Mr.

Mark Mburu Njoroge

Submitted for the Degree of Master of Science in

Machine Learning

Department of Computer Science
Royal Holloway University of London
Egham, Surrey TW20 0EX, UK

June 16, 2020

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 10000

Student Name: Mark Mburu Njoroge

Date of Submission: 20-July-20

Signature:

Abstract

This project is an audio feature extraction and classification with the ECS-10 data set and ESC-50 dataset audio dataset .ECS-10 audio data is included. It consists of 10 classes of different environmental sounds (person sneezing, person coughing, etc.).The main objective of this project is to compare classification accuracies for the 6 tested classifiers. The dependencies we are using for this projects are Librosa (audio loading, audio visualization and feature extraction),Sci-kit learn, Keras (Theano backend),Numpy, Matplotlib and Pandas (data visualization). The scripts for feature extraction and classification have been added as .ipynb files and are all loaded in the google colab. Running feature_extraction.py creates a numpy array for features (feature.npy) and one for labels (label.npy). These files will be saved in the content directory.The audio extracted from the dataset are MFCC, Chroma, Mel spectrogram, Tonal centroid feature and Spectral contrast. The classifiers implemented are:

- Support Vector Machine (SVM)
- K Nearest neighbors(KNN)
- Random Forest (RF)
- Naive Bayes (NB)
- Convolutional Neural Network (CNN)
- Multilayer Perceptron (MLP)
- Recurrent Neural Network (RNN)

Direct comparison between classifiers can't be done yet since their parameters haven't been tuned to optimize accuracy yet. Out of 400 audio samples, the test set consisted on the 33% of this.

- SVM: 81.7%
- RF: 83.33%
- NB: 68.94%
- KNN:61.88%
- CNN: 78.125% (100 epochs)
- MLP: 79.3%(100 epochs)
- RNN: 72.00%(100 epochs)

To improve the accuracy obtained we can compute other features: MFCC + ZCR features improve classification accuracy for speech, noise and music labels to see if it also works for the 10 classes, tune optimization hyper parameters (for every classifier): Weight initialization, decaying learning rate and data scaling and feature normalization (MFCC).

Data Preparation:

In order to work implement the classification algorithms to make predictions I needed to clean and transform raw data to enable me to process and analysis it. This process of cleaning and transforming data is known as data preparation.The code below illustrate how I transformed the dataset to get the features and labels as numpy arrays file that I later used to train and predict on my models.

```
Import glob
import os
  import librosa
  import numpy as np
  import matplotlib.pyplot as plt
  from matplotlib.pyplot import specgram
  import soundfile as sf


  def feature_extraction(file_name):
     X, sample_rate = librosa.load(file_name)
     if X.ndim > 1:
        X = X[:, 0]
     X = X.T


     # Get features
     stft = np.abs(librosa.stft(X))
     mfccs = np.mean(librosa.feature.mfcc(y=X, sr=sample_rate, n_mfcc=40).T, axis=0)  # 40 values
     # zcr = np.mean(librosa.feature.zero_crossing_rate)
     chroma = np.mean(librosa.feature.chroma_stft(S=stft, sr=sample_rate).T, axis=0)
     mel = np.mean(librosa.feature.melspectrogram(X, sr=sample_rate).T, axis=0)
     contrast = np.mean(librosa.feature.spectral_contrast(S=stft, sr=sample_rate).T, axis=0)
     tonnetz = np.mean(librosa.feature.tonnetz(y=librosa.effects.harmonic(X), sr=sample_rate).T, # tonal centroid features
                       axis=0)


   # Return computed features
     return mfccs, chroma, mel, contrast, tonnetz
```

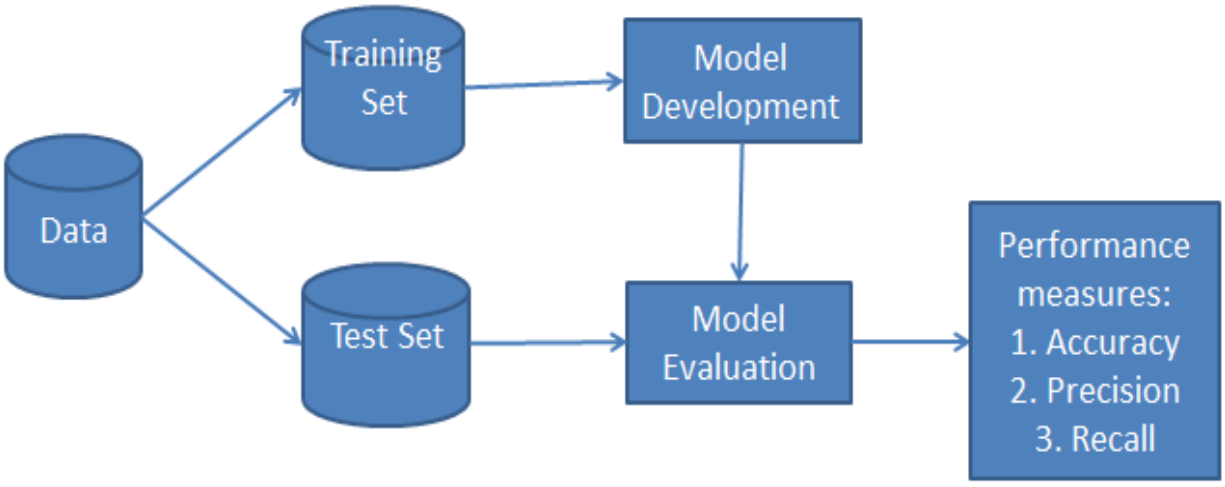
```
# Process audio files: Return arrays with features and labels
def parse_audio_files(parent_dir, sub_dirs, file_ext='*.ogg'):  ## .ogg audio format
    features, labels = np.empty((0, 193)), np.empty(0)  # 193 features total. This can vary


    for label, sub_dir in enumerate(sub_dirs):  ##Enumerate() function adds a counter to an iterable.
        for file_name in glob.glob(os.path.join(parent_dir, sub_dir, file_ext)): ##parent is audio-data, sub_dirs are audio classes
            try:
                mfccs, chroma, mel, contrast, tonnetz = feature_extraction(file_name)
            except Exception as e:
                print("[Error] there was an error in feature extraction. %s" % (e))
                continue


            extracted_features = np.hstack(
                [mfccs, chroma, mel, contrast, tonnetz])  # Stack arrays in sequence horizontally (column wise)
            features = np.vstack([features, extracted_features])  # Stack arrays in sequence vertically (row wise).
            labels = np.append(labels, label)
        print("Extracted features from %s, done" % (sub_dir))
    return np.array(features), np.array(labels, dtype=np.int)


# Read sub-directories (audio classes)
audio_directories = os.listdir("audio-data/")
audio_directories.sort()


# Function call to get labels and features
# This sabes a feat.npy and label.npy numpy-files in the current directory
features, labels = parse_audio_files('audio-data', audio_directories)
np.save('feat.npy', features)
np.save('label.npy', labels)
```

The code above shows how I transformed my dataset which was a folder containing other folders with audio files. I first import the necessary modules that I needed which were glob which helped in finding the pathnames of the dataset, os which is a module that provides of using

dependent functionality, librosa which I used for audio analysis since the dataset had audio filse, soundfile which I used to read audio samples, numpy which we use to work with arrays and matplotlib which we use to visualize our dataset. After importing the dataset I defined a function feature extraction that took one parameter of file name and used librosa to load the audio files that were passed as filenames from our dataset. After loading the file names I got the features and computed them and returned the computed features that were ready to be processed. Then I processed the features to return an array of features and labelled and stored them in feat.npy file and label.npy file which represented the features and labels respectively

1.1.1 Naïve Bayes

Naive Bayes is a fast and is a classification algorithm which is the most straightforward and is good for classifying large datasets. Naive Bayes classifier is mostly used in various applications for classification such as filtering spam emails where it can be used to classify spam emails , text classification where is used to classify product ratings and comments from consumers , sentiment analysis where its used to categorize opinions from pieces of texts , and systems that do recommendation. It uses probability of prediction from Bayes theorem of the unknown class. Whenever we perform classification, the first step is to understand the problem and identify potential features and label. Features or independent variables are those characteristics or attributes which affect the results of the label which are dependent variables. For example, in the case of a loan distribution, bank managers identify customer’s occupation, income, age, location, previous loan history, transaction history, and credit score. These characteristics are known as features which help the model classify and categorize the customers. The classification has two phases, a learning phase and the evaluation phase. The classifier trains its model on a given dataset in the learning phase, and it tests the classifier performance in the evaluation phase. Performance is evaluated on the basis of various parameters such as accuracy, error, precision or positive predictive value which is the fraction of relevance among the retrieved instances, and recall which is also referred to sensitivity is the fraction of the total amount of relevant instances that were.



