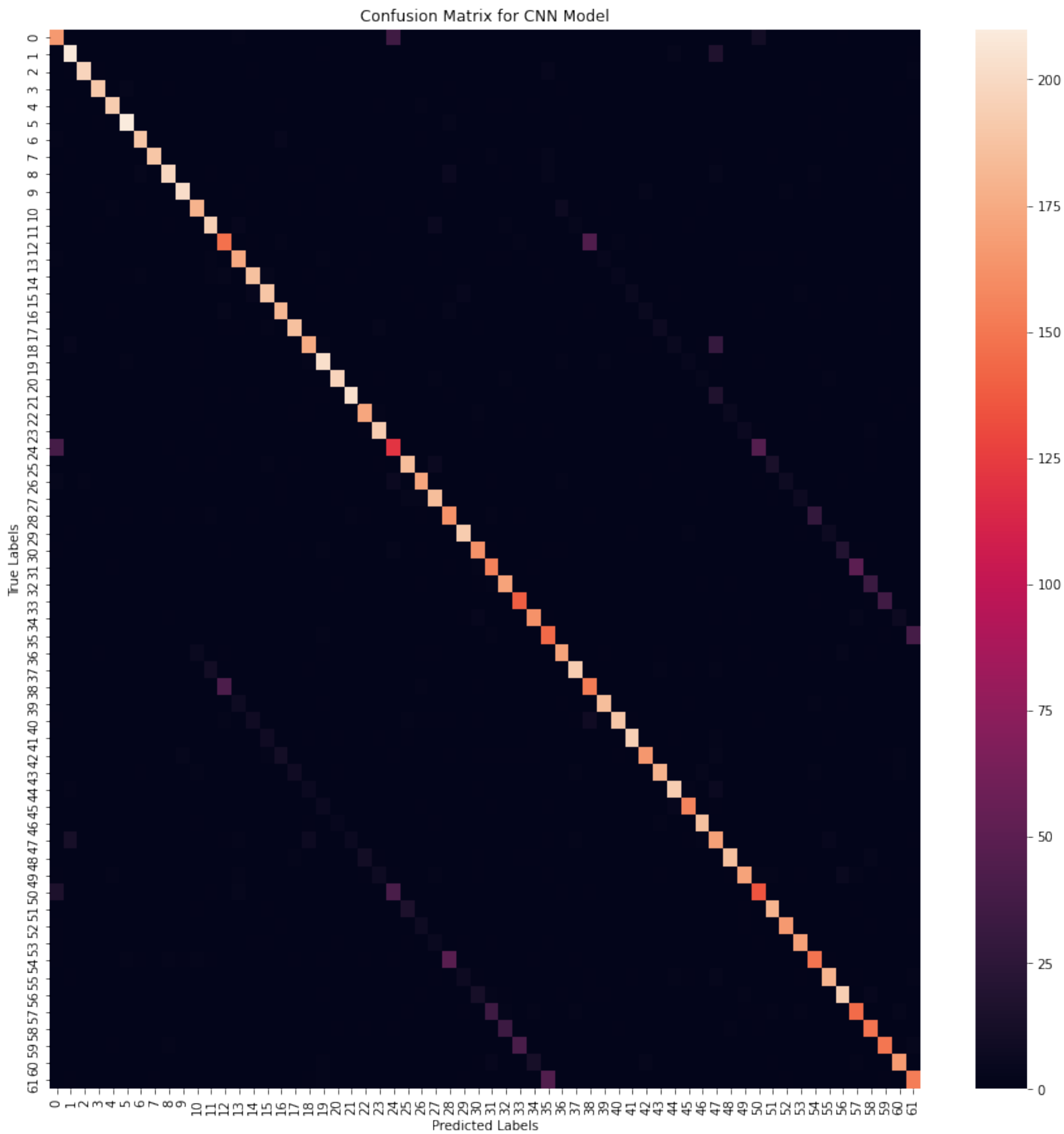

**We can evaluate the performance of the Convolutional Neural Network (CNN) on the test set using the function .evaluate()**

