Music Genre Based Playlist Generator

Machine Learning Project-Phase2

Submitted by:
Bharath Prathap Nair
(AM.EN.U4CSE19113)
S5 CSE B

Problem Definition

The aim of the project is to implement an algorithm to sort and create a song playlist based on our mood at a point of time or a playlist classified based on different genres made by analyzing various mood factors like danceability, loudness etc in a song.

Dataset

The datasets used in the project are the data derived from the spotify API derived from various sources like Kaggle. The datasets contain details about various moods that a song possesses based on which the songs are classified to different Genres.

Source : Kaggle
[https://www.kaggle.com/cnic92/spotify-past-decades-songs-50s10s?select=2010.csv]

Our dataset contains details about the top 100 songs in 2010-2019. The dataset contains 15 columns including the target variable - genre. The dataset has 15 columns with 100 entries.The target column is **top genre**.

Features :

- Number: ID
- title: Name of the Track
- artist: Name of the Artist
- year: Release Year of the track
- bpm: The tempo of the song
- nrgy: The energy of a song - the higher the value, the more energetic. song
- dnce: The higher the value, the easier it is to dance to this song.
- dB: The higher the value, the louder the song.
- live: The higher the value, the more likely the song is a live recording
- val: The higher the value, the more positive the mood for the song.

- dur: The duration of the song.
- acous: The higher the value the more acoustic the song is.
- spch: The higher the value the more spoken words the song contains
- pop: The higher the value the more popular the song is.

```
   Number                      title          artist     top genre  year  bpm  \
0       1                    bad guy   Billie Eilish    electropop  2019  135
1       2                    7 rings   Ariana Grande     dance pop  2019  140
2       3        Old Town Road - Remix       Lil Nas X   country rap  2019  136
3       4                  SeÃ±orita    Shawn Mendes  canadian pop  2019  117
4       5  rockstar (feat. 21 Savage)     Post Malone       dfw rap  2018  160

   nrgy  dnce  dB  live  val  dur  acous  spch  pop
0    43    70 -11    10   56  194     33    38   94
1    32    78 -11     9   33  179     59    33   90
2    62    88  -6    11   64  157      5    10   89
3    55    76  -6     8   75  191      4     3   88
4    52    59  -6    13   13  218     12     7   88
```

Data Cleaning/Data Pre-Processing

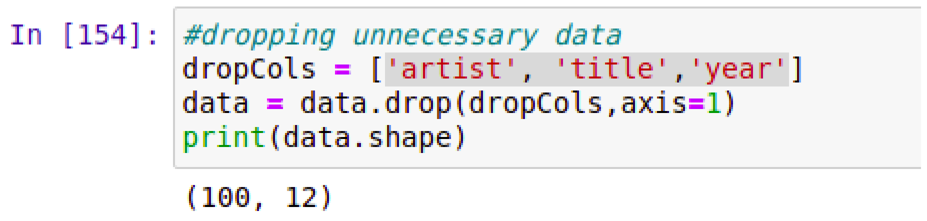
Firstly we first check for null values and remove them if they exit.

```
]: #Checking for null values
   print(data.isnull().sum())
```

```
Number       0
title        0
artist       0
top genre    0
year         0
bpm          0
nrgy         0
dnce         0
dB           0
live         0
val          0
dur          0
acous        0
spch         0
pop          0
dtype: int64
```

Here we can see that none of the columns are having null values.

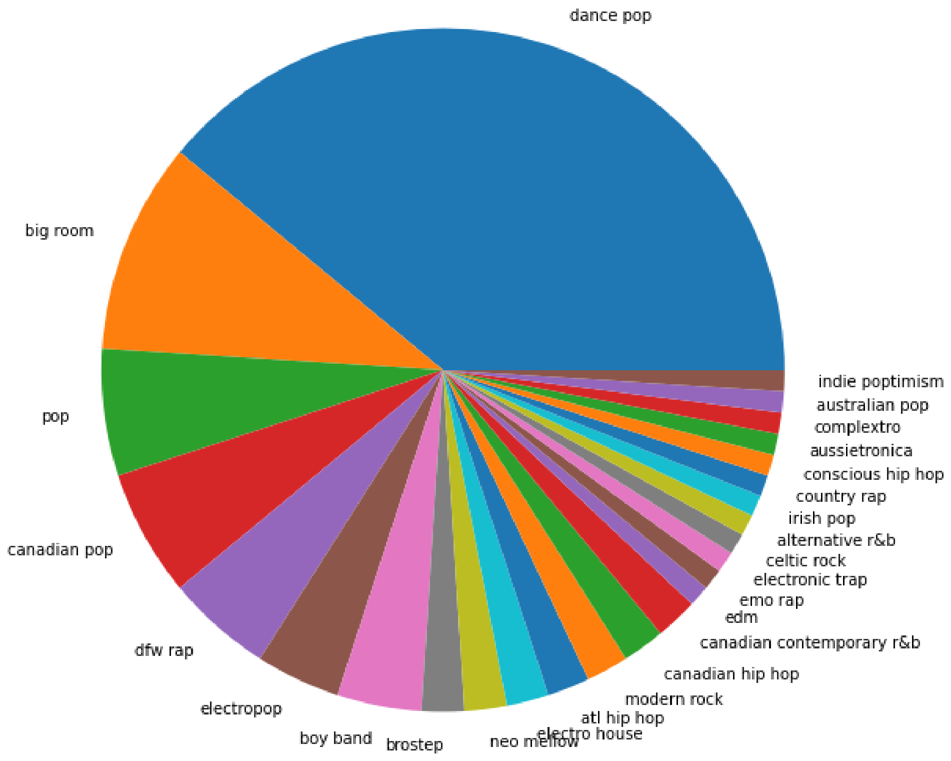
Secondly, in this given dataset there are few details that are not necessary for our study. So there's a need of removing those unnecessary data. For this, we drop the columns '**artist', 'title'** and '**year'**.



```
In [154]: #dropping unnecessary data
          dropCols = ['artist', 'title','year']
          data = data.drop(dropCols,axis=1)
          print(data.shape)

          (100, 12)
```