Naive Bayes classifier is based on a statistical classification technique of Bayes Theorem. It is part of machine learning algorithms under the supervised learning algorithms. Naive Bayes classifier is the fast, accurate and reliable algorithm. Naive Bayes classifiers have high accuracy and speed on large datasets. Naive Bayes classifiers assumes that the effect of a particular feature in a class is independent of other independent variables. For example, a loan applicant is desirable or not depending on his/her income, previous loan and transaction history, age, and location. Even if these features are interdependent, these features are still considered independently. It is considered as naïve because it assumes that features are interdependent which simplifies computation and this is one of the reason its fast on large dataset. This

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

assumption is called class conditional independence.

- P(h): the probability of hypothesis h being true regardless of the data. This is known as the prior probability of h.
- P(D): the probability of the data regardless of the hypothesis. This is known as the prior probability.

Whether	Play
Sunny	No
Sunny	No
Overcast	Yes
Rainy	Yes
Rainy	Yes
Rainy	No
Overcast	Yes
Sunny	No
Sunny	Yes
Rainy	Yes
Sunny	Yes
Overcast	Yes
Overcast	Yes
Rainy	No

Frequency Table

Whether	No	Yes
Overcast		4
Sunny	2	3
Rainy	3	2
Total	5	9

Likelihood Table 1

Whether	No	Yes		
Overcast		4	=4/14	0.29
Sunny	2	3	=5/14	0.36
Rainy	3	2	=5/14	0.36
Total	5	9		
	=5/14	=9/14		
	0.36	0.64		

Likelihood Table 2

Whether	No	Yes	Posterior Probability for No	Posterior Probability for Yes
Overcast		4	0/5=0	4/9=0.44
Sunny	2	3	2/5=0.4	3/9=0.33
Rainy	3	2	3/5=0.6	2/9=0.22
Total	5	9		

- P(h|D): the probability of hypothesis h given the data D. This is known as posterior probability.
- P(D|h): the probability of data d given that the hypothesis h was true. This is known as posterior probability.

Naive Bayes classifier calculates the probability of an event in the following steps we will first calculate the prior probability for given class labels the we will find likelihood probability with each attribute for each class then we put these value in Bayes Formula and calculate posterior probability and lastly we will see which class has a higher probability, given the input belongs to the higher probability class.

We use the two tables frequency and the likelihood tables after simplifying posterior and prior probability calculations. Both of these tables above will help us calculate the prior and posterior probability. The Frequency table contains the occurrence of labels for all features. There are two likelihood tables. Likelihood in table 1 is showing prior probabilities of labels and Likelihood Table 2 is showing the posterior probability.

Now suppose if we want to calculate the probability of playing when the weather is overcast.

Probability of playing:

Prob(yes or overcast) is equal to prob(Overcast or Yes) Prob(Yes) / Prob (Overcast) .....(1)

- To calculate the prior probabilities:
    - Prob(overcast) = 4/14 = 0.29
    - Prob(Yes)= 9/14 = 0.64
- Calculate Posterior Probabilities:
    - Prob(Overcast or Yes) i=4/9 = 0.44
- Putting the prior and posterior probabilities in the first equation (1)
    - Prob (Yes or Overcast) = 0.44 * 0.64 / 0.29 = 0.98(Higher)

The Probability of not playing:

Prob(No or Overcast) = Prob(Overcast or No) Prob(No) / Prob (Overcast) .....(2)

- To calculate the prior probabilities we have:
    - Prob(Overcast) = 4/14 = 0.29
    - Prob(No)= 5/14 = 0.36
- To calculate the Posterior Probabilities:
    - Prob(Overcast or No) = 0/9 = 0
- Putting the posterior and prior probabilities in the second equation(2)
    - Prob (No or Overcast) = 0 * 0.36 / 0.29 = 0

The probability of a 'Yes' class is higher and we can determine if the weather is overcast then players will play the sport. Suppose that we want to calculate the probability of playing when the weather is overcast, and the temperature is mild.

The Probability of playing:

Prob(Play is Yes or Weather is Overcast, Temp is Mild)is equal to Prob(Weather is Overcast, Temp is Mild or Play is Yes)*Prob(Play is Yes) .....(1)*

Prob(Weather is Overcast, Temp is Mild or Play is Yes)is equal to Prob(Overcast or Yes) *Prob(Mild orYes) ……..(2)*

- To calculate the prior probabilities: Prob(Yes) is 9/14 = 0.64
- To calculate the posterior probabilities: Prob(Overcast orYes) is 4/9 = 0.44 Prob(Mild orYes) = 4/9 = 0.44
- Putting the Posterior probabilities in the second equation (2) Prob(Weather is Overcast, Temp is Mild or Playis Yes) is equal to 0.44 * 0.44 = 0.1936(Higher)
- Putting the posterior and prior probabilities in the first equation (1) Prob(Play is Yes or Weather is Overcast, Temp is Mild) is equal to0.1936*0.64 = 0.124

The Probability of not playing:

Prob(Play is No or Weather is Overcast, Temp is Mild) is eaqual to Prob(Weather is Overcast, Temp is Mild or Play is No)*Prob(Play is No) .....(3)*

Prob(Weather is Overcast, Temp is Mild or Play is No) is equal to Prob(Weather is Overcast or Play is No) *P(Temp is Mild* or Play is No) …….(4)

- To calculate the prior probabilities: Prob(No)= 5/14 = 0.36
- To calculate the posterior probabilities: Prob(Weather is Overcast or Play is No) is equal to 0/9 = 0 P(Temp is Mild or Play is No)=2/5=0.4
- Putting the posterior probabilities in the fourth equation (4) Prob(Weather is Overcast, Temp is Mild or Play is No) = 0 * 0.4= 0
- Putting the posterior and prior probabilities in the third equation (3) Prob(Play is No or Weather is Overcast, Temp is Mild) = 0*0.36=0

The probability of a 'Yes' class is higher. So you can say here that if the weather is overcast than players will play the sport.

Classifier Building in Scikit-learn

Naive Bayes Classifier:

To use scikit-learn knn classifier algorithm, I first imported the modules that I needed to use with my dataset which are:

- Numpy to read my data
- Accuracy score from sklearn to calculate the accuracy of my model
- GaussianNB Classifier from sklearn naïve_bayes.
- Train test split to split my data

```
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np


#Load data
X = np.load('/content/feat.npy')
X = np.load('/content/label.npy')


#Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
random_state=42)


# Initialize classifier
gnb_clf= GaussianNB() #check input params


# Train model
gnb_clf.fit(X_train, y_train)


# To the Make predictions after training and fitting the model
prediction = gnb_clf.predict(X_test)


print('Predicted values')
#print the prediction output
print(prediction)
print('Actual values')
# this is to print the testing labels
```

```
print(y_test)


# Evaluate accuracy
acc = gnb_clf.score(X_test, y_test)
#print the accuracy of the model
print("Accuracy = %0.3f" %acc)
```

The above code is how I implemented the Naïve bayes classifier on the dataset. First I imported the necessary modules then I loaded the data and stored them and X and y variables where x stored the features or the independent variables and y stored the labels or the dependent variable .Then I spitted the data into training set and testing set using train test split method from sklearn . Then I initialized the classifier which is GaussianNB and I assigned it to gnb_clf variable. Then I trained my data by using the fit method and I passed two parameters or arguments into it which are the training set. Then I did my predictions by passing the testing set into the predict method of gnb_clf variable. Then I printed out the prediction and I calculated the accuracy of my model which was 0.6894 or 68.94%.