[31]:
```
score = model.evaluate(X_test_input, y_test_input, verbose=0)
print(f'Test loss: {round(score[0], 2)} / Test accuracy: {round(score[1], 2)}')
print(f'Test loss: {round(score[0], 2)} / Test accuracy: {score[1]}')
```

```
Test loss: 0.61 / Test accuracy: 0.87
Test loss: 0.61 / Test accuracy: 0.8663386106491089
```

The confusion matrix can be generated using:

[32]:
```
# Generate predictions for test set
y_preds = model.predict(X_test_input)

cm_cnn = confusion_matrix(y_test_input.argmax(axis=1), y_preds.argmax(axis=1))

plt.figure(figsize = (15,15))
ax = plt.subplot()
sns.heatmap(cm_cnn, ax=ax)
ax.set_ylabel("True Labels")
ax.set_xlabel("Predicted Labels")
ax.set_title("Confusion Matrix for CNN Model")
plt.show()
```




### CNN metrics

[33]:
```
print(classification_report(y_test_input.argmax(axis=1), y_preds.
↪argmax(axis=1)))
```

```
              precision    recall  f1-score   support

           0       0.70      0.78      0.74       214
           1       0.89      0.90      0.90       232
           2       0.99      0.96      0.97       206
           3       0.97      0.96      0.96       199
           4       0.96      0.95      0.96       202
```

| | | | | |
|---|---|---|---|---|
| 5 | 0.96 | 0.97 | 0.96 | 217 |
| 6 | 0.96 | 0.96 | 0.96 | 197 |
| 7 | 0.99 | 0.94 | 0.97 | 201 |
| 8 | 0.96 | 0.89 | 0.92 | 224 |
| 9 | 0.98 | 0.95 | 0.96 | 213 |
| 10 | 0.92 | 0.90 | 0.91 | 199 |
| 11 | 0.90 | 0.91 | 0.91 | 214 |
| 12 | 0.72 | 0.74 | 0.73 | 199 |
| 13 | 0.88 | 0.90 | 0.89 | 193 |
| 14 | 0.90 | 0.91 | 0.91 | 206 |
| 15 | 0.90 | 0.90 | 0.90 | 210 |
| 16 | 0.90 | 0.91 | 0.91 | 199 |
| 17 | 0.92 | 0.92 | 0.92 | 204 |
| 18 | 0.88 | 0.80 | 0.84 | 220 |
| 19 | 0.88 | 0.93 | 0.90 | 219 |
| 20 | 0.95 | 0.95 | 0.95 | 207 |
| 21 | 0.95 | 0.88 | 0.92 | 231 |
| 22 | 0.91 | 0.93 | 0.92 | 187 |
| 23 | 0.91 | 0.93 | 0.92 | 208 |
| 24 | 0.57 | 0.58 | 0.58 | 208 |
| 25 | 0.88 | 0.87 | 0.87 | 213 |
| 26 | 0.87 | 0.87 | 0.87 | 199 |
| 27 | 0.88 | 0.92 | 0.90 | 202 |
| 28 | 0.71 | 0.79 | 0.75 | 206 |
| 29 | 0.91 | 0.94 | 0.93 | 204 |
| 30 | 0.84 | 0.83 | 0.83 | 196 |
| 31 | 0.74 | 0.74 | 0.74 | 210 |
| 32 | 0.80 | 0.81 | 0.80 | 210 |
| 33 | 0.74 | 0.79 | 0.76 | 177 |
| 34 | 0.88 | 0.90 | 0.89 | 182 |
| 35 | 0.70 | 0.77 | 0.73 | 187 |
| 36 | 0.91 | 0.90 | 0.90 | 190 |
| 37 | 0.94 | 0.92 | 0.93 | 210 |
| 38 | 0.72 | 0.77 | 0.75 | 197 |
| 39 | 0.95 | 0.93 | 0.94 | 199 |
| 40 | 0.96 | 0.86 | 0.91 | 220 |
| 41 | 0.96 | 0.92 | 0.94 | 213 |
| 42 | 0.92 | 0.85 | 0.88 | 195 |
| 43 | 0.93 | 0.89 | 0.91 | 201 |
| 44 | 0.92 | 0.92 | 0.92 | 210 |
| 45 | 0.92 | 0.91 | 0.92 | 171 |
| 46 | 0.94 | 0.95 | 0.95 | 196 |
| 47 | 0.65 | 0.81 | 0.72 | 210 |
| 48 | 0.95 | 0.91 | 0.93 | 206 |
| 49 | 0.92 | 0.83 | 0.87 | 206 |
| 50 | 0.67 | 0.68 | 0.68 | 198 |
| 51 | 0.89 | 0.90 | 0.89 | 200 |
| 52 | 0.94 | 0.93 | 0.93 | 179 |

```
          53       0.92      0.92      0.92       184
          54       0.78      0.70      0.73       214
          55       0.92      0.89      0.91       202
          56       0.85      0.88      0.87       221
          57       0.71      0.74      0.72       196
          58       0.76      0.79      0.77       189
          59       0.77      0.76      0.77       198
          60       0.90      0.87      0.88       193
          61       0.75      0.74      0.75       206

    accuracy                           0.87     12599
   macro avg       0.87      0.87      0.87     12599
weighted avg       0.87      0.87      0.87     12599
```