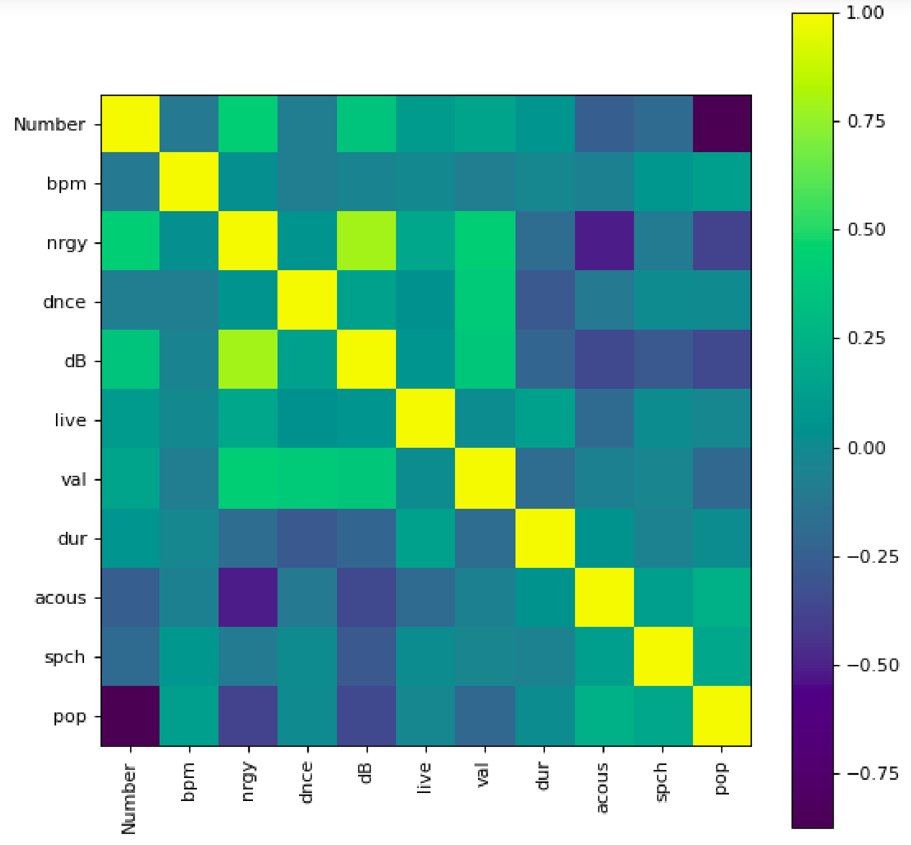
Data Visualisation

1.Plotting pie chart of the number of songs per genre



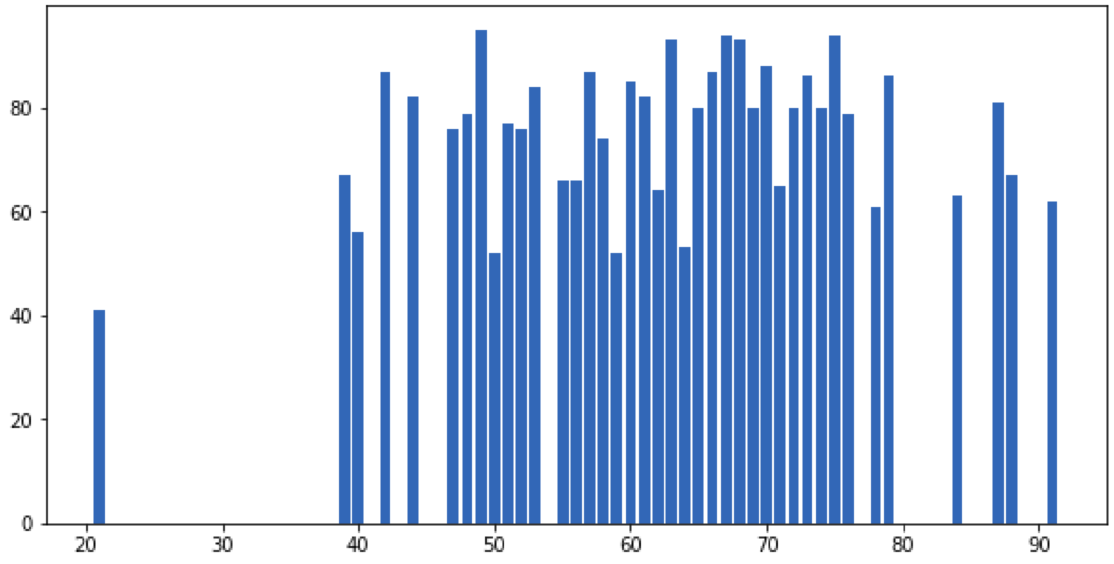
Here we have plotted the number of songs per each genre.We can see that dance pop songs are the most , followed by big room songs and Pop.

2. Correlation Matrix Plot



In this correlation matrix, we can see that any positive value shows that an attribute is more dependent on the other attribute . Any negative value shows that how much an attribute is less dependent on the other attribute.

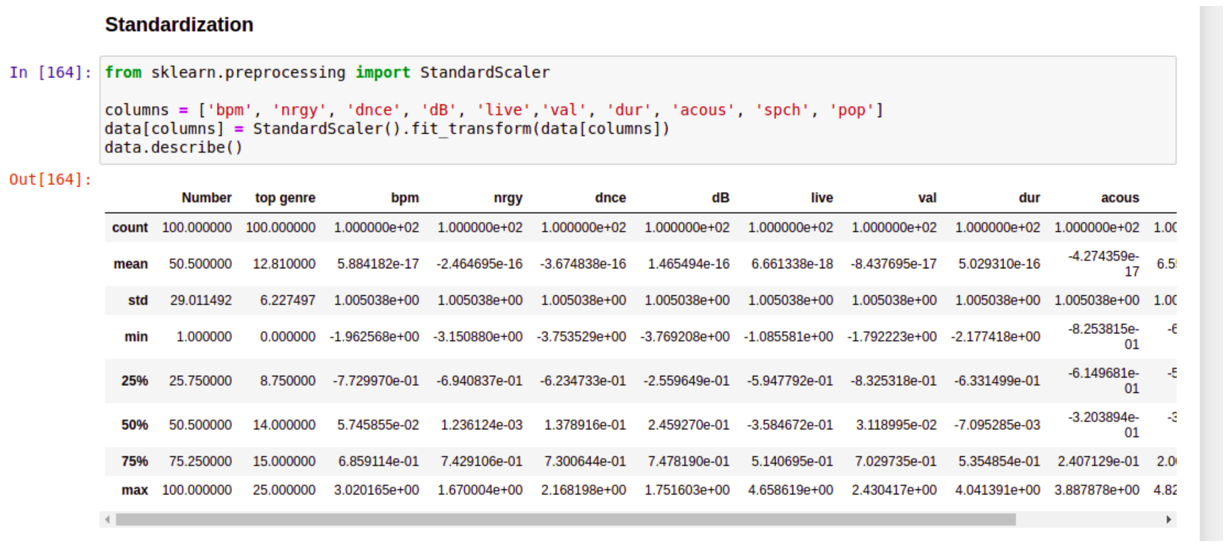
3. Bar Plot



Here we are plotting a bar graph showing the relation between dnce and nrgy .

Standardisation

In our dataset there might exist huge differences in the data values.So inorder to remove this, we need to standardise our dataset. For standardising our dataset we use StandardScaler() function.



Standardization

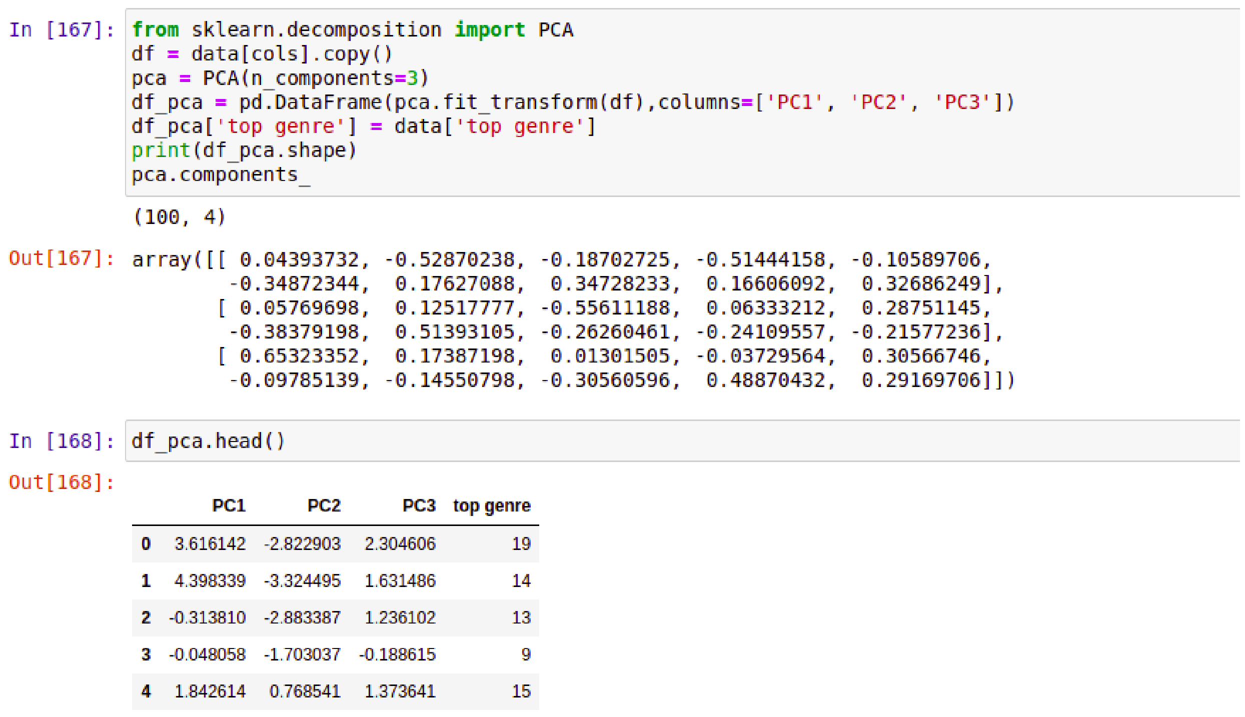
```
In [164]: from sklearn.preprocessing import StandardScaler

          columns = ['bpm', 'nrgy', 'dnce', 'dB', 'live','val', 'dur', 'acous', 'spch', 'pop']
          data[columns] = StandardScaler().fit_transform(data[columns])
          data.describe()
```

Out[164]:

	Number	top genre	bpm	nrgy	dnce	dB	live	val	dur	acous	
count	100.000000	100.000000	1.000000e+02	1.000000e+02	1.000000e+02	1.000000e+02	1.000000e+02	1.000000e+02	1.000000e+02	1.000000e+02	1.00
mean	50.500000	12.810000	5.884182e-17	-2.464695e-16	-3.674838e-16	1.465494e-16	6.661338e-18	-8.437695e-17	5.029310e-16	-4.274359e-17	6.5
std	29.011492	6.227497	1.005038e+00	1.005038e+00	1.005038e+00	1.005038e+00	1.005038e+00	1.005038e+00	1.005038e+00	1.005038e+00	1.00
min	1.000000	0.000000	-1.962568e+00	-3.150880e+00	-3.753529e+00	-3.769208e+00	-1.085581e+00	-1.792223e+00	-2.177418e+00	-8.253815e-01	-6
25%	25.750000	8.750000	-7.729970e-01	-6.940837e-01	-6.234733e-01	-2.559649e-01	-5.947792e-01	-8.325318e-01	-6.331499e-01	-6.149681e-01	-5
50%	50.500000	14.000000	5.745855e-02	1.236124e-03	1.378916e-01	2.459270e-01	-3.584672e-01	3.118995e-02	-7.095285e-03	-3.203894e-01	-3
75%	75.250000	15.000000	6.859114e-01	7.429106e-01	7.300644e-01	7.478190e-01	5.140695e-01	7.029735e-01	5.354854e-01	2.407129e-01	2.0
max	100.000000	25.000000	3.020165e+00	1.670004e+00	2.168198e+00	1.751603e+00	4.658619e+00	2.430417e+00	4.041391e+00	3.887878e+00	4.82