1.2 Advantages of Using Naïve Bayes

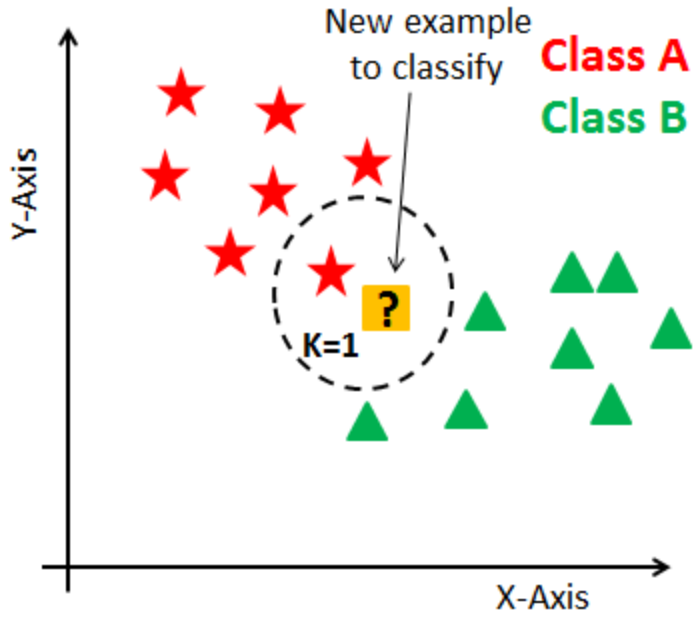
- It is not only a simple approach but also a fast and accurate method for prediction.
- Naive Bayes has very low computation cost.
- It can efficiently work on a large dataset.
- It performs well in case of discrete response variable compared to the continuous variable.
- It can be used with multiple class prediction problems.
- It also performs well in the case of text analytics problems.
- When the assumption of independence holds, a Naive Bayes classifier performs better compared to other models like logistic regression.

1.3 Disadvantages of Naive Bayes

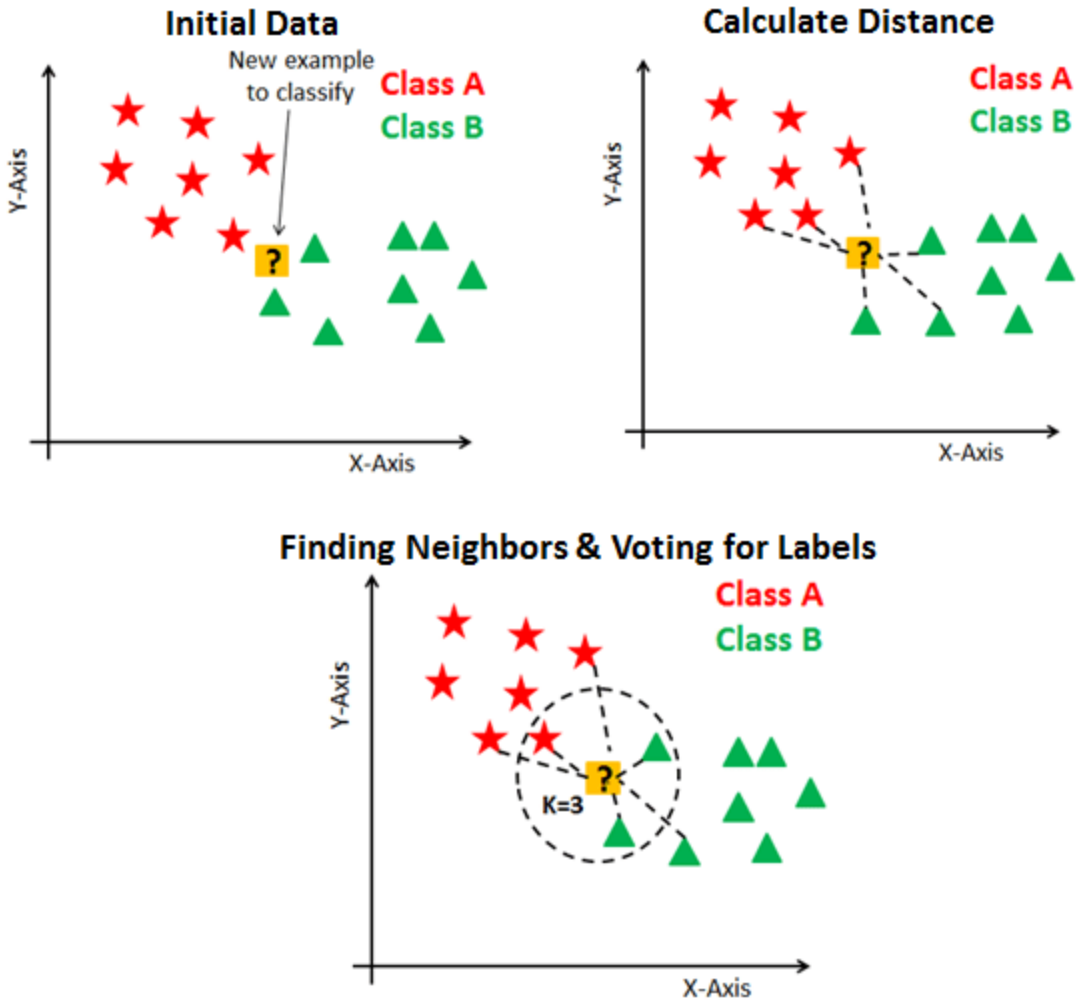
- The assumption of independent features. In practice, it is almost impossible that model will get a set of predictors which are entirely independent.
- If there is no training tuple of a particular class, this causes zero posterior probability. In this case, the model is unable to make predictions. This problem is known as Zero Probability/Frequency Problem.

1.3.1 KNN Classification Algorithm

K Nearest Neighbor (KNN) is used in many different applications such as finance and banking, healthcare, political science to classifying potential voters in two classes will vote or won't vote, detection of handwriting, image and video recognition. Financial institutes make prediction on the credit rating of customers. It's used by banking institutes to predict whether the loan is safe or risky for loan disbursement. KNN algorithm can be used for both classification as well as regression problems and they are based on feature similarity approach. KNN is a lazy learning algorithm as well as non-parametric. Non-parametric is where the underlying data distribution is not assumed which means that the model structure is determined from the dataset. The real world dataset does not follow the mathematical and theoretical assumption hence its very helpful if the model structure is determined from the dataset. Lazy learning algorithm is case where the algorithm does not need any training data points for model generation which means that all the training data is used in the testing phase. This makes testing phase slower and costlier and training faster where costly testing phase means it takes a lot of time and memory. In some cases the KNN takes a lot of time to scan all data points and scanning all data points this makes it to use more memory for storing training data. The number of nearest neighbors is represented by k and it is mostly the core deciding factor. If the number of classes is two then K is generally an odd number. It is referred as the nearest neighbor algorithm if the k is equal to one and this is its simplest case. Suppose the point for which the label needs to predict is P1. First, you will calculate the closest point to P1 and then the label assigned to the nearest point of it.