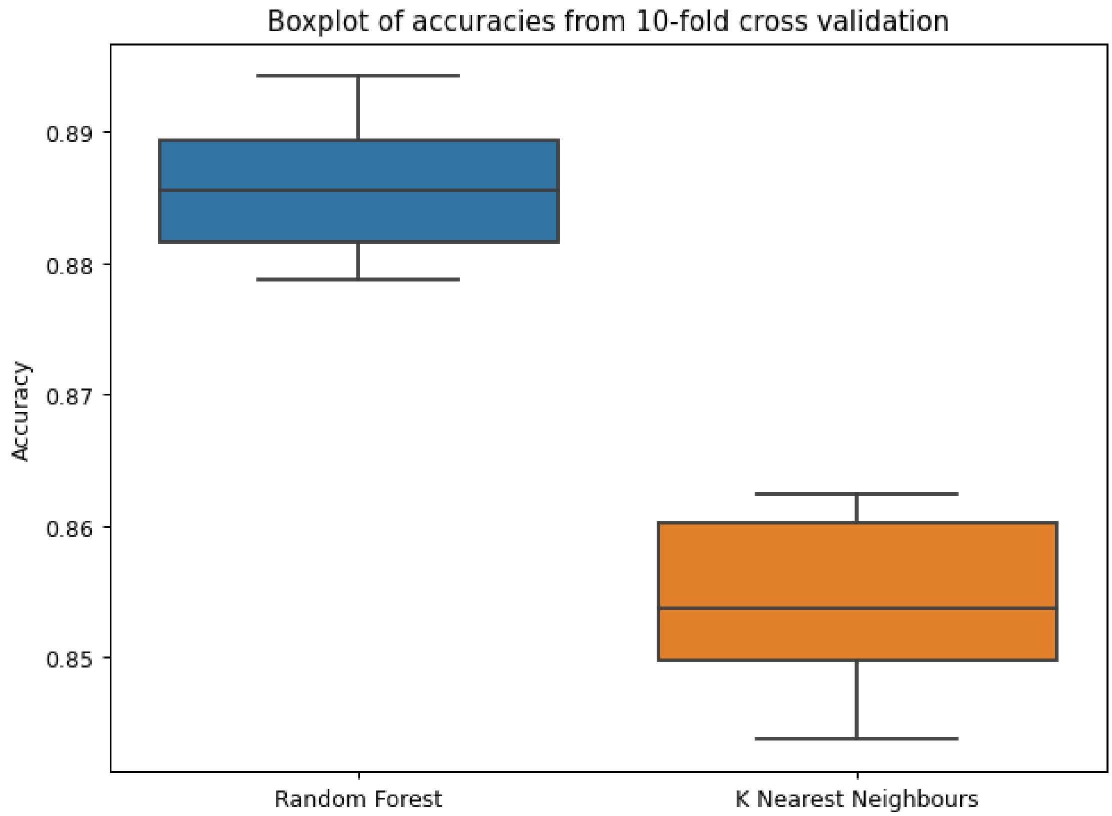
### 1.8 Result Function

[ ]:
```
def result(actual, predicted):

    # confusion matrix
    y_Predicted = np.reshape(predicted, (predicted.shape[0], 1))
    y_Actual = np.reshape(actual, (actual.shape[0], 1))
    df_cm = pd.DataFrame(np.hstack((y_Predicted, y_Actual)),␣
 ↪columns=['y_Actual','y_Predicted'])
    confusion_matrix = pd.crosstab(df_cm['y_Actual'], df_cm['y_Predicted'],␣
 ↪rownames=['Actual'], colnames=['Predicted'])
    print("Confusion Matrix: \n",confusion_matrix)

    # classification report for precision, recall f1-score and accuracy
    matrix = classification_report(actual,predicted, zero_division = 0)
    print('Classification report : \n',matrix)

    plt.figure(figsize = (15,15))
    ax = plt.subplot()
    sns.heatmap(confusion_matrix, ax=ax)
    ax.set_ylabel("True Labels")
    ax.set_xlabel("Predicted Labels")
    ax.set_title("Confusion Matrix for Random Forest Classifier")
    plt.show()
```

### 1.9 Comparing the Random Forest and KNN

[20]:
```
fig = plt.subplots(figsize=(8, 6), dpi=90)



sns.boxplot(data = [cross_val_scores_rf,cross_val_scores_knn])
plt.xticks(plt.xticks()[0],["Random Forest","K Nearest Neighbours"])
```

```
plt.ylabel("Accuracy")
plt.title("Boxplot of accuracies from 10-fold cross validation")
plt.show()
```




## 1.10 Appendix

### 1.10.1 XGBoost

#### Model with default parameters

[ ]:
```
# # First XGBoost model
# # fit model to training data

# model = XGBClassifier()
# model.fit(X_train, y_train)
# # make predictions for test data
# y_pred = model.predict(X_val)
# accuracy = accuracy_score(y_val, y_pred)
# print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

#### Hyper Parameter Tuning

[ ]:

```