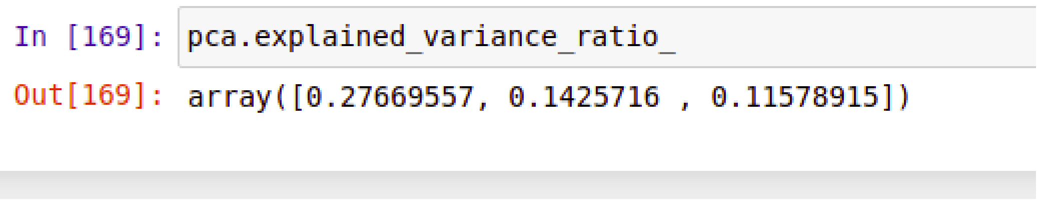
Dimensionality Reduction

We can see that our dataset has around 15 columns , which certainly is a high dimension. For this we need to perform Dimensionality Reduction to reduce the dimension of the dataset. Primary component analysis (PCA) is used to apply dimensionality reduction to our dataset.



```
In [167]: from sklearn.decomposition import PCA
          df = data[cols].copy()
          pca = PCA(n_components=3)
          df_pca = pd.DataFrame(pca.fit_transform(df),columns=['PC1', 'PC2', 'PC3'])
          df_pca['top genre'] = data['top genre']
          print(df_pca.shape)
          pca.components_

          (100, 4)

Out[167]: array([[ 0.04393732, -0.52870238, -0.18702725, -0.51444158, -0.10589706,
                  -0.34872344,  0.17627088,  0.34728233,  0.16606092,  0.32686249],
                 [ 0.05769698,  0.12517777, -0.55611188,  0.06333212,  0.28751145,
                  -0.38379198,  0.51393105, -0.26260461, -0.24109557, -0.21577236],
                 [ 0.65323352,  0.17387198,  0.01301505, -0.03729564,  0.30566746,
                  -0.09785139, -0.14550798, -0.30560596,  0.48870432,  0.29169706]])

In [168]: df_pca.head()

Out[168]:
```

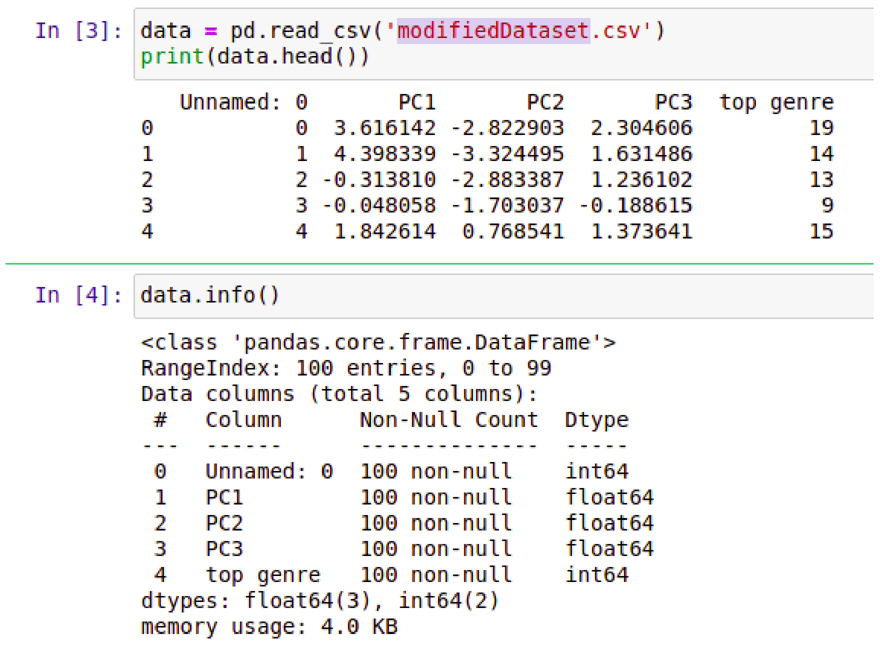
	PC1	PC2	PC3	top genre
0	3.616142	-2.822903	2.304606	19
1	4.398339	-3.324495	1.631486	14
2	-0.313810	-2.883387	1.236102	13
3	-0.048058	-1.703037	-0.188615	9
4	1.842614	0.768541	1.373641	15



```
In [169]: pca.explained_variance_ratio_

Out[169]: array([0.27669557, 0.1425716 , 0.11578915])
```

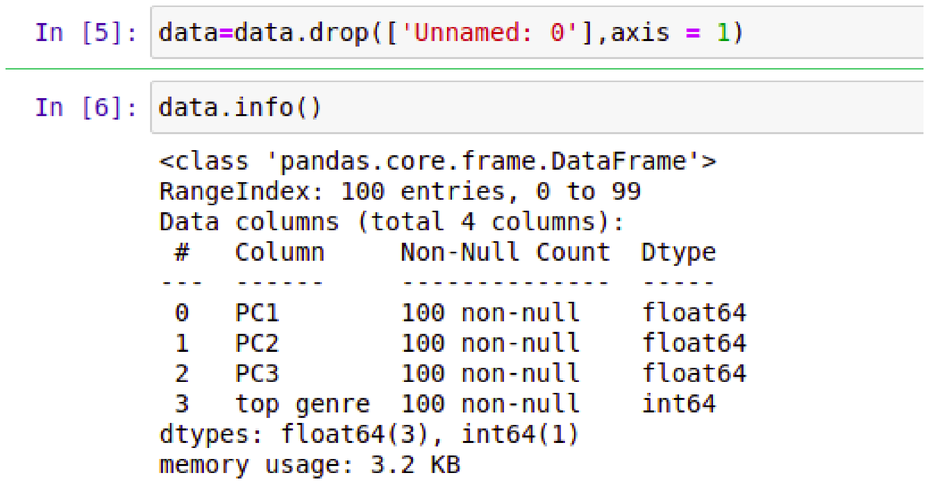
Here we reduce the number of dimensions from 15 to 3.

After performing dimensionality reduction, we export the reduced data as a new csv file named **modifiedDataset.csv .** On exporting the data to a new csv file, we can see that an unnamed column is created showing each data's respective index.



```
In [3]: data = pd.read_csv('modifiedDataset.csv')
        print(data.head())

           Unnamed: 0       PC1       PC2       PC3  top genre
        0           0  3.616142 -2.822903  2.304606         19
        1           1  4.398339 -3.324495  1.631486         14
        2           2 -0.313810 -2.883387  1.236102         13
        3           3 -0.048058 -1.703037 -0.188615          9
        4           4  1.842614  0.768541  1.373641         15

In [4]: data.info()

        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 100 entries, 0 to 99
        Data columns (total 5 columns):
         #   Column      Non-Null Count  Dtype
        ---  ------      --------------  -----
         0   Unnamed: 0  100 non-null    int64
         1   PC1         100 non-null    float64
         2   PC2         100 non-null    float64
         3   PC3         100 non-null    float64
         4   top genre   100 non-null    int64
        dtypes: float64(3), int64(2)
        memory usage: 4.0 KB
```

Since the index is not necessary for our study, we will remove this data using the drop command.



```
In [5]: data=data.drop(['Unnamed: 0'],axis = 1)

In [6]: data.info()

        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 100 entries, 0 to 99
        Data columns (total 4 columns):
         #   Column     Non-Null Count  Dtype
        ---  ------     --------------  -----
         0   PC1        100 non-null    float64
         1   PC2        100 non-null    float64
         2   PC3        100 non-null    float64
         3   top genre  100 non-null    int64
        dtypes: float64(3), int64(1)
        memory usage: 3.2 KB
```

Python Packages

1. **from sklearn.neighbors import KNeighborsClassifier**

This package from Scikit Learn is used to implement KNN classification algorithm. The K in the name of this classifier stands for K nearest neighbors. The classifier implements learning based on the K nearest neighbors for which the value of K is given by the user.

2. **from sklearn.svm import SVC**

This package from Scikit Learn is used to implement SVM or Support Vector Machine Algorithm. SVM is a powerful yet flexible algorithm used for classification, regression, and outlier detection.

3. **from sklearn.linear_model import LogisticRegression**

This package from Scikit Learn is used to implement Logistic Regression algorithm. Despite regression in its name, Logistic Regression is a classification algorithm rather than a regression algorithm.