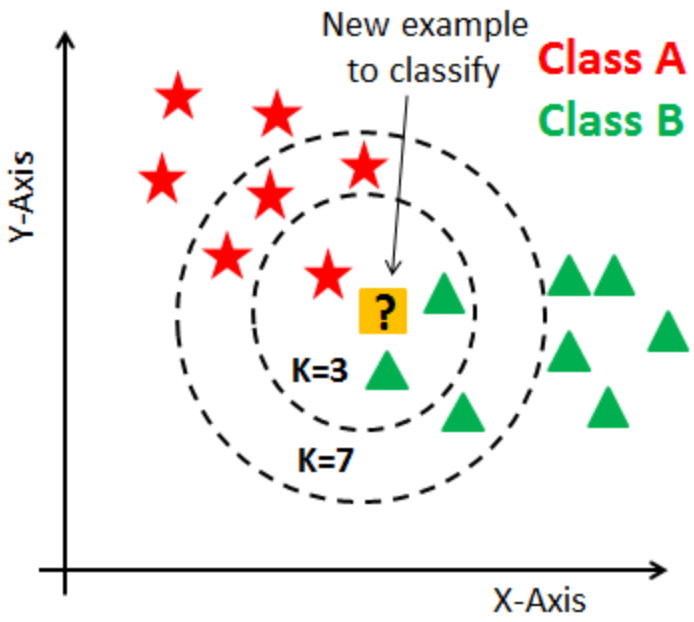


Suppose the point for which the label needs to predict is P1 then we first find the k closest point to p1 and then the majority vote of k neighbors are used to classify the points. The prediction is taken from the class with the most votes and each object votes for their class. To find the closest similar points we find the distance between points using distance measures method such as Manhattan distance, Euclidean distance, Minkowski distance and Hamming distance. The following steps are followed by KNN such as calculate the distance ,find the closest neighbors and vote for the labels



Eager learners are learners that are ready, active and eager to classify unobserved data points and when given the training points will constructs a generalized model before performing prediction on new points to classify. Lazy learning classify the data points at the last minute as well as the model does not need to train or learn and all of the data points used at the time of prediction. For Lazy learner the training dataset is merely stored and waits until classification needs to perform. It performs if the dataset has a tuple. It perform generalization to classify the tuple based on its similarity to the stored training tuples. Lazy learners do less work in the training phase and more work in the testing phase to make a classification unlike eager training or learning methods. They

store the training points or instances, and all learning is based on instances hence they are known as instance-based learners. KNN performs well with fewer number of features than a large number of features. The problem of overfitting is caused by the Increase in dimensionality. The data needed will need to grow exponentially as we increase the number of dimensions to avoid overfitting. Curse of Dimensionality is the problem of higher dimension.To deal with t curse of dimensionality, before applying any machine learning algorithm we need to perform principal component analysis, or we can use feature selection approach. We can prefer other measures such as cosine similarity since research has shown that in large dimension Euclidean distance is not useful anymore, which get less affected by high dimension. Now that we understand the working mechanism. of KNN algorithm , we have to come up with a way to choose the optimal number of neighbors and understand the effects of it to the classifier. The number of neighbors (K) in KNN is a hyper parameter that we need to choose at the time of model building and it is a controlling variable for the prediction model .No optimal number of k neighbors suits all kind of data sets which means that each dataset has its own unique requirements. The noise have a higher influence on the result In the case of a small number of neighbors and it is computationally expensive for large number of neighbors. A large number of neighbors will have a smoother decision boundary which means lower variance but higher bias and small amount of neighbors are most flexible to fit which will have low bias but high variance. We choose an odd number if the number of classes is even and check by generating the model on different values of k and check their performance. We can also use Elbow method to perform this process.



Building Classifier in Scikit-learn

KNN Classifier:

To use scikit-learn knn classifier algorithm, I first imported the modules that I needed to use with my dataset which are:

- Numpy to read my data
- Accuracy score from sklearn to calculate the accuracy of my model
- K Neighbours Classifier from sklearn neighbours which is the algorithm
- Train test split to split my data

```
#Importing the libraries
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
import numpy as np
from sklearn.metrics import accuracy_score

#Load the dataset
X =  np.load('/content/feat.npy')
y =  np.load('/content/label.npy').ravel()

#split the dataset into training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=0)

#Initialize the classifier
model = KNeighborsClassifier(n_neighbors=3)

# Train the model using the training sets
```

```
model.fit(X_train,y_train)

#predict the outcome
prediction = model.predict(X_test)

#Check the accuracy
acc = model.score(X_test, y_test)

#print the accuracy
print("Accuracy = %0.4f" %acc)
```

The above code is how I implemented the K Nearest Neighbors classifier on the dataset. First I imported the necessary modules then I loaded the data and stored them and X and y variables where x stored the features or the independent variables and y stored the labels or the dependent variable .Then I split the data into training set and testing set using train test split method from sklearn . Then I initialized the classifier which is K Neighbors Classifier with three neighbors and I assigned it to model variable. Then I trained my data by using the fit method and I passed two parameters or arguments into it which are the training set. Then I did my predictions by passing the testing set into the predict method of model variable. Then I printed out the prediction and I calculated the accuracy of my model which was 0.6188 or 61.88%.

Advantages of K-Nearest Neighbor (KNN)

Compared to other classification algorithms K-nearest neighbor classification is much faster in the training phase. It is known as the simple and instance-based learning algorithm thus there is no need to train a model for generalization and can be in case of nonlinear data it is very useful. The average of K closest neighbors value computes the output value for the object and can be used to solve regression problem,

1.4 Disadvantages of K-Nearest Neighbor (KNN)

It is slower, costlier in terms of time and memory for the testing phase of the K-nearest neighbor classification. In order to store and predict the entire training dataset the algorithm requires large memory. It uses the Euclidean distance between two data points to find nearest neighbors and to scale the data. Euclidean distance is sensitive to magnitudes. It’s not suitable for large dataset and the features with low magnitude weigh less than those with high magnitude

1.5 How to improve KNN

It is highly recommended to normalize data on the same scale for better result. The normalization range is considered to be between 0 and 1. KNN is not suitable for the large dimensional data. In cases of high dimensions, to improve the performance we need to reduce the dimension. Fixing missing values in the dataset helps us in improving results.

1.5.1 Random Forest

The decision trees are combined to make a random forest .A large number of individual decision trees that operate as an ensemble belongs to a random forest. Our model predictions are made up of the class with the most votes and a class prediction is spit from each individual tree in a random forest. The wisdom of crowds is the fundamental concept behind the random forest. The reason that the random forest model works so well is because a large number of uncorrelated relatively trees or models operating as a committee outperforms the other individual constituent models. The key thing is low correlation between models. This is how investments with low correlations like stocks and bonds come together and form a portfolio that is greater than the sum of its parts and uncorrelated models produce ensemble predictions that are more accurate than any of the individual predictions. As long as tress do not have constant all error in the same direction they protect each other from individual error. While many other trees will be right, some trees may be wrong so as a group the trees that are right are able to move in the correct direction. The prerequisites for random forest to perform well are such as the need to have some actual signal in

our features so that models that are built using those features do better than random guessing and the predictions that are made by the individual trees should have low correlations with each other. The wonderful effects of having many uncorrelated models is such a critical concept that I want to show you an example to help it really sink in. Imagine that we are playing the game which to produce a number I use a uniformly distributed random number generator. You win the game if I generate a number that is greater than or equal to 40, so you have victory chance of 60% and I pay you some money. I win If it the number is below 40 and you would have to pay me the same amount of money. Now I give you the some choices to follow. We can either: play 100 times, betting $1 each time or play 10 times, betting $10 each time or play one time, betting $100.Note that the expected value of each game is the same which are (0.60*1 + 0.40*-1)*100 = 20 or (0.60*10 + 0.40*-10)*10 = 20 or 0.60*100 + 0.40*-100 = 20.If we visualize the results with a Monte Carlo simulation we will run 10,000 simulations of each game type; i.e. we will simulate 10,000 times the 100 plays of the first choice. The outcome distributions are vastly different going from positive and narrow to binary even though the expected values are the same**.** The first choice where we play 100 times offers up the best chance of making some money because you make money in 97% out of the 10,000 simulations that I ran. For the second choice where we play 10 times we get a drastic decline because you make money in 63% of the simulations and a drastic increase in your probability of losing money. For the third choice where we only play once you only make money in 60% of the simulations as expected which we conclude that the outcome distribution are completely different even though the games share the same expected value. We will make money if we split up our $100 bet into different play. This works because each play is independent of the other ones. If each of the tree is like one play in our choices the random forest is the same. Chances of making correct prediction increases with the number of uncorrelated tress in our random forest model which is similar to how our chances of making money increased the more time we played. The random forest uses two methods to ensure that the behavior of each individual tree is not too correlated with the behavior of any other trees in the model. These methods are Bagging or bootstrap aggregation where small changes to the training set can cause

