**DATA MINING**

Harokopio University of Athens

Kokkalis Konstantinos – it22047

Professor: Varlamis Iraklis

Contents

[**A.DATA PRE-PROCCESSING** 3](#_Toc156753124)

[**STEPS** 3](#_Toc156753125)

[**Data frame Initialization** 5](#_Toc156753126)

[**Remove redundant columns** 6](#_Toc156753127)

[**Missing values** 9](#_Toc156753128)

[**Data formatting** 13](#_Toc156753129)

[**One-Hot encoding** 14](#_Toc156753130)

[**Delete duplicates** 17](#_Toc156753131)

[**Extra functions** 18](#_Toc156753132)

[**Result** 20](#_Toc156753133)

[**B1.CLASSIFICATION** 21](#_Toc156753134)

[**STEPS** 23](#_Toc156753135)

[**Load and Prepare Data** 23](#_Toc156753136)

[**Model Training** 25](#_Toc156753137)

[**B2.PREDICTIONS** 28](#_Toc156753138)

[**C.CLUSTERING** 29](#_Toc156753139)

[**STEPS** 29](#_Toc156753140)

[**Load Dataset and Scale** 30](#_Toc156753141)

[**Clustering** 31](#_Toc156753142)

[**Plot Results** 32](#_Toc156753143)

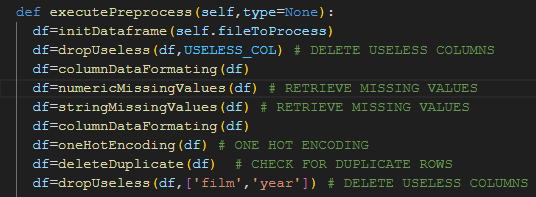
[**CLOSING** 46](#_Toc156753144)

[**.readme** 46](#_Toc156753145)

[**Links** 47](#_Toc156753146)

# **A.DATA PRE-PROCCESSING**

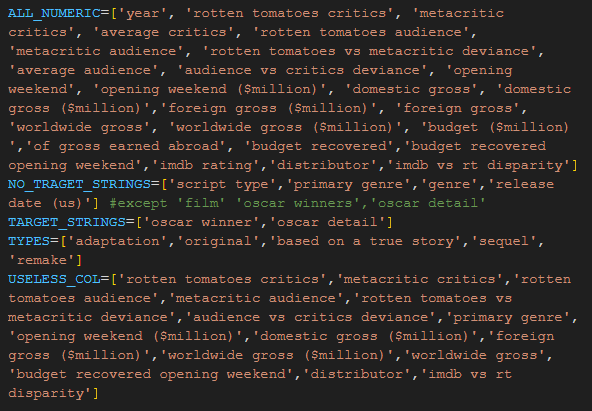
## **STEPS**



1. Data frame Initialization.
2. Remove redundant columns.
3. Handle missing values.
4. Data formatting.
5. One Hot Encoding.
6. Delete duplicates.

**Note:**

At the beginning we save in some arrays the numeric columns, the redundant columns, the target strings, the no target strings as long as the script types. The reason we do that is to handle easier decision changes in the future, such as which columns we are not going to use in our classification model.



**ALL\_NUMERICS** = all the columns that they contain numeric values.

**TARGET\_STRINGS**=all the string columns that have Oscar details.

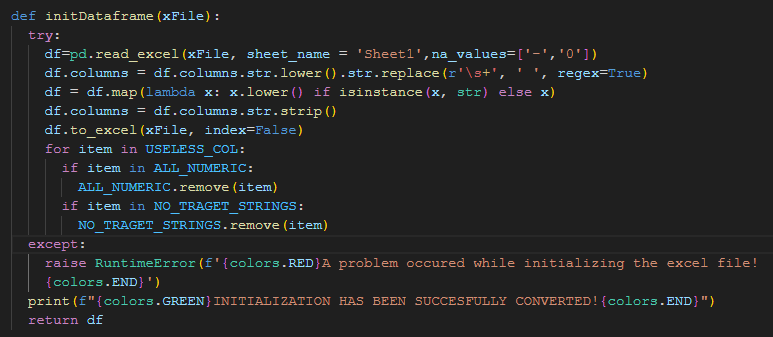
**NO\_TARGET\_STRINGS**=all the other string columns except from target strings.