4. **from sklearn.tree import DecisionTreeClassifier**

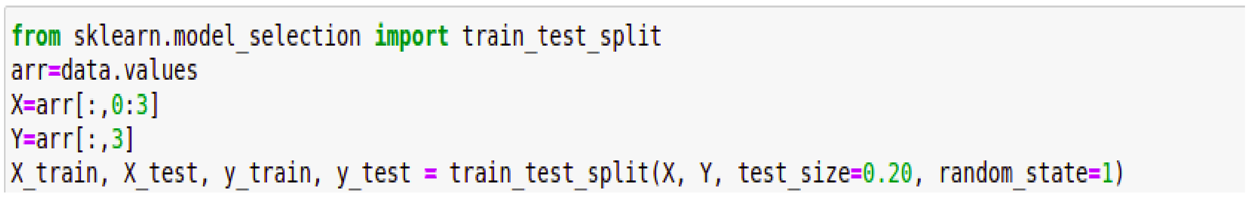
This package from Scikit Learn is used to implement the Decision tree algorithm. This learning method can be used for classification as well as regression tasks.

5. **from sklearn.naive_bayes import GaussianNB**

This package from Scikit Learn is used to implement the Naive Bayes algorithm. Naive Bayes can be implemented in multiple ways using Scikit learn packages. One such way is by using Gaussian Naive bayes . Gaussian Naive Bayes classifier assumes that the data from each label is drawn from a simple Gaussian distribution.

Learning Algorithms

The data we have consists of 4 columns (PC1, PC2, PC3, top genre) and 100 rows. This data now has to be split into training and testing data using the **train_test_split** function. The data is split into the ratio of 80 (training data) and 20 (testing data)



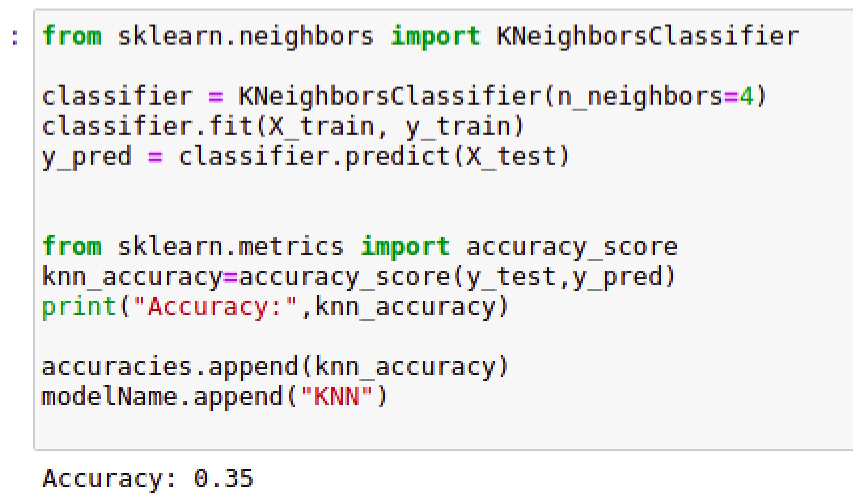
```
from sklearn.model_selection import train_test_split
arr=data.values
X=arr[:,0:3]
Y=arr[:,3]
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.20, random_state=1)
```

KNN

KNN or K Nearest Neighbor is a supervised machine learning algorithm used for classification and regression predictive problems. KNN is a non-parametric learning algorithm because it doesn't assume anything about the underlying data. The K in the name of this classifier stands for K nearest neighbors. The classifier implements learning based on the K nearest neighbors for which the value of K is given by the user. For its implementation, we import KNeighborsClassfier from sklearn library.

Initially, we try implementing the algorithm without applying any kind of cross-validation. Here the value K=4, we get an accuracy of **35%.**

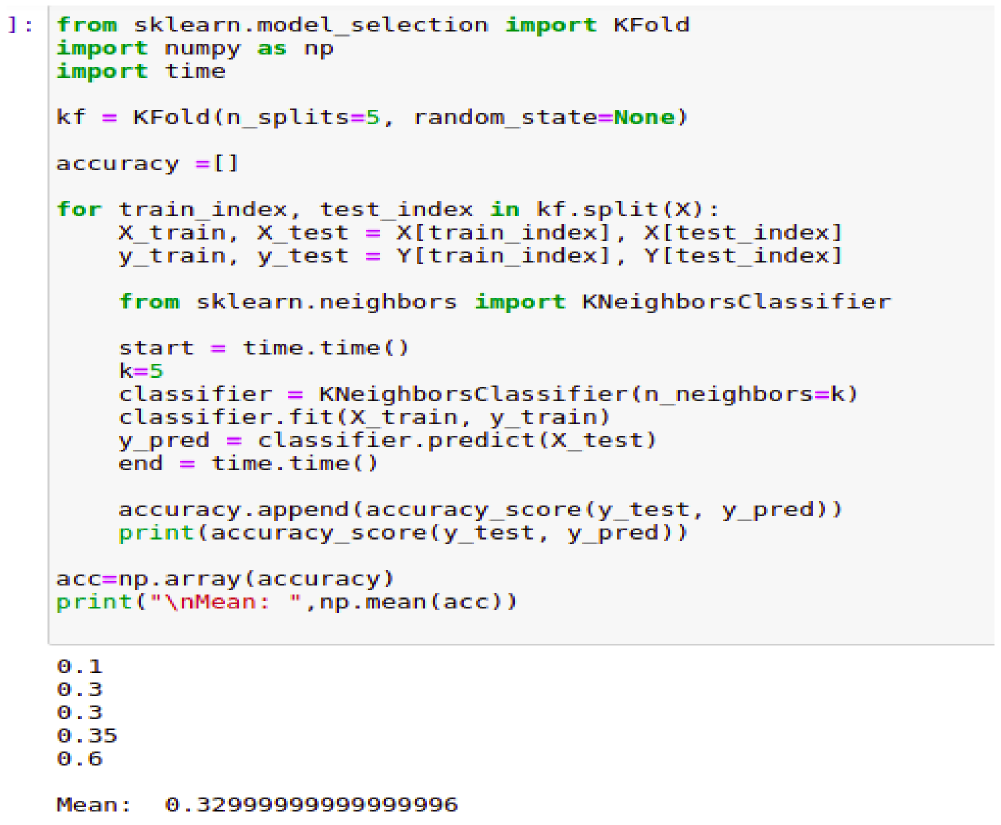
KNN



```
from sklearn.neighbors import KNeighborsClassifier

classifier = KNeighborsClassifier(n_neighbors=4)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)


from sklearn.metrics import accuracy_score
knn_accuracy=accuracy_score(y_test,y_pred)
print("Accuracy:",knn_accuracy)

accuracies.append(knn_accuracy)
modelName.append("KNN")
```

```
Accuracy: 0.35
```

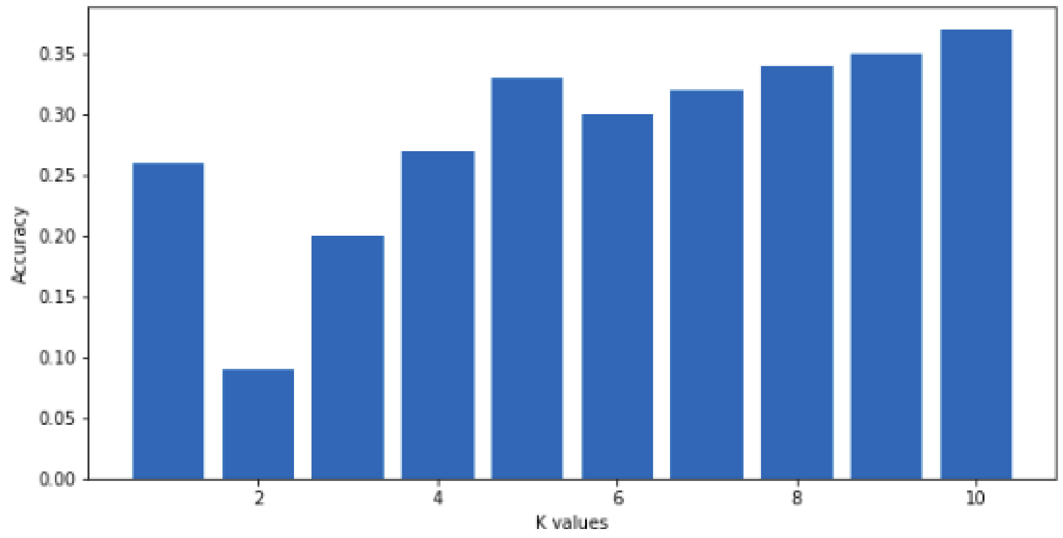
Now we apply K-fold cross-validation with the number of splits during validation as 5 and value of k also as 5. After performing validation and training on the model we got the mean accuracy as **32%.**