an outcome that is significantly different tree structures because decision trees are very sensitive to the data they are trained on. So to get result in different trees Random forest allows each individual tree to randomly sample from the dataset with replacement while taking advantage of decision trees being sensitive to the data they are trained on and this process is known as bagging. Notice that with bagging we are not sub setting the training data into smaller chunks and training each tree on a different chunk. Rather, if we have a sample of size S, we are still feeding each tree a training set of size S. But instead of the original training data, we take a random sample of size S with replacement i.e. we might give one of our trees the following list [1, 2, 2, 3, 6, 6] if our training data was [1, 2, 3, 4, 5, 6]. Note we sample with replacement that is why both lists are of length six and that "2" and "6" are both repeated in the randomly selected training data we give to our tree. Feature Randomness is when splitting the node in a normal decision tree every possible features is considered and we pick the one that produces the most separation between the observations in the left and for those in the right node. In a random forest each tree can only pick from a random subset of features. This ultimately results in lower correlation across trees and more diversification because it forces even more variation amongst the trees in the model .For random forests when building a bunch of uncorrelated models and each with a positive expected return we then put them together in a portfolio to earn massive alpha. The random forest is a classification algorithm consisting of many decisions trees and it uses feature randomness and bagging when building each individual tree to try to create an uncorrelated forest of trees which predict more accurately than that of any individual decision tree. In order for our random forest. In order to make these accurate class predictions We need the features to have at least some predictive power so that if we put garbage in then we will get garbage out and the trees of the forest and more importantly their predictions should be uncorrelated or should at least have few/low correlations with each other. While the algorithm itself tries to engineer these low correlations for us via feature randomness, the features we select and the hyper-parameters we choose may as well impact the ultimate correlations.

Implementing Random Forest:

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier #Random Forest classifier
from sklearn.metrics import accuracy_score
import pandas as pd
import numpy as np
# Initialize classifier
rf_clf = RandomForestClassifier(n_jobs=2, random_state=0)

# Train model
rf_clf.fit(X_train, y_train)

# Make predictions
y_prediction = rf_clf.predict(X_test)
#getting the accuracy
acc = rf_clf.score(X_test, y_test)
#printing the accuracy
print("Accuracy = %0.4f" %acc)
rf_clf.predict_proba(X_test)[0:10]