**USELESS\_COLUMNS**=all the columns that we do not need for the classification model.

### **Data frame Initialization**

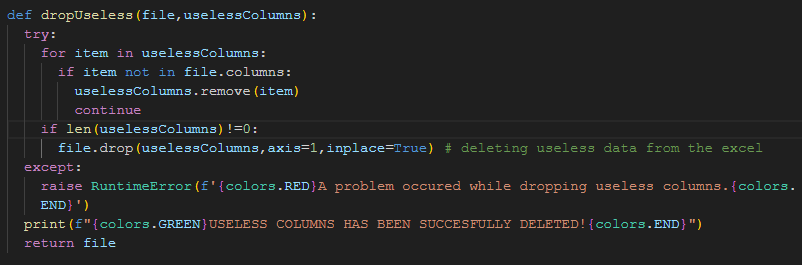
1. We say the to the data frame to replace **‘-’** and **‘0’** with Not a Number values so we can handle them later.
2. We remove redundant spaces from column names and we transform them to lower case so we can handle them easier. We also do the same for the whole dataset.
3. We update the dataset.
4. We update the ALL\_NUMERIC and the NO\_TRAGET\_STRINGS table according to the USELESS COLUMNS array.
5. Return the updated dataset.





### **Remove redundant columns**





USELESS\_COL=['rotten tomatoes critics','metacritic critics','rotten tomatoes audience','metacritic audience','rotten tomatoes vs metacritic deviance','audience vs critics deviance','primary genre','opening weekend ($million)','domestic gross ($million)','foreign gross ($million)','worldwide gross ($million)','worldwide gross','budget recovered opening weekend','distributor','imdb vs rt disparity']

*Why each of them has been deleted?*

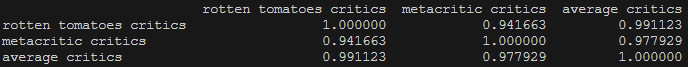
**distributor, imdb vs rt disparity, primary genre**: We dropped these columns because we had no, or a few data comparing to the size of the tuples, which means they would not give any information and they would not help the training model.

**opening weekend ($million), domestic gross ($million), foreign gross ($million), worldwide gross ($million)**: We dropped these columns because we have already the information of each, in other columns with a bigger precision (opening weekend, domestic gross, foreign gross).

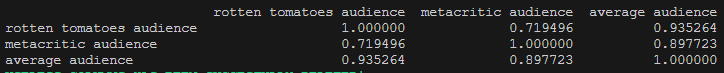
**rotten tomatoes vs metacritic deviance, audience vs critics deviance**: We dropped these columns because they do not provide any useful information.

**worldwide gross**: We dropped this column because worldwide gross is a result of domestic and foreign gross columns that we already kept, so there is no point to keep those two features and at the same time the sum of those two features.

**rotten tomatoes critics, metacrtic critics, rotten tomatoes audience, metacritic audience**: Here we checked the corelation of these attributes and the averages columns to decide if we are going to keep the averages or each one individually. The result was to keep the averages and drop the individuals. The decision was made according to these results:

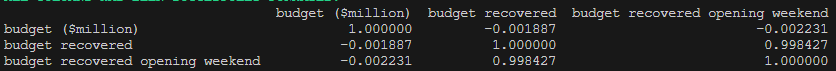


We can notice in the previous picture that all the corelations are very close to 1 and they also very similar between them. That means that we can keep only one of them. In that case we will keep average critics because it is also the mean of the other two features.



We can notice here that all the corelations are very close to 1 and they also very similar between them. That means that we can keep only one of them. In that case we will keep average audience because it is also the mean of the other two features.

**budget recovered opening weekend**: Here we checked again the corelation of this attribute, budget recovered and budget ($million). The decision was made according to these results:



We can notice here that all the corelations are very close to 0 (so there is no corelation) and they also very similar between them. We can only notice that budget recovered and budget recovered opening weekend they have corelation close to 1 so one of them could be deleted. So, we chose to keep budget recovered feature.

Code for getting corelation:

subset = df[['budget ($million)', 'budget recovered', 'budget recovered opening weekend']]

    # subset = df[['rotten tomatoes critics', 'metacritic critics','average critics']]

    # subset = df[['rotten tomatoes audience','metacritic audience','average audience']]

    getCorrelation(subset)

def getCorrelation(items):

  correlation\_matrix = items.corr()

  print(correlation\_matrix)

**Note:**

**film, year**: We delete film and year because the name of the film is a string that will not give helpful information to the classification model, on the other side it will confuse it. Year also does not provide any useful information, but we can use both to retrieve other data. So, we do not drop them yet.

**script type, genre, Oscar detail**: We will one-hot encode them and some of them so we will delete the main feature of one-hot encoding after this process.