# ##### Tuning Max depth #####
# max_depth = range(3,10,1)
# max_depth_accuracies = []
# for mxd in max_depth:
#     xgboost_model = XGBClassifier(max_depth=mxd,n_jobs = -1)
#     xgboost_model.fit(X_train,y_train)
#     max_depth_accuracies.append(xgboost_model.score(X_val,y_val))

# max_index = max_depth_accuracies.index(max(max_depth_accuracies))
# best_maxdepth = max_depth[max_index]
# print("Optimal max depth: {}.".format(best_maxdepth))

# ##### Plot Accuracies for different max-depth #####
# plt.plot(max_depth,max_depth_accuracies)
# plt.title("Accuracies for different max depth")
# plt.xlabel("max_depth")
# plt.ylabel("Accuracies")
# plt.show()


# ##### Tuning min_child_weight #####
# min_child_weight = range(1,6,1)
# min_child_weight_accuracies = []
# for mcw in min_child_weight:
#     xgboost_model = XGBClassifier(max_depth=best_maxdepth,
#                                   min_child_weight = mcw,
#                                   n_jobs = -1)
#     xgboost_model.fit(X_train,y_train)
#     min_child_weight_accuracies.append(xgboost_model.score(X_val,y_val))
# max_index = min_child_weight_accuracies.
↪index(max(min_child_weight_accuracies))
# best_min_child_weight = min_child_weight[max_index]
# print("Optimal min_child_weight: {}.".format(best_min_child_weight))

# ##### Plot Accuracies for different min child weight #####
# plt.plot(min_child_weight,min_child_weight_accuracies)
# plt.title("Accuracies for different min child weight")
# plt.xlabel("min child weight")
# plt.ylabel("Accuracies")
# plt.show()

# ##### Tuning learning rate #####
# learning_rate = [0.0001, 0.001, 0.01, 0.1, 0.2, 0.3]
# learning_rate_accuracies = []
# for lr in learning_rate:
#     xgboost_model = XGBClassifier(max_depth=best_maxdepth,
#                                   min_child_weight = best_min_child_weight,
```

```
#                                    learning_rate = lr,
#                                    n_jobs = -1)
#     xgboost_model.fit(X_train,y_train)
#     learning_rate_accuracies.append(xgboost_model.score(X_val,y_val))
# max_index = learning_rate_accuracies.index(max(learning_rate_accuracies))
# best_learning_rate = learning_rate[max_index]
# print("Optimal Learning rate: {}.".format(best_learning_rate))