#label decoding
label_classes = np.array(['Dog bark','Rain','Sea waves','Baby cry','Clock tick','Person snee
ze','Helicopter','Chainsaw','Rooster',
                          'Fire crackling'])
# Decoding predicted and actual classes (numeric to written)
prediction_decoded = label_classes[y_prediction]
actual_value_decoded = label_classes[y_test]
```

The above code is how I implemented the random forest classifier on the dataset. First I imported the necessary modules then I loaded the data and stored them and X and y variables where x stored the features or the independent variables and y stored the labels or the dependent variable .Then I split the data into training set and testing set using train test split method from sklearn . Then I initialized the classifier which is Random forest classifier with two arguments of random state and jobs and I assigned it to rf_clf variable. Then I trained my data by using the fit method and I passed two parameters or arguments into it which are the training set. Then I did my predictions by passing the testing set into the predict method of rf_clf variable. Then I printed out the prediction and I calculated the accuracy of my model which was 0.8333 or 83.33%.This was the highest accuracy I got from all classifiers that I implemented.

Advantages of Random Forest

- They are considered as a highly accurate and robust method because of the number of decision trees participating in the process.

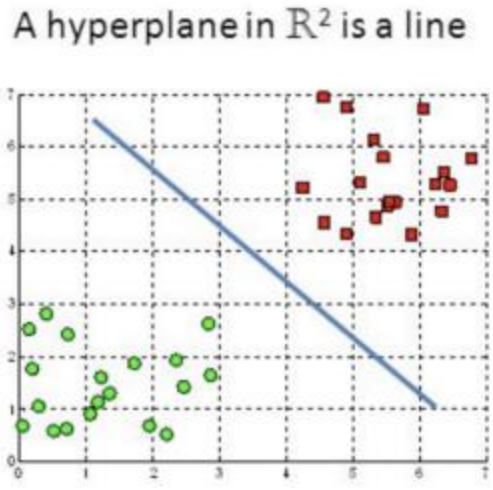
- They do not suffer from the overfitting problem. The main reason is that it takes the average of all the predictions, which cancels out the biases.
- Random forest can be used in both classification and regression problems.
- They can handle missing values by either using median values to replace continuous variables and computing the proximity-weighted average of missing values or we can get the relative feature importance which helps in selecting the most contributing features for the classifier.

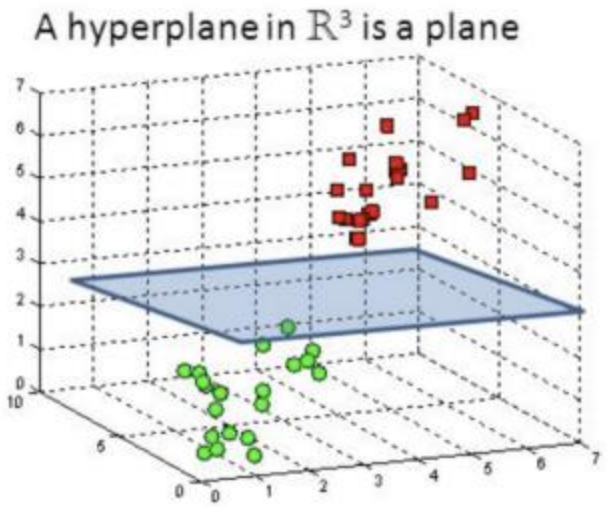
Disadvantages of Random forest Classifier

- Because of it having multiple decision trees it is slow in generating predictions. For it to make a prediction, all the trees in the forest have to make a prediction for the same given input and then perform voting on it which is time consuming.
- It is difficult to interpret a random forest model compared to a decision tree, where you can easily make a decision by following the path in the tree.

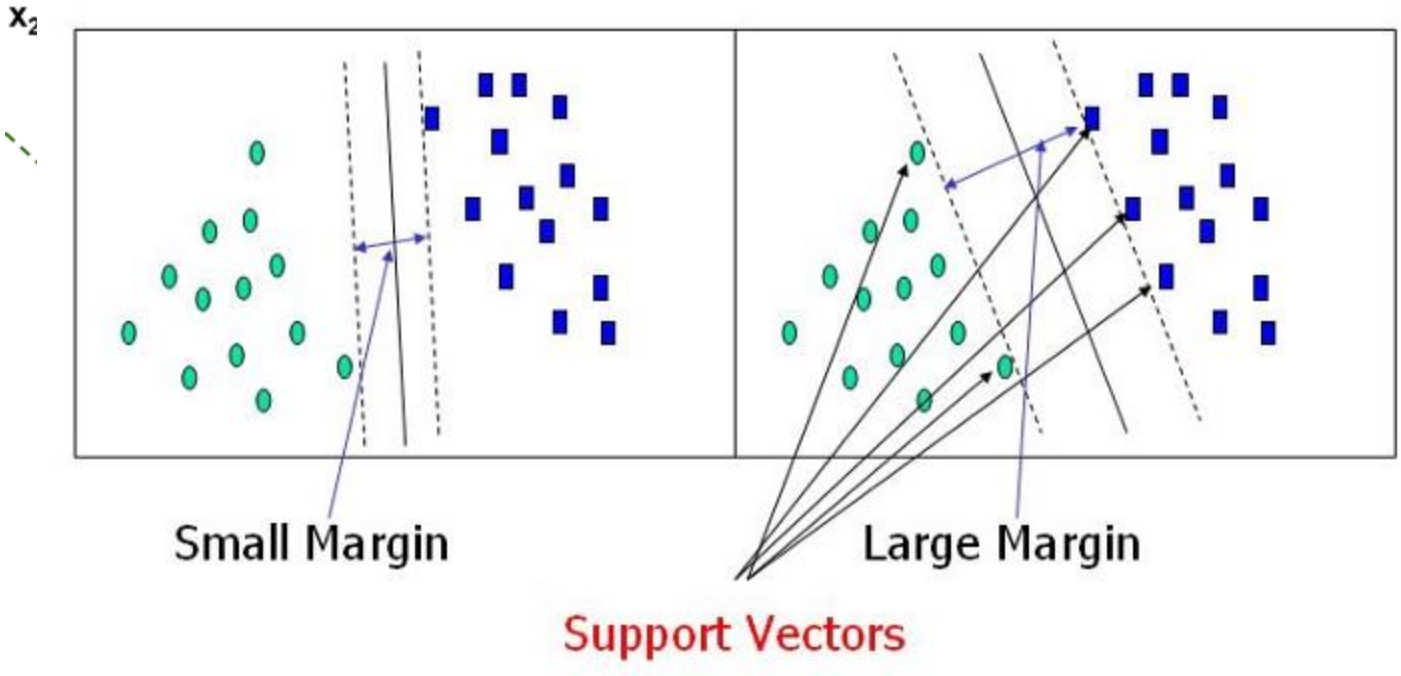
1.5.2 Support Vector Machine

Support Vector Machine finds the decision boundary that separate the different classes and maximize the margins. Margins are those dots closet to the line and the perpendicular distances between the lines. Support Vector Machine also finds the optimal line with the constraint of correctly classifying either class by only looking into the **separate hyperplanes** because the hyperplanes classify classes correctly and pick up the one that maximizes the **margin to** conduct optimization. We will use the concept of margin and separate hyperplane. For an n-dimensional space. For a 2-dimension space, hyperplane is an n minus 1-dimensional subspace and a hyperplane will always be 1-dimension in which it is just a line. Hyperplane will be 2-dimension, for a 3-dimension space which is a plane that slice the cube.





Assuming the label y is either positive (1) or negative (-1) and all those three lines below is known as separating hyperplanes. Separate hyperplane constraint is written mathematically above because they all share the same properties. In a perfect case a linear separable case, the constraint can be met by Support Vector Machine but if the case of non-separable, we will need to loosen it. M**argin** is where we have a hyperplane that is a line X calculate the perpendicular distance from line X to all those 40 dots, it will be 40 different distances Out of the 40 and the shortest distance will be our margin. The distance between either side of the dashed line to the solid line is the margin. We can think of this optimal line as the mid-line of the widest stretching we can possibly have between blue and green dots.



Linear seperable cases of support vector machine have constrain that each make observation is on the correct side of the Hyperplane and then picks up the optimal line so that the distance from those closest dots to the Hyperplane, so-called margin, is maximize. They try to find the hyperplane that will maximize the margins with the condition that both classes are classified correctly. But the reality is that the datasets are probably never linearly separable, so the condition of 100% correctness should be classified by a hyperplane will never be met. Also support vector machine addresses non-linearly separable cases by introducing the two concepts of **Soft Margin** and **Kernel Tricks.** If we add one blue dot in the green cluster, the dataset becomes linear non separable. This is a problem and its solutions are **Soft Margin that will** try to find a line to separate, but tolerate one or few misclassified dots and the other is **Kernel Trick that** try to find a non-linear decision boundary. There are two type of misclassifications tolerated by support vector machine under soft machine which are the dot is on the wrong side of the decision boundary but on the correct side on the margin or the dot might be on the wrong side of the decision boundary and on the wrong side of the margin. When we use Soft Margin, SVM may tolerates a few dots that get misclassified and will try to balance the trade-off between finding a

line that maximizes the margin and minimizes the misclassification. The degree of tolerance is how much tolerance we want to give when we try to find the decision boundary and it is an important hyper-parameter for the SVM .It is represented as the penalty term 'C ' In Sklearn. The more penalty SVM gets when it makes misclassification the larger the C and the narrower the margin is and fewer the decision boundary will depend on the support vector. Kernel Trick utilizes the existing features, applies some transformations and creates new features. These new features helps find the nonlinear decision boundary and are key to SVM. The svm.SVC() method in Sklearn, have choices that are linea, poly, rbf, sigmoid, precomputed that are callable as our kernel/transformation. The two most popular kernels are the Polynomial and Radial Basis Function(RBF). ***Polynomial Kernel is*** a transformer or processor that generates new features by applying the polynomial combination of all the existing features. i.e. Existing Feature: X = np.array([-2,-1,0, 1,2]) and Label: Y = np.array([1,1,0,1,1])

it's impossible for us to find a line to separate them.But, if we apply transformation X^2 to get:New Feature: X = np.array([4,1,0, 1,4]) by combing the existing and new feature, certainly we can draw a line that separates the dots .Support vector machine with a polynomial kernel can generate a non-linear decision boundary using those polynomial features. ***Radial Basis Function (RBF) kernel is*** a transformer/processor that generates new features by measuring the distance between all the other dots to a specific dot at the centers. The influence of new features is controlled by **gamma (γ)** and **Φ(x, center) is** on the decision boundary. If there are more influence of the features on the decision boundary the higher the gamma will be and the more wiggling the boundary will be. By combining both the **soft margin** which is tolerance of misclassifications and **kernel trick**, Support Vector Machine will be able to structure the decision boundary for linear non-separable cases. The wiggling of the SVM decision boundary is controlled by gamma and hyper-parameters like thus when there are more penalty given to SVM when it misclassified the higher the C, and therefore the less wiggling the decision boundary will be and when there are more influences on the feature data points on the decision boundary the higher the gamma, therefore the more wiggling the boundary will be.

Implementing Support Vector Machine:

```
# Support Vector Machine
import sklearn
from sklearn.svm import SVC, LinearSVC
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
import numpy as np
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import reciprocal, uniform
# Data scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train.astype(np.float32))
X_test_scaled = scaler.transform(X_test.astype(np.float32))
# Implement simple linear SVM
svm_clf = SVC(C=28.0, gamma = 0.00001, decision_function_shape="ovr") #These para
meters can be modified