### **Missing values**

df=numericMissingValues(df) # RETRIEVE MISSING VALUES

df=stringMissingValues(df) # RETRIEVE MISSING VALUES

def numericMissingValues(file): # replacing ',' and missing values with the mean of the year it belongs to

  try:

    for item in ALL\_NUMERIC:

      if item not in file.columns:

        ALL\_NUMERIC.remove(item)

        continue

    if len(ALL\_NUMERIC)==0:

      print(f'{colors.GREEN}NO NUMERIC COLUMNS TO BE PROCESSED.SUCCESFULLY COMPLETED{colors.END}')

      return file

  # file=externalIMDb(file) # IMDb

    for element in ALL\_NUMERIC:

      for index, row in file.iterrows():

        year = row['year']

        mean = file[file['year'] == year][element].mean()

        file[element] = pd.to\_numeric(file[element], errors='coerce')

        if pd.isna(row[element]):

          if pd.isna(mean):

            file[element]=file[element].ffill()

            continue

          file.loc[index, element] = mean

      file[element].bfill(inplace=True)

  except:

    print(element,index,row,mean)

    raise ValueError(f'{colors.RED}A problem occured while processing numeric missing values{colors.END}')

  print(f"{colors.GREEN}NUMERIC MISSING VALUES HAS BEEN SUCCESFULLY RESTORED!{colors.END}")

  return file

For numeric missing values we get **external knowledge** only for imdb rating feature. For all the others we take the mean value of each numeric feature according to year the film released.

(e.g Missing value at foreign gross. Year=2007, we complete the missing value with the foreign gross mean of the films released in 2007).

In case we cannot retrieve external knowledge or calculate the mean for a missing value we are doing a ffil() and if this is also not possible we do bfill().

def stringMissingValues(file):

  try:

    for item in NO\_TRAGET\_STRINGS:

      if item not in file.columns:

        NO\_TRAGET\_STRINGS.remove(item)

        print(f"{item} column removed from the array because it doesn't exist in the dataset")

        continue

      if len(ALL\_NUMERIC)==0:

        print(f'{colors.GREEN}NO STRING COLUMNS TO BE PROCESSED.SUCCESFULLY COMPLETED{colors.END}')

        return file

    file=externalGenre(file)

    for index, row in file.iterrows():

        if 'oscar winners' in file.columns:

          if (pd.isna(row['oscar winners'])):

            file.loc[index, 'oscar winners'] = 0

          else:

            file.loc[index, 'oscar winners'] = 1

    for j in NO\_TRAGET\_STRINGS:

      if file[j].isnull().any():

        file[j]=file[j].ffill()

      else:

        file[j]=file[j].bfill()

  except:

    raise RuntimeError(f'{colors.RED}A problem occured while processing string missing values{colors.END}')

  print(f"{colors.GREEN}OTHER MISSING VALUES HAS BEEN SUCCESFULLY RESTORED!{colors.END}")

  return file

In string missing values we get **external knowledge** only for genre.

If any other value is missing we are doing a ffill() and if this is also not possible we do bfill().

#### **External knowledge**

In this part of external knowledge we just use a python library of IMDb and we get the genre.If we cannot get the genre then we complete the value as NaN.

def externalGenre(file):

  ia=IMDb()

  try:

    for index,row in file.iterrows():

      if pd.isna(row['genre']):

        movies = ia.search\_movie(row['film'])

        if movies:

            movie = ia.get\_movie(movies[0].movieID)

            genre=",".join(movie['genres']).lower()

        else:

            print("Movie not found.")

            genre=np.nan

        file.at[index, 'genre'] = genre

  except:

      raise RuntimeError(f'{colors.RED}A problem occured while receiving external knowledge in string missing values{colors.END}')

  return file

In this part of external knowledge we just use a python library of IMDb and we get the imdb rating. If we cannot get the rating then we complete the value as NaN and they will be replaced later with the mean, ffill() or bfill()

def externalIMDb(file):

  ia = IMDb()

  i=2

  for movieTitle in file['film']:

    movies = ia.search\_movie(f"{movieTitle}")

    try:

      if movies:

          movie = ia.get\_movie(movies[0].movieID)

          print(f"Title: {movie['title']}")

          year=movie['year']

          rating=movie['rating']

      else:

          print("Movie not found.")

          continue

      print(i," ",rating)

      if not rating:

        file.at[i-2, 'imdb rating'] = np.nan

        continue

      else:

        file.at[i-2, 'imdb rating'] = rating

        file.at[i-2,'year']=year

    except:

      file.at[i-2, 'imdb rating'] = np.nan

      continue

    i=i+1

  print(f'{colors.GREEN}EXTERNAL KNOWLEDGE \'IMDb\' HAS BEEN SUCCESFULLY ADDED{colors.END}')

  # file.to\_excel("clone.xlsx")

  return file

**Note:**

1.This process has already been executed and we copied the data to another file (**moviesUpdated.xlsx**), because the process to fetch data for almost 1400 rows takes a lot of time.