# ##### Plot Accuracies for different Learning Rate #####
# plt.plot(learning_rate,learning_rate_accuracies)
# plt.title("Accuracies for different Learning Rate")
# plt.xlabel("Learning Rate")
# plt.ylabel("Accuracies")
# plt.show()

# ##### Tuning n_estimators #####
# n_estimators = range(5,1000,100)
# n_estimators_accuracies = []
# for nest in n_estimators:
#     xgboost_model = XGBClassifier(max_depth=best_maxdepth,
#                                    min_child_weight = best_min_child_weight,
#                                    learning_rate = best_learning_rate,
#                                    n_estimators = nest,
#                                    n_jobs = -1)
#     xgboost_model.fit(X_train,y_train)
#     n_estimators_accuracies.append(xgboost_model.score(X_val,y_val))
# max_index = n_estimators_accuracies.index(max(n_estimators_accuracies))
# best_n_estimators = n_estimators[max_index]
# print("Optimal number of estimator: {}.".format(best_n_estimators))

# ##### Plot Accuracies for different number of estimators #####
# plt.plot(n_estimators,n_estimators_accuracies)
# plt.title("Accuracies for different number of estimators")
# plt.xlabel("Number of Estimators")
# plt.ylabel("Accuracies")
# plt.show()

# ##### Tuning Gamma parameter ####
# gamma = [i/10.0 for i in range(0,5)]
# gamma_accuracies = []
# for gm in gamma:
#     xgboost_model = XGBClassifier(max_depth=best_maxdepth,
#                                    min_child_weight = best_min_child_weight,
#                                    learning_rate = best_learning_rate,
#                                    n_estimators = best_n_estimators,
#                                    gamma = gm,
#                                    n_jobs = -1)
```

```
#     xgboost_model.fit(X_train,y_train)
#     gamma_accuracies.append(xgboost_model.score(X_val,y_val))
# max_index = gamma_accuracies.index(max(gamma_accuracies))
# best_gamma = gamma[max_index]
# print("Optimal gamma: {}.".format(best_gamma))

# ##### Plot Accuracies for different gamma #####
# plt.plot(gamma,gamma_accuracies)
# plt.title("Accuracies for different gamma")
# plt.xlabel("Gamma")
# plt.ylabel("Accuracies")
# plt.show()

# ##### Tuning subsample #####
# subsample = [i/10.0 for i in range(6,10)]
# subsample_accuracies = []
# for sm in subsample:
#     xgboost_model = XGBClassifier(max_depth=best_maxdepth,
#                                   min_child_weight = best_min_child_weight,
#                                   learning_rate = best_learning_rate,
#                                   n_estimators = best_n_estimators,
#                                   gamma = best_gamma,
#                                   subsample = sm,
#                                   n_jobs = -1)
#     xgboost_model.fit(X_train,y_train)
#     subsample_accuracies.append(xgboost_model.score(X_val,y_val))
# max_index = subsample_accuracies.index(max(subsample_accuracies))
# best_subsample = subsample[max_index]
# print("Optimal Subsample: {}.".format(best_subsample))

# ##### Plot Accuracies for different Subsample #####
# plt.plot(subsample,subsample_accuracies)
# plt.title("Accuracies for different Subsample")
# plt.xlabel("Subsample")
# plt.ylabel("Accuracies")
# plt.show()

# ##### Tuning colsample_bytree #####
# colsample_bytree = [i/10.0 for i in range(6,10)]
# colsample_bytree_accuracies = []
# for cst in colsample_bytree:
#     xgboost_model = XGBClassifier(max_depth=best_maxdepth,
#                                   min_child_weight = best_min_child_weight,
#                                   learning_rate = best_learning_rate,
#                                   n_estimators = best_n_estimators,
#                                   gamma = best_gamma,
#                                   subsample = best_subsample,
```

```
#                                   colsample_bytree = cst,
#                                   n_jobs = -1)
#     xgboost_model.fit(X_train,y_train)
#     colsample_bytree_accuracies.append(xgboost_model.score(X_val,y_val))
# max_index = colsample_bytree_accuracies.
 ↪index(max(colsample_bytree_accuracies))
# best_colsample_bytree = colsample_bytree[max_index]
# print("Optimal colsample_bytree: {}.".format(best_colsample_bytree))