# Fit model
svm_clf.fit(X_train, y_train)
#Make predictions
y_predict = svm_clf.predict(X_test)
# Accuracy
acc = svm_clf.score(X_test, y_test)
print("accuracy=%0.4f" %acc)
```

The above code is how I implemented the support vector machine classifier on the dataset. First I imported the necessary modules then I loaded the data and stored them and X and y variables where x stored the features or the independent variables and y stored the labels or the dependent variable .Then I split the data into training set and testing set using train test split method from sklearn . Then I initialized the classifier which is SVC with three parameters of c which I set to 28,gamma which I set to 0.00001 and decision function shape that I set to ovr and these parameters can be modified to improve the accuracy of the algorithm and I assigned it to svm_clf variable. Then I trained my data by using the fit method and I passed two parameters or arguments into it which are the training set. Then I did my predictions by passing the testing set into the predict method of svm_clf variable. Then I printed out the prediction and I calculated the accuracy of my model which was 0.8194 or 81.94%.

1.2 Multilayer Perceptron

There can be more than one linear layer combinations of **neurons** in the Multilayer perceptron.

The first layer will be the *input layer* and last will be *output layer* and middle layer will be

called *hidden layer* for a simple three-layered network multiplayer perceptron.*We* take the output from the output layer and We feed our input data into the input layer . To make the model more complex according to our task then we can increase the number of the hidden layer as much as we want. The goal of a Feed Forward Network is to approximate some function f () and is the most typical neural network model. The multilayer perceptron tries to find the best approximation to a classifier $\mathbf{y = f * (x)}$ that maps an input of x to an output of class y by defining a mapping of $y = f(x; \theta)$ and the best learning parameters θ for it. The multilayer perceptron networks are composed of many functions that are chained together and the form f(x) = f (3)(f (2)(f (1)(x))) is a network with three layers or functions. Each of these functions or layers are composed of a number of units that perform an affine transformation of a linear sum of inputs and $y = f(WxT + b)$ is a formula that represents each layer . in this formula W is the set of parameter or weights in the layer, f is the activation function, b is the bias vector and x is the input vector which can also be the output of the previous layer. The layers or functions of a multilayer perceptron consists of several fully connected layers because each unit in a layer is connected to all the units in the previous layer. The parameters of each unit are independent of the rest of the units in the layer in a fully connected layer which means that each unit possess a number of unique set of weights.Each input vector is associated with a label, or ground truth which defines its class or class label which is given with the data for a supervised classification system. For each input, the output of the network gives a class score, or prediction. The loss function is defined to measure the performance of the classifier. *In case* the predicted class does not correspond to the true class the loss will be high and it will be low otherwise. At the time of training the model problem of overfitting and under fitting occurs sometimes. If that is the case, the model might perform very well on training data but not on testing data. An optimization procedure is required in order to train the network and for this we need an optimizer and loss function. This process minimizes the loss function by finding the values for the set of weights, W. To get a lower loss we use a popular strategy that initializes the weights to random values and refine them iteratively. This refinement is achieved by moving on the direction defined by the gradient of the loss function. In every

iteration it is very important to set a learning rate defining the amount in which the algorithm is moving. Activation functions also known non- linearity, describe the input-output relations in a non-linear way. This gives the model power to be more flexible in describing arbitrary relations. Here are some popular activation functions Sigmoid, Relu, and TanH. To train the model we basically use the following three steps which are Forward pass, Calculate error or loss and Backward pass. In forward pass step of training the model, we just pass the input to model and multiply with weights and add the bias at every layer and find the calculated output of the model. To Calculate error or loss when passing the data instance we get some output from the model that is called predicted output or pred_out and we have the label with the data that is real output or expected output or Expect_out. Based upon these both we calculate the loss that we have to backpropagate (using Backpropagation algorithm). There are various loss function that we use based on our output and requirement. After calculating the loss, we backpropagate the loss and updates the weights of the model by using gradient. This is the main step in the training of the model. In this step, weights will adjust according to the gradient flow in that direction.

Implementing Multilayer Perceptron:

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD
from sklearn.model_selection import train_test_split
X =  np.load('/content/feat.npy')
y =  np.load('/content/label.npy').ravel()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=0)
# Build the Neural Network
model = Sequential()
model.add(Dense(512, activation='relu', input_dim=193)) ## Dense method for MLP
model.add(Dropout(0.5))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy',
        optimizer='rmsprop',
        metrics=['accuracy'])
# Convert label to onehot encoding
```

```
y_train = keras.utils.to_categorical(y_train-
1, num_classes=10) # Convert class vector into binary Matrix
y_test = keras.utils.to_categorical(y_test-1, num_classes=10)
# Train and test
model.fit(X_train, y_train, epochs=100, batch_size=64) # Epochs are tunable
score, acc = model.evaluate(X_test, y_test, batch_size=32)
print('Test score:', score)
print('Test accuracy:', acc)
Test score: 3.8469386458396913
Test accuracy: 0.793749988079071
```

The code above shows how I implemented the multilayer perceptron to train and predict the outcome of my dataset. First I started by importing the modules I needed to use and I was using keras to implement my neural network. Then I loaded the data into X and y which represented the features and labels respectively. Then I split the data into training and testing set using train test split method from sklearn. Then I started building my Neural Network model and I started by initializing Sequential and assigning it to the model variable. Then I added several layers of my network with different activation functions which are relu and softmax. Then I used the loss function of multiple class classification which was categorical cross entropy and I used adam's optimizer and set the metrics to accuracy. Then I converted the labels into onehot encoding to convert class vectors to binary class matrix for both the training set and testing set. Then I converted the input for model training from a two dimensional to three dimensional and trained the network with 100 epochs until it converged. Then I computed the accuracy and loss of my model and I got test score of 3.8469386458396913 and a test accuracy of 0.793749988079071 which is 79.37499%.This was the highest accuracy of the neural networks I implemented.

2 Convolution Neural Network

An algorithm that takes in an input data and assign important learnable weights and biases to various aspects or objects in the data and can differentiate one from the other is referred to as a convolution Neural Network. When we compare it to other classification algorithms it has a lower pre-processing .Convolution Neural Network is able to learn filters or characteristics with enough training unlike primitive methods filters that are hand-engineered. The connectivity pattern of Neurons in the Human Brain is analogous to the connectivity of a convolution Neural Network .A Receptive Field is the Individual neurons respond to stimuli only in a restricted region of the visual field and a collection of such fields overlap to cover the entire visual area. Through the application of relevant filters, a convolution Neural Network captures the Spatial and Temporal dependencies of the dataset successfully. Due to the reduction in the number of parameters involved and reusability of weights the architecture performs a better fitting to the dataset. Without losing features which are critical for to get a good prediction the role of the convolution Neural Network is to reduce the dataset into a form which is easier to process. When we are try to design an architecture it is important to reduce the dataset into form which is easier to process which is good at learning features but also is scalable to massive datasets.

Implementing Convolution Neural Network:

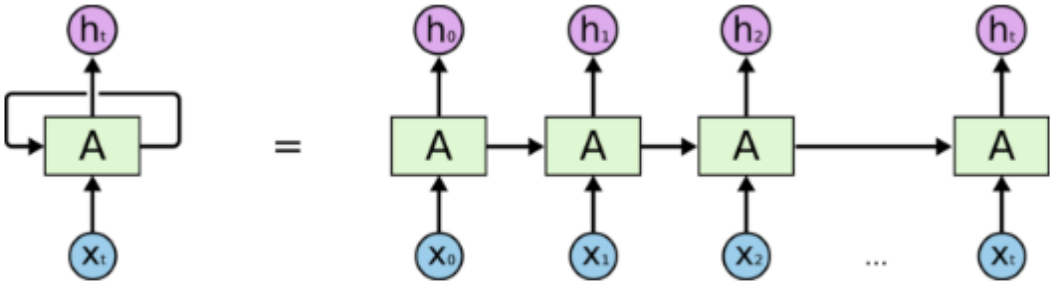
```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalAveragePooling1D, MaxPooling1D
from keras.optimizers import SGD
from sklearn.model_selection import train_test_split
# Load data
X = np.load("feat.npy")
y = np.load('label.npy').ravel()

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.4, random_state = 23
3)

# Neural Network Construction
model = Sequential()
# Network Architecture
model.add(Conv1D(64, 3, activation='relu', input_shape = (193, 1)))
model.add(Conv1D(64, 3, activation='relu'))
model.add(MaxPooling1D(3))
model.add(Conv1D(128, 3, activation='relu'))
model.add(Conv1D(128, 3, activation='relu'))
model.add(GlobalAveragePooling1D())
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
# Most used loss function for muliple-class classification
model.compile(loss='categorical_crossentropy',
        optimizer='adam',
        metrics=['accuracy'])
```

```
# Convert label to onehot encoding
y_train = keras.utils.to_categorical(y_train - 1, num_classes=10) # Converts a class vecto
r (integers) to binary class matrix
y_test = keras.utils.to_categorical(y_test - 1, num_classes=10)
# Make 2-dim into 3-dim array for input for model training
X_train = np.expand_dims(X_train, axis=2)
X_test = np.expand_dims(X_test, axis=2)

# Train Network
model.fit(X_train, y_train, batch_size=64, epochs=100) # Epochs are tunable
# Compute Accuracy and Loss
score, acc = model.evaluate(X_test, y_test, batch_size=16)
print('Test score:', score)
print('Test accuracy:', acc)
Test score: 0.8873702585697174
Test accuracy: 0.78125
```

The code above shows how I implemented the convolution neural network to train and predict the outcome of my dataset. First I started by importing the modules I needed to use and I was using keras to implement my neural network. Then I loaded the data into X and y which represented the features and labels respectively. Then I split the data into training and testing set using train test split method from sklearn. Then I started building my Neural Network model and I started by initializing Sequential and assigning it to the model variable. Then I added several layers of my network with different activation functions which are relu and softmax and maxpooling. Then I used the loss function of multiple class classification which was categorical cross entropy and I used adam's optimizer and set the metrics to accuracy. Then I converted the labels into onehot encoding to convert class vectors to binary class matrix for both the training set and testing set. Then I converted the input for model training from a two dimensional to three dimensional and trained the network with 100 epochs until it converged. Then I computed the accuracy and loss of my model and I got test score of 0.88737 which is 88.737% and a test accuracy of 0.78125 which is 78.125%.