2.Textblob library is used in the dataset for the genre feature, because a lot of genres are misspelled.

### **Data formatting**

df=columnDataFormating(df)

After receiving all the new values and we have already dropped the most of the columns we do not need, we have to format all the data to be the same so the model can work correctly.

def columnDataFormating(file):

  try:

    #  Convert each value of the df that is ending with the '%' to decimal (divide by 100)

    file = file.map(lambda val: float(val.rstrip('%')) / 100 if isinstance(val, str) and val.endswith('%') else val)

  except:

    raise RuntimeError(f'{colors.RED}A problem occured while converting precentages to decimal.{colors.END}')

  try:

    file[ALL\_NUMERIC] = file[ALL\_NUMERIC].replace(',', '', regex=True).apply(pd.to\_numeric, errors='coerce')

    file['budget ($million)'] = file['budget ($million)'] \* 1000000

    # file[NO\_TRAGET\_STRINGS] = file[NO\_TRAGET\_STRINGS].replace(',', '', regex=True)

    file['genre'] = file['genre'].str.replace(',', ' ').str.replace('.', ' ').str.replace('\s+', ' ', regex=True).str.strip()

    # file['oscar detail'] = file['oscar detail'].str.extract(r'([^\(]+)')

    file['oscar detail'] = file['oscar detail'].str.split('(', n=1).str[0].str.strip()

  except:

    raise RuntimeError(f'{colors.RED}A problem occured while replacing charachters.{colors.END}')

  print(f"{colors.GREEN}ALL COLUMNS HAS BEEN SUCCESFULLY FORMATED!{colors.END}")

  return file

### **One-Hot encoding**

    df=oneHotEncoding(df) # ONE HOT ENCODING

We do have nominal features that we need to represent them with a way to help the model to work better. Those are script type, oscar detail, release date (us), genre

# SCRIPT TYPE

  if 'script type' in file.columns:

    try:

      one\_hot\_encoded = pd.get\_dummies(file['script type'].apply(lambda x: next((t for t in TYPES if str(x).startswith(t)), None))).astype(int)

      file = pd.concat([file, one\_hot\_encoded], axis=1)

      file=dropUseless(file,['script type'])

    except:

      raise RuntimeError(f'{colors.RED}A problem occured while one-hot encoding script-type{colors.END}')

# OSCAR DETAILS

  if 'oscar detail' in file.columns:

    try:

      one\_hot\_encoded = file['oscar detail'].str.get\_dummies(', ').astype(int)

      file = pd.concat([file, one\_hot\_encoded], axis=1)

      file=dropUseless(file,['oscar detail'])

    except:

      raise RuntimeError(f'{colors.RED}A problem occured while one-hot encoding oscar details{colors.END}')

In the previous ones we did directly normal one-hot encoding but genre and release date (us) need a special treatment.

As far as the genre is concerned there were a lot genres with wrong spelling mistakes or the same genre with another word , for example ‘thriller’ and ‘thrill’. All of those had to be handled before one-hot encoding.

# GENRE

  if 'genre' in file.columns:

    try:

      file['genre']=file['genre'].str.lower()

      genres=set()

      correctGenre=[]

      for item in file['genre']:

        for word in item.split():

          # correctedWords=[]

          correctedWord=str(Word(word).correct().lower())

          # print(correctedWord)

          correctGenre.append(correctedWord)

      genres.update(correctGenre)

      # print("c:",genres)

      words\_to\_remove = set()

      for word1 in genres:

            for word2 in genres:

                if word1 != word2 and len(word1) >3 and len(word2) > 3:

                    common\_substrings = set([word1[i:i+5] for i in range(len(word1)-4) if word1[i:i+5] in word2])

                    if common\_substrings:

                        shorter\_word = word1 if len(word1) < len(word2) else word2

                        words\_to\_remove.add(shorter\_word)

      genres.difference\_update(words\_to\_remove)

      # print(genres)

      file['genre'] = file['genre'].apply(lambda cell: ' '.join(

        [next((word\_set\_word for word\_set\_word in genres if word\_set\_word[:3] == word[:3]), word) for word in cell.split()]

    ))

      print('Set:::',genres)

      for genre in genres:

          file[genre] = file['genre'].apply(lambda x: 1 if genre in x.split() else 0)

      file=dropUseless(file,['genre'])

    except:

      raise RuntimeError(f'{colors.RED}A problem occured while one-hot encoding genres{colors.END}')

As far as the date is concerned before one hot encoding we played with the format and we kept only the released month instead of the whole day, month and year, because we believed all of these information will not give us any advantage.