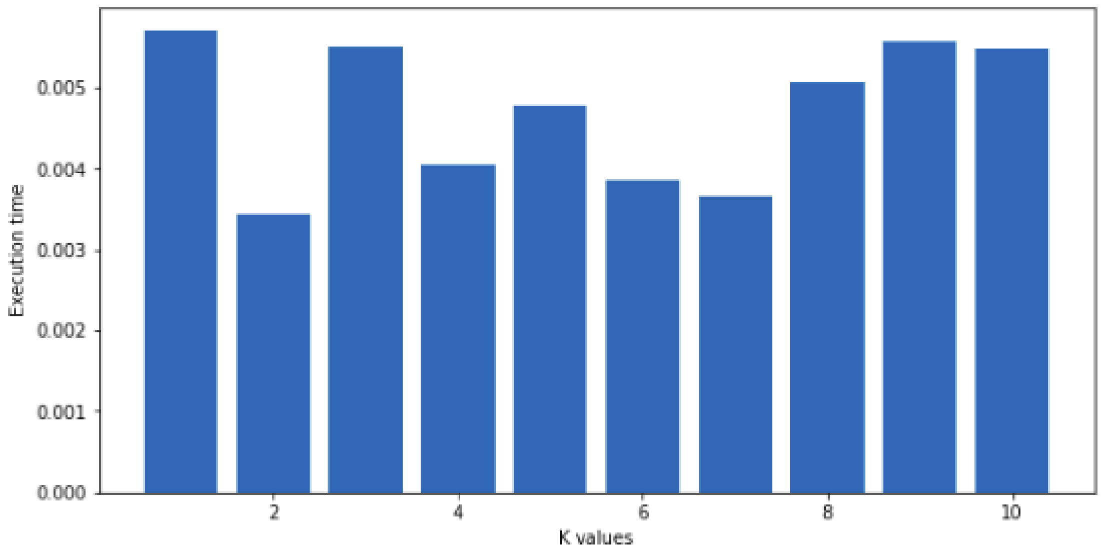
```
from sklearn.model_selection import KFold
import numpy as np
import time

kf = KFold(n_splits=5, random_state=None)

accuracy =[]

for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = Y[train_index], Y[test_index]

    from sklearn.neighbors import KNeighborsClassifier

    start = time.time()
    k=5
    classifier = KNeighborsClassifier(n_neighbors=k)
    classifier.fit(X_train, y_train)
    y_pred = classifier.predict(X_test)
    end = time.time()

    accuracy.append(accuracy_score(y_test, y_pred))
    print(accuracy_score(y_test, y_pred))

acc=np.array(accuracy)
print("\nMean: ",np.mean(acc))
```

```
0.1
0.3
0.3
0.35
0.6

Mean:  0.32999999999999996
```

In addition to this we also performed training on the data on different values of k for which we got varied results as shown below.



We can observe that , we obtain the highest accuracy when k value is 10 and the lowest when k is 2. To observe the execution time taken by each of k , we calculated their respective execution time using the time function and here are the results.

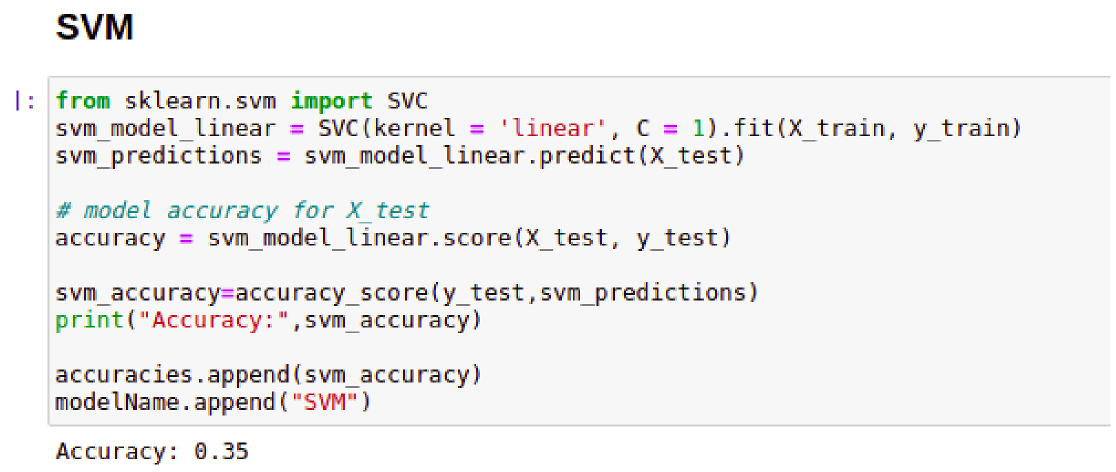


When the value of k is 1 , the execution time is the highest and lowest when k value is 2.

SVM

SVM or Support Vector Machine is a supervised machine learning algorithm which is used for classification, regression, and outlier detection. The goal of the algorithm is to divide the datasets into a number of classes in order to find a maximum marginal hyperplane (MMH). Its advantage is that it has a higher speed and better performance with a limited number of samples .For its implementation, we import SVCfrom sklearn library.

Initially on performing the learning without applying validation , we receive an accuracy value of **35%.**

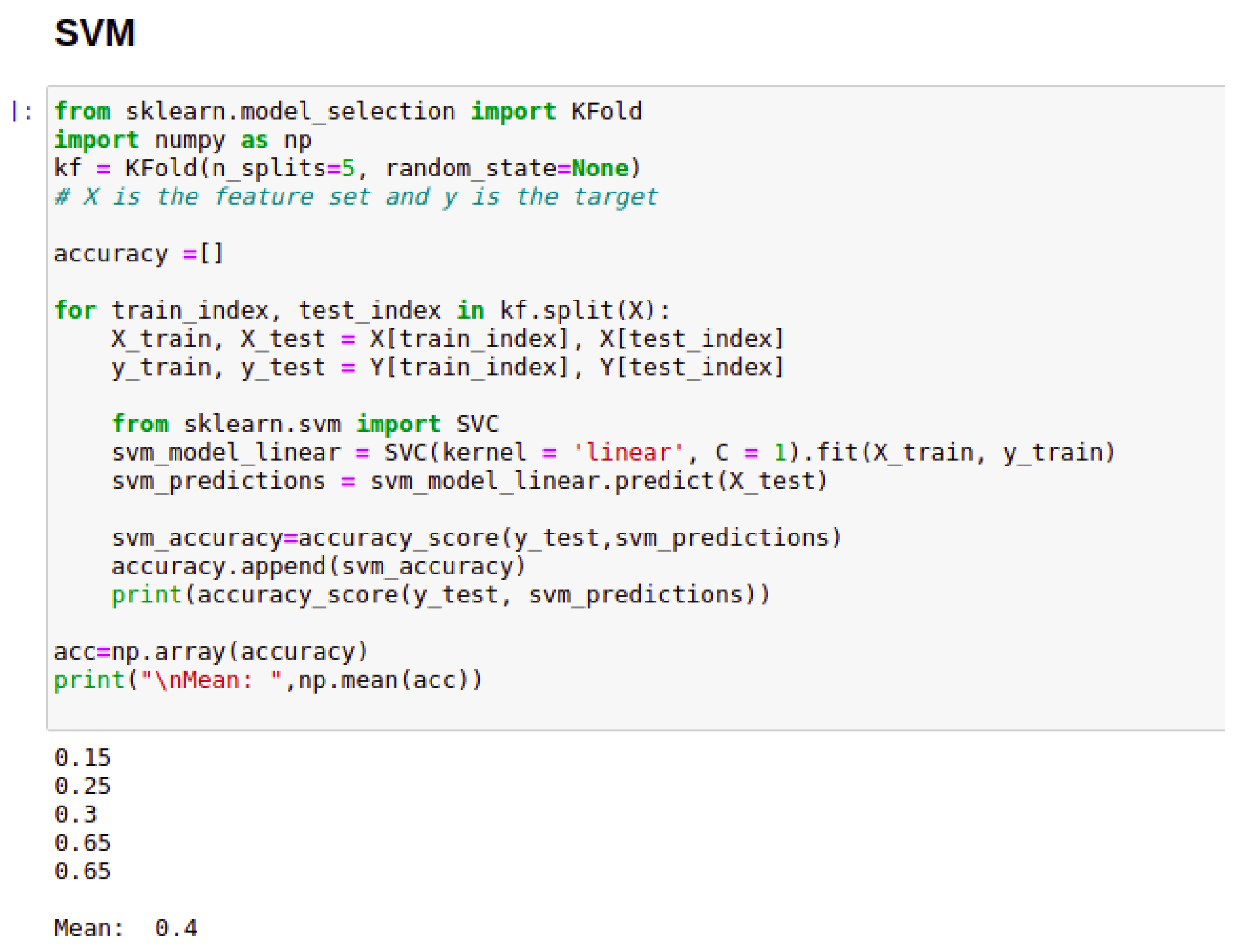


SVM

```
from sklearn.svm import SVC
svm_model_linear = SVC(kernel = 'linear', C = 1).fit(X_train, y_train)
svm_predictions = svm_model_linear.predict(X_test)

# model accuracy for X_test
accuracy = svm_model_linear.score(X_test, y_test)

svm_accuracy=accuracy_score(y_test,svm_predictions)
print("Accuracy:",svm_accuracy)

accuracies.append(svm_accuracy)
modelName.append("SVM")
```

```
Accuracy: 0.35
```