1 Recurrent Neural Network.

A set of algorithms that closely resemble the human brain and have been designed to recognize patterns are known as neural networks. They use a machine perception, labelling or clustering raw input to interpret sensory data. All real-world data (images, sound, text or time series), must be translated into numerical patterns contained in vectors so that the neural network algorithm can recognize the data. Recurrent neural network is a type of neural network that has an internal memory and can be generalization of feedforward neural network. A feed forward neural network is where the connection between units do not form a cycle and they are artificial neural networks which are composed of a large number of highly interconnected processing elements (neuron) working together to solve a problem. They are used for supervised learning in cases where the data to be learned is neither sequential nor time-dependent. The output of the current input depends on the past one computation because the recurrent neural network is recurrent in nature because it performs the same function for every input of data that is given. They make a decision after they consider the current input and the output that has learned from the previous input data. The output that is produced is copied and sent back into the recurrent network. They have internal state memory which they can use to process sequences of inputs data unlike the feedforward neural networks. Recurrent Neural networks are mostly used in unsegmented, connected handwriting recognition or speech recognition. The inputs are related to each other in recurrent neural network unlike other neural networks where the inputs are independent of each other. The diagram below shows an example of an unrolled recurrent neural network:



An unrolled recurrent neural network.

Its outputs h(0) which together with X(1) is the input after it takes the X(0) from the sequence of input. After this first step h(0) and X(1) is the input for the next step. it keeps remembering the context while training because h(1) from the next is the input with X(2) for the next step and so

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}),$$

on. Its current state formula is:
h(t) is current hidden state and is the current input x(t) and is a function f of the previous hidden state h(t-1).The symbol theta is the parameters of the function f.
When we apply the activation function we get:

$$h_t = \tanh\left(W_{hh}h_{t-1} + W_{xh}x_t\right)$$

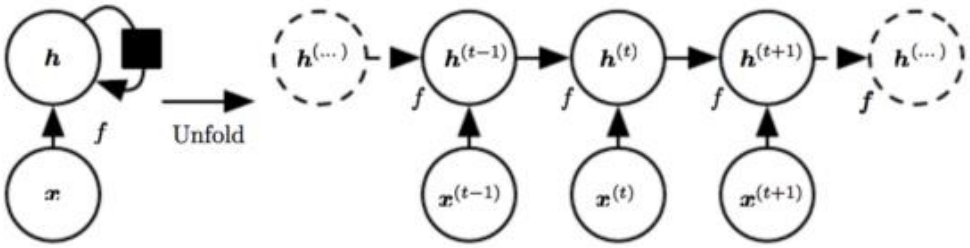


Figure 10.2: A recurrent network with no outputs. This recurrent network just processes

The figure below shows how unfolding maps from left to right where the black square indicates that an interaction takes place with a delay of a time step(i.e 1), from the state at time t to the state at time t + 1.Unfolding or parameter sharing is better than using different parameters per position: less parameters to estimate, generalize to various length.

Advantages of Recurrent Neural Network:

- Recurrent neural network can model sequence of data so that each sample can be assumed to be dependent on previous ones
- Recurrent neural network an be used with convolutional layers to extend the effective pixel neighbourhood.

Disadvantages of Recurrent Neural Network:

- The gradient vanishes and has exploding problems.
- Training an RNN is a very difficult task.
- It ‘s unable to process very long sequences if using relu as an activation function.

Implementing Recurrent Neural Network:

```
####### Recurrent Neural Network #####
import os
import numpy as np
from sklearn.model_selection import train_test_split
import keras
from keras.models import Sequential
from keras.layers.recurrent import LSTM
from keras.layers import Dense
from keras.optimizers import Adam
# Load data
X = np.load("feat.npy")
y = np.load('label.npy').ravel()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=233)
```

```
# Reshape data for LSTM (Samples, Timesteps, Features)
X_train = np.expand_dims(X_train, axis=2)  # (280,193,1)
X_test = np.expand_dims(X_test, axis=2)
y_train = keras.utils.to_categorical(y_train - 1, num_classes=10)  # Converts a class vect
or (integers) to binary class matrix
y_test = keras.utils.to_categorical(y_test - 1, num_classes=10)
# Build RNN Neural Network
print('Build LSTM RNN model ...')
model = Sequential()
model.add(LSTM(128, return_sequences=True, input_shape=X_train.shape[1:]))
model.add(LSTM(32, return_sequences=False))
model.add(Dense(y_train.shape[1], activation='softmax'))
print("Compiling ...")
model.compile(loss='categorical_crossentropy',  # loss function for multi-classification
        optimizer='adam',
        metrics=['accuracy'])

print(model.summary())
print("Training ...")
model.fit(X_train, y_train, batch_size=35, epochs=100)
print("\nValidating ...")
score, accuracy = model.evaluate(X_test, y_test, batch_size=35, verbose=1)
print("Loss:  ", score)
print("Accuracy:  ", accuracy)
Loss:   0.9351997882127762
Accuracy:   0.7200000286102295
```

The code above shows how I implemented the recurrent neural network to train and predict the outcome of my dataset. First I started by importing the modules I needed to use and I was using keras to implement my neural network. Then I loaded the data into X and y which represented the features and labels respectively. Then I split the data into training and testing set using train test split method from sklearn. Then I started building my Neural Network model and I started by initializing Sequential and assigning it to the model variable. Then I added several layers of my network with different activation functions which were relu and softmax. Then I used the loss function of multiple class classification which was categorical cross entropy and I used adam's optimizer and set the metrics to accuracy. Then I converted the labels into onehot encoding to convert class vectors to binary class matrix for both the training set and testing set. Then I converted the input for model training from a two dimensional to three dimensional and trained the network with 100 epochs until it converged. Then I computed the accuracy and loss of my model and I got test score of 0.935199 which is 91.5199% and a test accuracy of 0.7200 which is 72%.

Conclusion

After running al the algorithm I found out that random forest had the highest accuracy of all the supervised learning classification algorithm and it had an accuracy of 83% and support vector machine came a second close with an accuracy of 81%.For Neural Networks I used three algorithms and after comparing the accuracy I found out that multilayer perceptron had the highest accuracy followed by convolution neural network. There is a room to improve the accuracy of the

algorithm and one can try to adjust the hyper parameters and use pipelines which can help improving the accuracy of the supervised learning model.

References:

https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e
https://en.wikipedia.org/wiki/Recurrent_neural_network
https://www.datacamp.com/community/tutorials/convolutional-neural-networks-python
https://www.datacamp.com/community/tutorials/deep-learning-python
https://machinelearningmastery.com/naive-bayes-classifier-scratch-python/
https://scikit-learn.org/stable/modules/naive_bayes.html
https://en.wikipedia.org/wiki/Random_forest#:~:text=Random%20forests%20or%20random%20decision,prediction%20(regression)%20of%20the%20individual