# DATE

  if 'release date (us)' in file.columns:

    try:

      file['release date (us)'] = pd.to\_datetime(file['release date (us)'],format='mixed')

      # Extract month and day

      file['release date (us)'] = file['release date (us)'].dt.strftime('%m').astype(int)

      monthMapping = {

        1: 'january',

        2: 'february',

        3: 'march',

        4: 'april',

        5: 'may',

        6: 'june',

        7: 'july',

        8: 'august',

        9: 'september',

        10: 'october',

        11: 'november',

        12: 'december'

    }

      # Nominalize the 'release date (us)' column

      file['release date (us)'] = file['release date (us)'].map(monthMapping)

      one\_hot\_encoded = pd.get\_dummies(file['release date (us)'], prefix='').astype(int)

      file = pd.concat([file, one\_hot\_encoded], axis=1)

      file=dropUseless(file,['release date (us)'])

    except:

      raise RuntimeError(f'{colors.RED}A problem occured while one-hot encoding dates{colors.END}')

**Note:**

Of course, after one-hot encoding we drop each feature that got one-hot encoded.

### **Delete duplicates**

df=deleteDuplicate(df)  # CHECK FOR DUPLICATE ROWS

after all the process we have to check if we have duplicate data.

def deleteDuplicate(file):

  try:

    if (file.duplicated().sum() != 0) or (not file[file.duplicated(subset=['film'])].empty):

      print(f'The dataset contains {(file.duplicated(subset=["film"])).sum()} duplicate films that need to be removed.')

      print(f'The dataset contains {file.duplicated().sum()} duplicate rows that need to be removed.')

      file.drop\_duplicates(inplace=True)

      file = file.drop\_duplicates(subset=['film'],keep='first')

  except:

    raise RuntimeError(f'{colors.RED}A problem occured while deleting duplicates{colors.END}')

  print(f"{colors.GREEN}DUPLICATE ROWS HAVE BEEN SUCCESFULLY DELETED!{colors.END}")

  return file

we check the whole subset for duplicates, but also, we check the subset film for duplicate film names. If a film has a duplicate name, then we keep the data of the film we first found in our dataset.

### **Extra functions**

We also created some functions for the preprocessing such as scaling and normalization because we do not know yet what classification model we are going to use and what are its prerequires for it to work.

def scaling(file):

  for item in ALL\_NUMERIC:

    if item not in ALL\_NUMERIC:

      ALL\_NUMERIC.remove(item)

      continue

  if len(ALL\_NUMERIC)!=0:

    try:

      scaler = StandardScaler()

      file[ALL\_NUMERIC] = scaler.fit\_transform(file[ALL\_NUMERIC])

    except:

      raise ValueError(f'{colors.RED}A problem occured while scaling values{colors.END}')

  print(f"{colors.GREEN}SCALING HAS BEEN SUCCESFULLY COMPLETED!{colors.END}")

  return file

def normalization(file):

  for item in ALL\_NUMERIC:

    if item not in ALL\_NUMERIC:

      ALL\_NUMERIC.remove(item)

      continue

  if len(ALL\_NUMERIC)!=0:

    try:

      columns\_to\_normalize = file[ALL\_NUMERIC]

      scaler = MinMaxScaler()

      normalized\_columns = pd.DataFrame(scaler.fit\_transform(columns\_to\_normalize), columns=ALL\_NUMERIC)