On applying k fold cross validation with 5 as the number of splits, we receive a mean accuracy of **40%.**



SVM

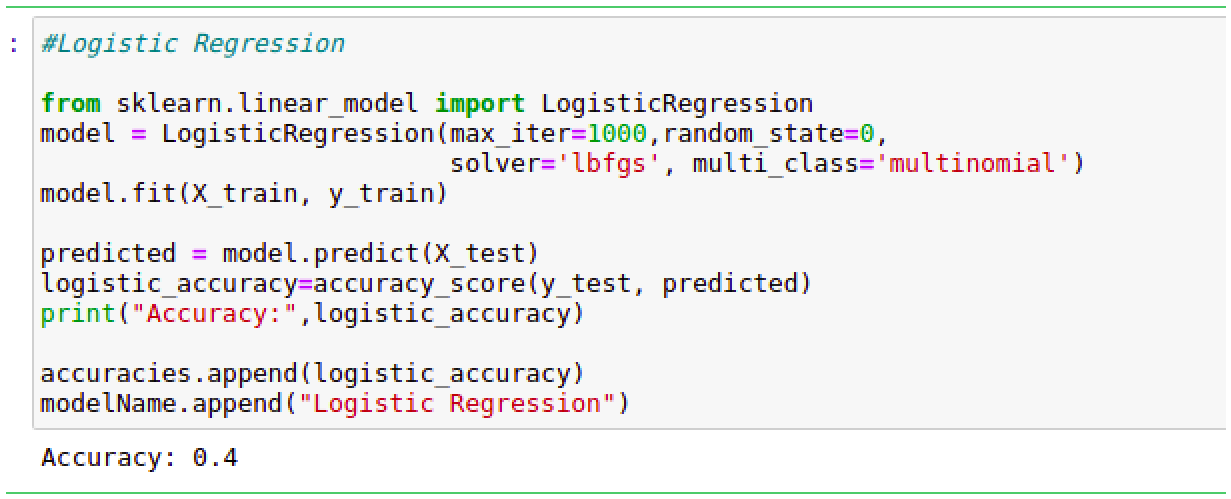
```
[ ]: from sklearn.model_selection import KFold
import numpy as np
kf = KFold(n_splits=5, random_state=None)
# X is the feature set and y is the target

accuracy =[]

for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = Y[train_index], Y[test_index]

    from sklearn.svm import SVC
    svm_model_linear = SVC(kernel = 'linear', C = 1).fit(X_train, y_train)
    svm_predictions = svm_model_linear.predict(X_test)

    svm_accuracy=accuracy_score(y_test,svm_predictions)
    accuracy.append(svm_accuracy)
    print(accuracy_score(y_test, svm_predictions))

acc=np.array(accuracy)
print("\nMean: ",np.mean(acc))
```

```
0.15
0.25
0.3
0.65
0.65

Mean:  0.4
```

Logistic Regression

Logistic Regression is a classification machine learning algorithm despite the regression in its name. Logistic regression is a process of modeling the probability of a discrete outcome given an input variable. It is a classification model, which is very easy to realize and achieves very good performance with linearly separable classes. For its implementation, we import LogisticRegression module from sklearn library.

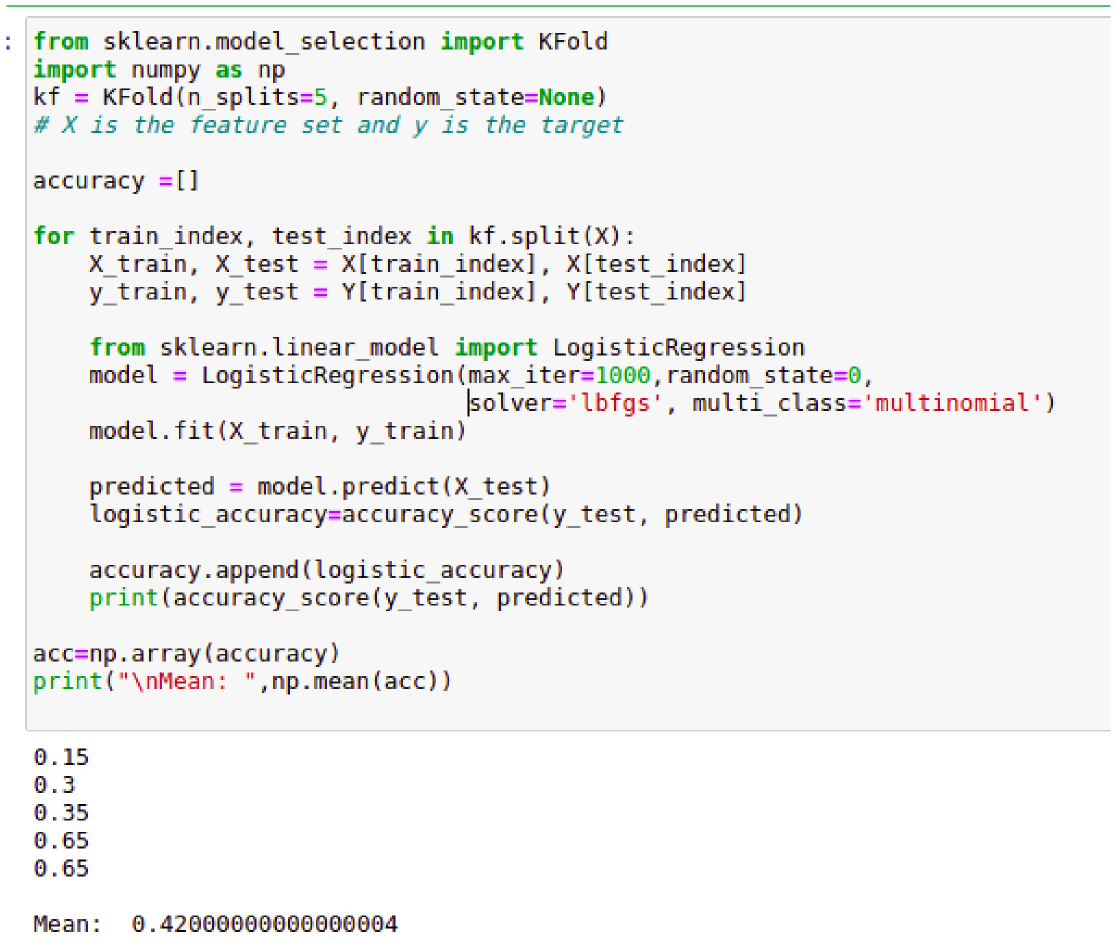
Before applying validation , on training our data with Logistic Regression we receive an accuracy of **40% .**



```
#Logistic Regression

from sklearn.linear_model import LogisticRegression
model = LogisticRegression(max_iter=1000,random_state=0,
                           solver='lbfgs', multi_class='multinomial')
model.fit(X_train, y_train)

predicted = model.predict(X_test)
logistic_accuracy=accuracy_score(y_test, predicted)
print("Accuracy:",logistic_accuracy)

accuracies.append(logistic_accuracy)
modelName.append("Logistic Regression")
```

```
Accuracy: 0.4
```

On applying k fold cross validation with the number of splits as 5, we receive a mean accuracy of **42% .**



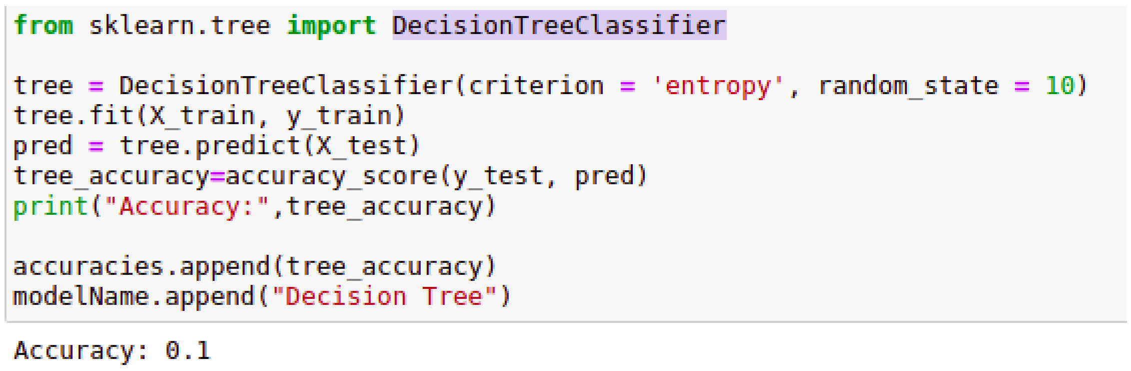
```
from sklearn.model_selection import KFold
import numpy as np
kf = KFold(n_splits=5, random_state=None)
# X is the feature set and y is the target

accuracy =[]

for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = Y[train_index], Y[test_index]

    from sklearn.linear_model import LogisticRegression
    model = LogisticRegression(max_iter=1000,random_state=0,
                               solver='lbfgs', multi_class='multinomial')
    model.fit(X_train, y_train)

    predicted = model.predict(X_test)
    logistic_accuracy=accuracy_score(y_test, predicted)

    accuracy.append(logistic_accuracy)
    print(accuracy_score(y_test, predicted))

acc=np.array(accuracy)
print("\nMean: ",np.mean(acc))
```

```
0.15
0.3
0.35
0.65
0.65

Mean:  0.42000000000000004
```

Decision Tree

Decision tree is a type of supervised Machine Learning where the data is continuously split according to a certain parameter. The tree consists mainly of two entities, decision nodes and leaves. The leaves are the decisions or the final outcomes whereas the decision nodes are where the data is split. Decision trees can be used to solve both classification and regression problems using Classification trees and Regression trees respectively. For the implementation of the decision tree, we import the DecisionTreeClassifier module from sklearn library.