# #####Plot Accuracies for different colsample_bytree #####
# plt.plot(colsample_bytree,colsample_bytree_accuracies)
# plt.title("Accuracies for different colsample_bytree")
# plt.xlabel("colsample_bytree")
# plt.ylabel("Accuracies")
# plt.show()

# ##### Tuning Regularization Parameters #####
# reg_alpha = [1e-5, 1e-2, 0.1, 1, 100]
# reg_alpha_accuracies = []
# for reg in reg_alpha:
#     xgboost_model = XGBClassifier(max_depth=best_maxdepth,
#                                   min_child_weight = best_min_child_weight,
#                                   learning_rate = best_learning_rate,
#                                   n_estimators = best_n_estimators,
#                                   gamma = best_gamma,
#                                   subsample = best_subsample,
#                                   colsample_bytree = best_colsample_bytree,
#                                   reg_alpha = reg,
#                                   n_jobs = -1)
#     xgboost_model.fit(X_train,y_train)
#     reg_alpha_accuracies.append(xgboost_model.score(X_val,y_val))
# max_index = reg_alpha_accuracies.index(max(reg_alpha_accuracies))
# best_reg_alpha = reg_alpha[max_index]
# print("Optimal Regularization parameter: {}.".format(best_reg_alpha))

# #####Plot Accuracies for different colsample_bytree #####
# plt.plot(reg_alpha,reg_alpha_accuracies)
# plt.title("Accuracies for different colsample_bytree")
# plt.xlabel("Regularization parameter")
# plt.ylabel("Accuracies")
# plt.show()
```

**Using optimal parameters to make predictions on test dataset**

[ ]:
```
# xgboost_final_model = XGBClassifier(max_depth=best_maxdepth,
#                                     min_child_weight = best_min_child_weight,
#                                     learning_rate = best_learning_rate,
```

```
#                                     n_estimators = best_n_estimators,
#                                     gamma = best_gamma,
#                                     subsample = best_subsample,
#                                     colsample_bytree = best_colsample_bytree,
#                                     reg_alpha = best_reg_alpha,
#                                     n_jobs = -1)
# xgboost_final_model.fit(X_train,y_train)

# y_pred_final = xgboost_final_model.predict(X_test)
# accuracy_final = accuracy_score(y_test, y_pred)
# print("Accuracy: %.2f%%" % (accuracy_final * 100.0))
```

#### 1.10.2 Gradient Boost

**Gradient Boost model with default parameters**

[ ]:
```
# gb_model = GradientBoostingClassifier(n_estimators = 15, random_state=1)
# gb_model.fit(X_train, y_train)
# # make predictions for validation data
# y_pred_gb = gb_model.predict(X_val)
# accuracy_1 = accuracy_score(y_val, y_pred_gb)
# print("Accuracy: %.2f%%" % (accuracy_1 * 100.0))
```

**Tuning Gradient model parameters**

[ ]:
```
# parameters = [{'n_estimators': [50, 100, 250, 500],
#                'max_depth':[5, 7, 9, 11, 13, 15],
#                'max_features':[0.05, 0.1, 0.2],
#                'subsample':[0.6,0.7,0.75,0.8,0.85,0.9]}]


# gbm_mod = GridSearchCV(GradientBoostingClassifier(verbose = 0, random_state =
↪1),
#          parameters, scoring='accuracy' )
# gbm_mod.fit(X_train, y_train)

# print("Detailed classification report:")
# y_true, y_pred = y_test, gbm_mod.predict(X_val)
# print(classification_report(y_true, y_pred))
```

**Using optimal parameters to make predictions on test dataset**

[ ]:
```
# gb_final_model = GradientBoostingClassifier(gbm_mod.best_params_,
↪random_state=1)
# gb_final_model.fit(X_train, y_train)

# # make predictions for validation data
# y_final_pred = gb_final_model.predict(X_test)
```

```python
# accuracy_final = accuracy_score(y_test, y_final_pred)
# print("Accuracy: %.2f%%" % (accuracy_final * 100.0))
```

#### 1.10.3 Adaptive Boosting

[ ]:
```python
# Initialize AdaBoost Classifier
# clf = AdaBoostClassifier(n_estimators=100, random_state=0)
# clf.fit(X_train, y_train)
# clf.score(X_test, y_test)
```

[ ]:
```python

```