      file[ALL\_NUMERIC] = normalized\_columns

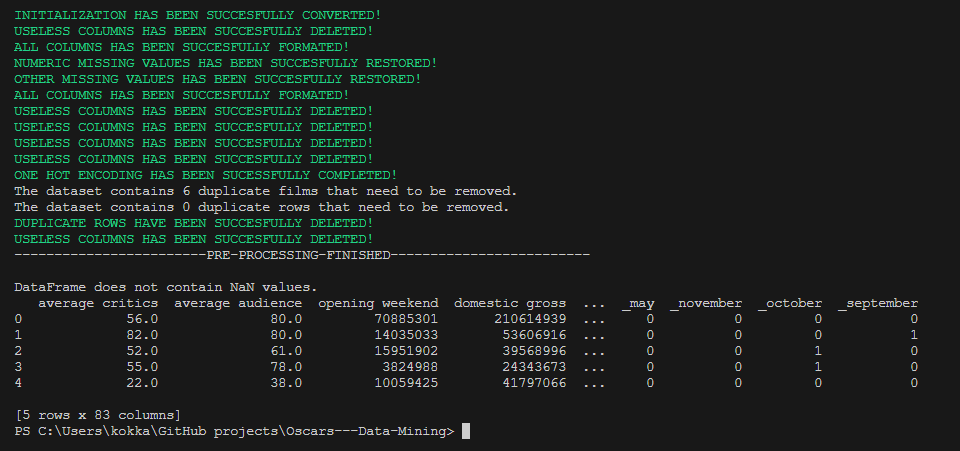
    except:

      raise ValueError(f'{colors.RED}A problem occured while normalising values{colors.END}')

  print(f"{colors.GREEN}NORMALISING HAS BEEN SUCCESFULLY COMPLETED!{colors.END}")

  return file

### **Result**



**Important Note:**

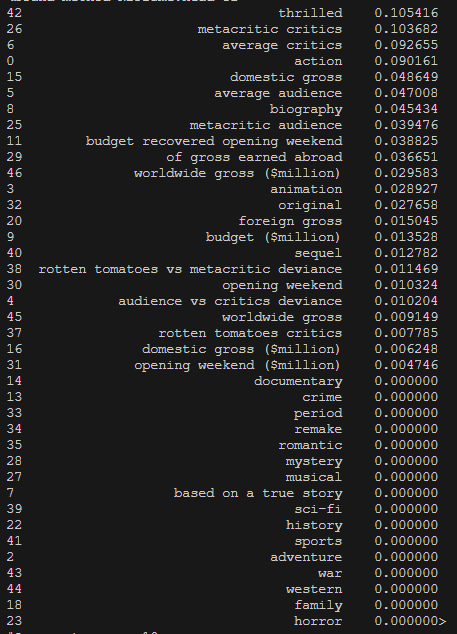
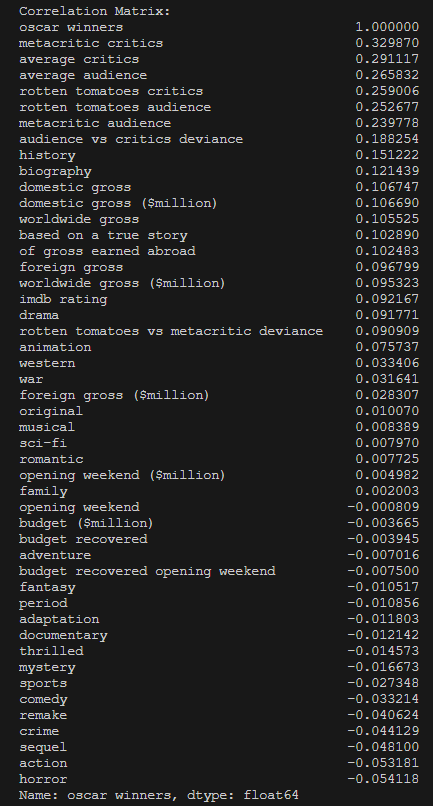
The features that have been selected to be used they are not the final as long as the classification model might not give us the best results. So modifications to USELESS\_COL might happen.

# **B1.CLASSIFICATION**

Before we start explaining about the classification process is important to mention that the features we use now are different. That is because with the previous features we could not receive very good results so the model was overfitting or it was not giving any results. The way we pre process the data is exactly the same. The only thing that changes is the columns we use.

The way we decided which columns to drop since we were not getting good results was running the model and checking at the same time the correlation and feature importance (after classification) of each feature with the target column ‘oscar winners’. Features with negative or very low correlation were deleted.