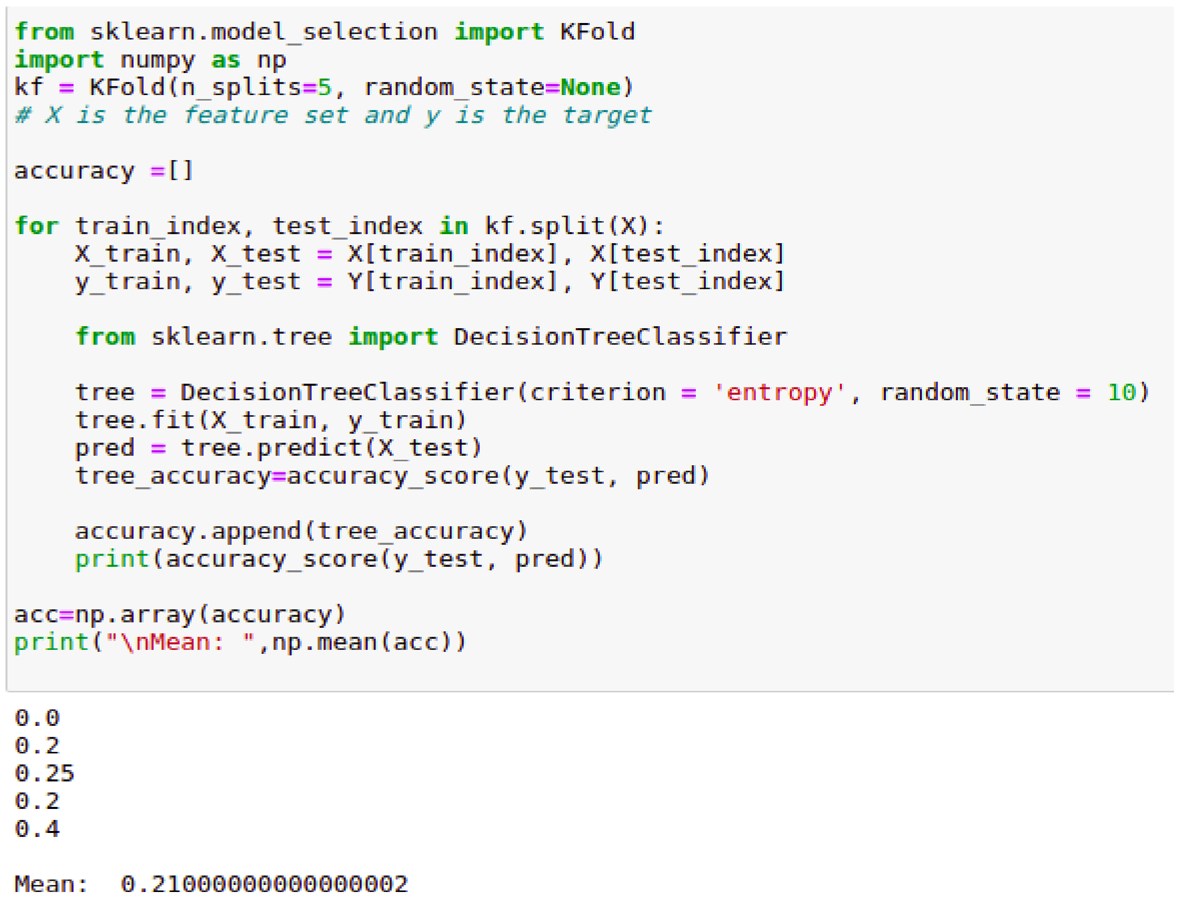
Before applying validation , on training our data with Decision trees we receive an accuracy of **10% .**



```
from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier(criterion = 'entropy', random_state = 10)
tree.fit(X_train, y_train)
pred = tree.predict(X_test)
tree_accuracy=accuracy_score(y_test, pred)
print("Accuracy:",tree_accuracy)

accuracies.append(tree_accuracy)
modelName.append("Decision Tree")
```

```
Accuracy: 0.1
```

On applying k fold cross validation with the number of splits as 5, we receive a mean accuracy of **21% .**



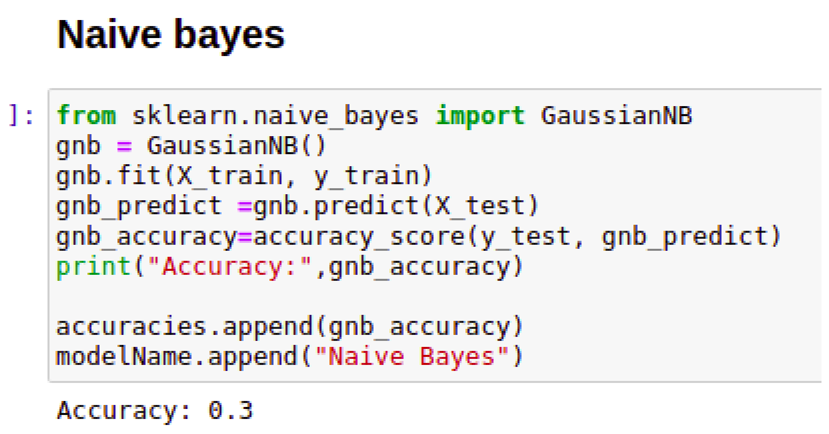
```
from sklearn.model_selection import KFold
import numpy as np
kf = KFold(n_splits=5, random_state=None)
# X is the feature set and y is the target

accuracy =[]

for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = Y[train_index], Y[test_index]

    from sklearn.tree import DecisionTreeClassifier

    tree = DecisionTreeClassifier(criterion = 'entropy', random_state = 10)
    tree.fit(X_train, y_train)
    pred = tree.predict(X_test)
    tree_accuracy=accuracy_score(y_test, pred)

    accuracy.append(tree_accuracy)
    print(accuracy_score(y_test, pred))

acc=np.array(accuracy)
print("\nMean: ",np.mean(acc))
```

```
0.0
0.2
0.25
0.2
0.4

Mean:  0.21000000000000002
```

Naive Bayes

Naive Bayes algorithm is a supervised machine learning algorithm, which is based on Bayes theorem and is used for solving classification problems . Naive Bayes Classifier one of the most effective classification algorithms which helps in building fast machine learning models that can make quick predictions. It is a probabilistic classifier, which means it predicts on the basis of the probability of an object.For the implementation of the Naive Bayes, we import the GaussianNB module from sklearn library. Naive Bayes can be implemented in multiple ways using Scikit learn packages. One such way is by using Gaussian Naive Bayes . Gaussian Naive Bayes classifier assumes that the data from each label is drawn from a simple Gaussian distribution.

Before applying validation , on training our data with Decision trees we receive an accuracy of **30% .**