By droping these columns as a first step we can make easier our choice on which features we should still drop.  
USELESS\_COL=['id','imdb vs rt disparity','oscar detail','distributor','primary genre','release date (us)']  
**Correlation Matrix: Feature Importance:**

****

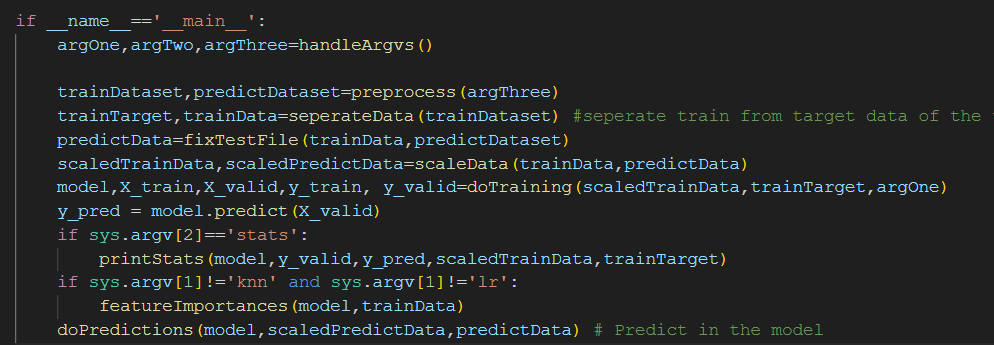
So according to the results above we ended up dropping the following features

New Dropped Columns:

USELESS\_COL=['id','imdb vs rt disparity','oscar detail','distributor','primary genre','genre','script type','release date (us)','opening weekend','budget recovered','budget ($million)','budget recovered opening weekend','domestic gross ($million)']

Now we can continue to classification.

## **STEPS**



1.Load Data

2.Prepare Data

3.Train Model

4.Predict

### 

### **Load and Prepare Data**

Before the training of the model and before making the predictions we have to be sure that there are no missing values in both training and test datasets as long as the two datasets must have exactly the same features. We also have to separate features from target in the training dataset and scale the data using MinMax scaler.

So first we preprocess both datasets

def preprocess(command='prepro'):

    if command=='prepro':

        dp=DataPreprocessor(TRAIN\_PATH,TRAIN\_PATH\_PROCESSED)

        df=DataPreprocessor(PREDICT\_PATH,PREDICT\_PATH\_PROCESSED)

        trainDataset=dp.executePreprocess()

        predictDataset=df.executePreprocess(predict=True)#options: predict=True/False

    else:

        # SAVING TIME-------------------

        trainDataset=pd.read\_excel(TRAIN\_PATH\_PROCESSED, sheet\_name = 'Sheet1')

        predictDataset=pd.read\_excel(PREDICT\_PATH\_PROCESSED, sheet\_name = 'Sheet1')

    print('all files has been succesfully preprocessed')

    return trainDataset,predictDataset

and right after we make sure that both datasets have the same columns

def fixTestFile(train\_file,test\_file):

  diff\_train\_columns = set(train\_file.columns) - set(test\_file.columns)

  diff\_test\_columns = set(test\_file.columns) - set(train\_file.columns)

  for test\_column in diff\_test\_columns:

      for train\_column in diff\_train\_columns:

          # Check if the columns share at least four letters

          if len(set(test\_column).intersection(train\_column)) >= 4:

              # Replace the column name in test\_file

              test\_file = test\_file.rename(columns={test\_column: train\_column})

              break  # Stop searching for similar columns once a match is found

  # Add missing columns to test\_file with values set to 0

  for train\_column in diff\_train\_columns:

      test\_file[train\_column] = 0

  test\_file = test\_file.loc[:, ~test\_file.columns.duplicated(keep='first')]

  test\_file = test\_file.drop(columns= set(test\_file.columns) - set(train\_file.columns))

  test\_file.to\_excel(PREDICT\_PATH\_PROCESSED)

  # print(train\_file.shape,test\_file.shape)

  return test\_file.sort\_index(axis=1)

separating features from target - and scaling

def seperateData(ds):

  if 'oscar winners' not in ds.columns:

    raise ValueError('Oscar winners not in the dataset')

  target=ds['oscar winners']

  data=ds.drop(columns='oscar winners')

  data=data.sort\_index(axis=1)

  return (target,data)

def scaleData(trainData,predictData):

    scaler=MinMaxScaler()

    scaledTrainData=scaler.fit\_transform(trainData)

    scaledPredictData=scaler.transform(predictData)

    return scaledTrainData,scaledPredictData

### **Model Training**

Now that the data are ready we can proceed to train the model by splitting our train dataset in train and test data.

def doTraining(scaledTrainData,trainTarget,modelName='knn'):

    X\_train, X\_valid, y\_train, y\_valid = train\_test\_split(scaledTrainData,trainTarget, test\_size=0.25,random\_state=42)

    if modelName=='rf':

        model = RandomForestClassifier(random\_state=42)

    elif modelName=='lr':

        model = LogisticRegression(max\_iter=1500, random\_state=42)

    elif modelName=='dtc':

        model = DecisionTreeClassifier(random\_state=42)

    elif modelName=='knn':