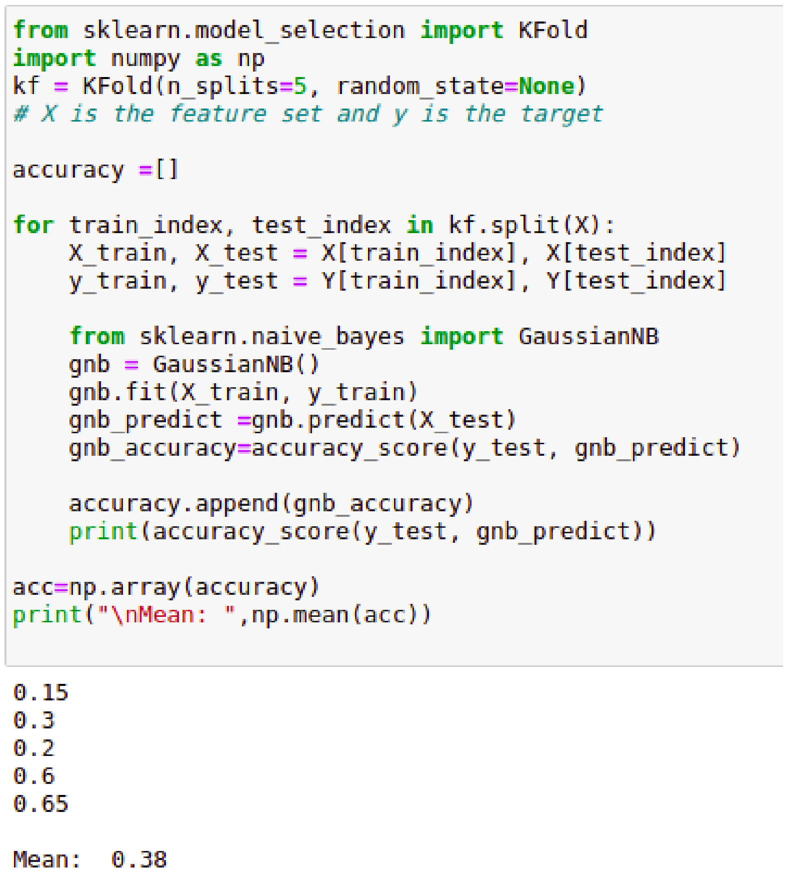


Naive bayes

```
]: from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train, y_train)
gnb_predict =gnb.predict(X_test)
gnb_accuracy=accuracy_score(y_test, gnb_predict)
print("Accuracy:",gnb_accuracy)

accuracies.append(gnb_accuracy)
modelName.append("Naive Bayes")
```

```
Accuracy: 0.3
```

On applying k fold cross validation with the number of splits as 5, we receive a mean accuracy of **38% .**

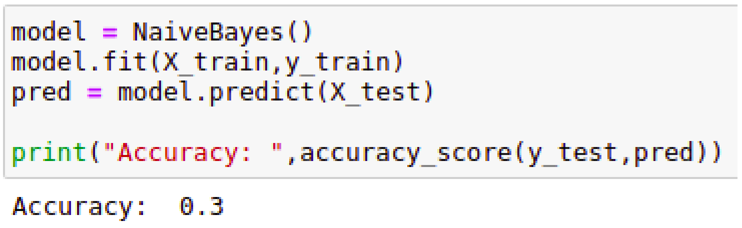


```
from sklearn.model_selection import KFold
import numpy as np
kf = KFold(n_splits=5, random_state=None)
# X is the feature set and y is the target

accuracy =[]

for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = Y[train_index], Y[test_index]

    from sklearn.naive_bayes import GaussianNB
    gnb = GaussianNB()
    gnb.fit(X_train, y_train)
    gnb_predict =gnb.predict(X_test)
    gnb_accuracy=accuracy_score(y_test, gnb_predict)

    accuracy.append(gnb_accuracy)
    print(accuracy_score(y_test, gnb_predict))

acc=np.array(accuracy)
print("\nMean: ",np.mean(acc))
```

```
0.15
0.3
0.2
0.6
0.65

Mean:  0.38
```

Naive Bayes Algorithm implementation from scratch

```
import math
class NaiveBayes():

   def fit(self, X, y):
       self.X = X
       self.y = y
       self.classes = np.unique(y)
       self.parameters = []
       for i, c in enumerate(self.classes):
           X_where_c = X[np.where(y == c)]
           self.parameters.append([])
           for col in X_where_c.T:
               parameters = {"mean": col.mean(), "var": col.var()}
               self.parameters[i].append(parameters)

   def likelihood(self, mean, var, x):
       m = 0.01
       gaussian = (1.0 / math.sqrt(2.0 * math.pi * var +
m))*(math.exp(-(math.pow(x - mean, 2) / (2 * var + m))))
       return gaussian

   def prior(self, target):
       return np.mean(self.y == target)

   def predict(self, X):
       y_pred = []
       for j in X:
           posteriors = []
           for i, c in enumerate(self.classes):
               posterior = self.prior(c)
               for feature_value, params in zip(j, self.parameters[i]):
                   likelihood = self.likelihood(params["mean"],
params["var"], feature_value)
                   posterior *= likelihood
               posteriors.append(posterior)
           y_pred.append(self.classes[np.argmax(posteriors)])
       return y_pred
```

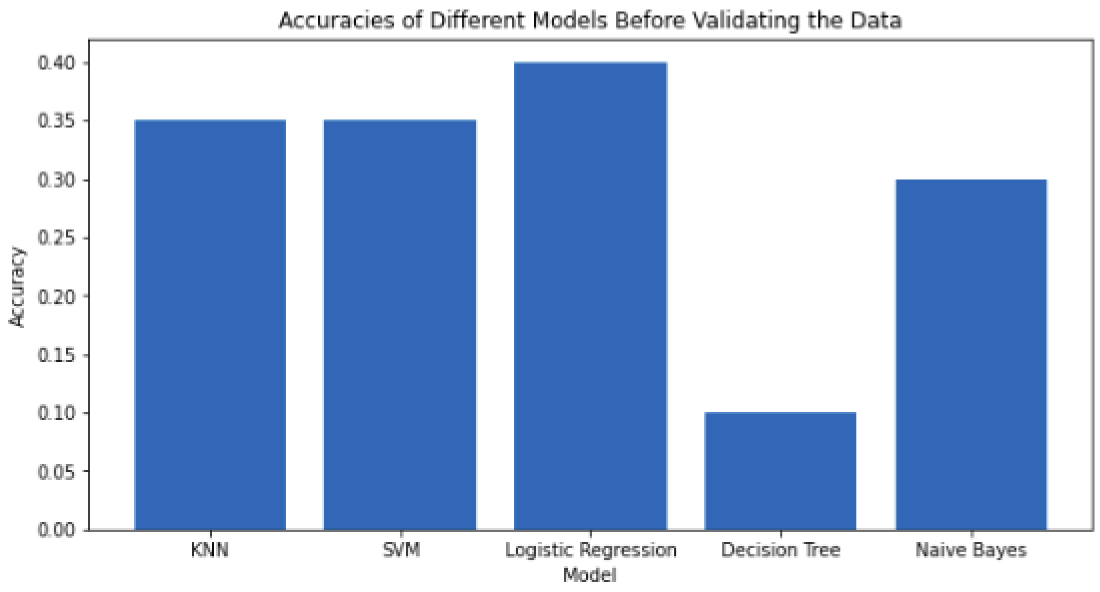


```
model = NaiveBayes()
model.fit(X_train,y_train)
pred = model.predict(X_test)

print("Accuracy: ",accuracy_score(y_test,pred))
```

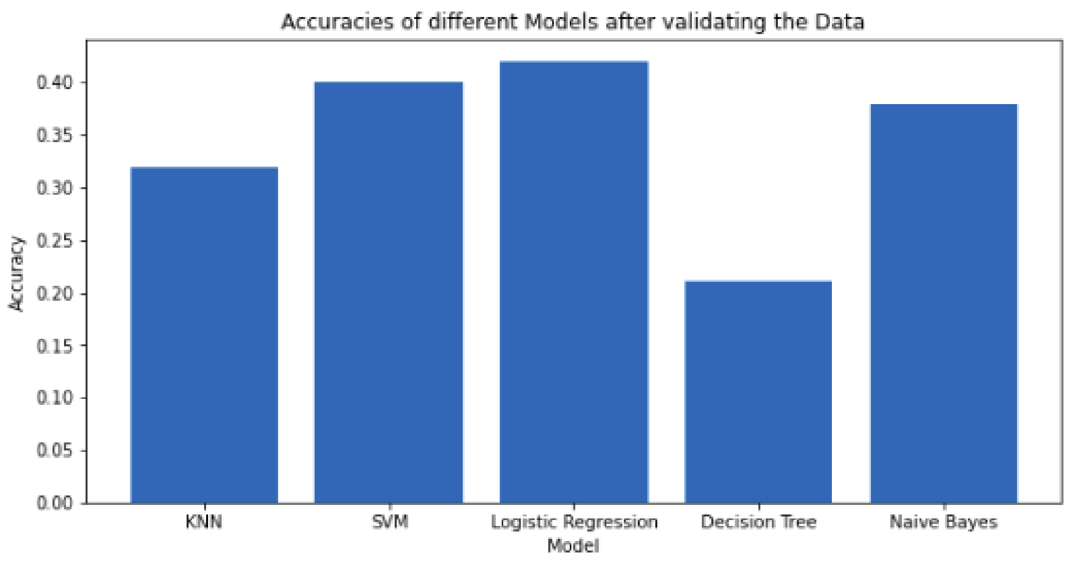
```
Accuracy:  0.3
```

For the above scratch implementation , we receive a **30%** accuracy .

Results



On performing the learning and training on different models before and after validating the data, we can observe some varied results. Before performing validation, we can see that we receive the highest accuracy in the Logistic regression model and least in the Decision tree. We can also observe that KNN and SVM have the same accuracy .



On the other hand we can observe that , after applying cross validation there are few changes in the trends. We can observe that we still receive the highest accuracy in the Logistic regression model and least in the Decision tree. But here SVM and KNN are not having similar accuracies , SVM has the second highest accuracy after Logistic regression.