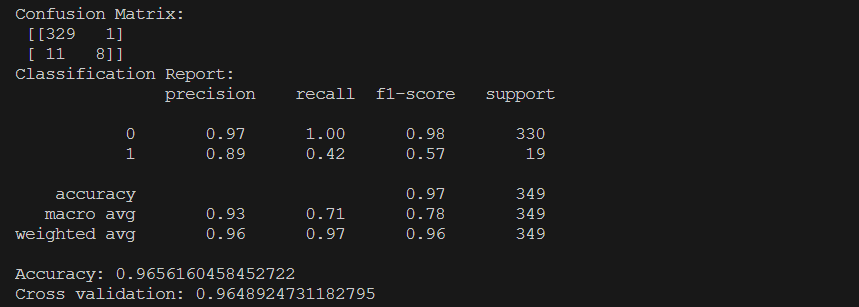
        model=KNeighborsClassifier(n\_neighbors=3)

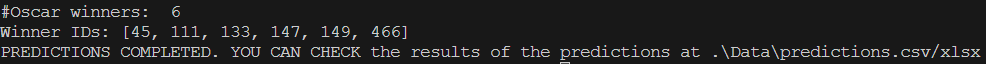
    model.fit(X\_train, y\_train)

    return model,X\_train,X\_valid, y\_train, y\_valid

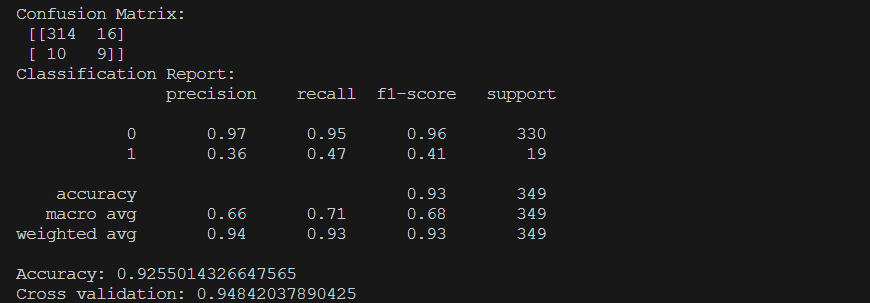
after the splitting we try different models to see which one works better for our dataset

#### **RANDOM FOREST**



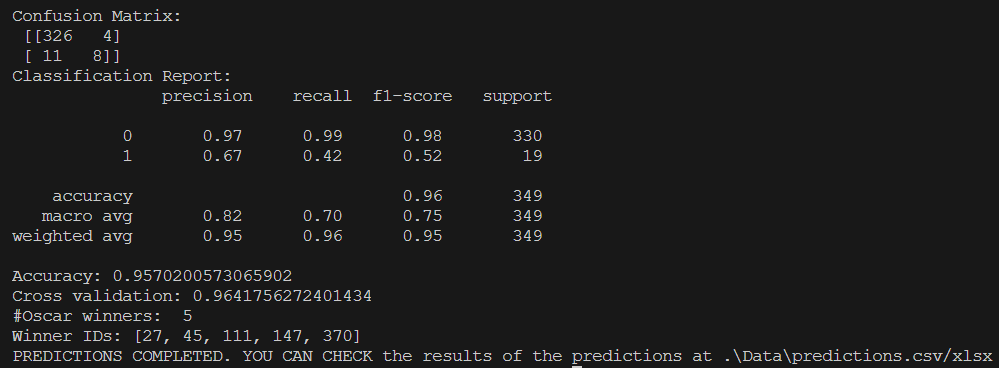


#### **DECISION TREE CLASSIFIER**

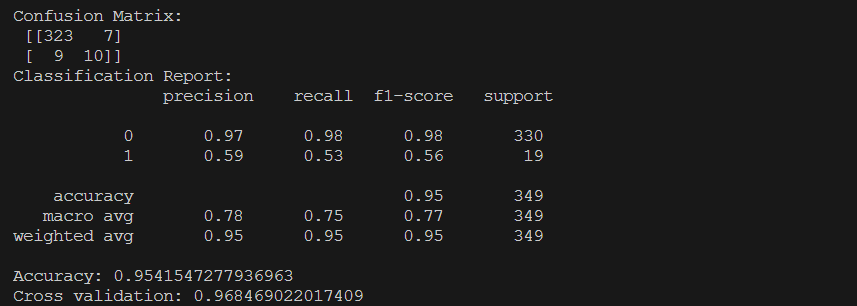




#### **K-NEAREST NEIGHBOURS**



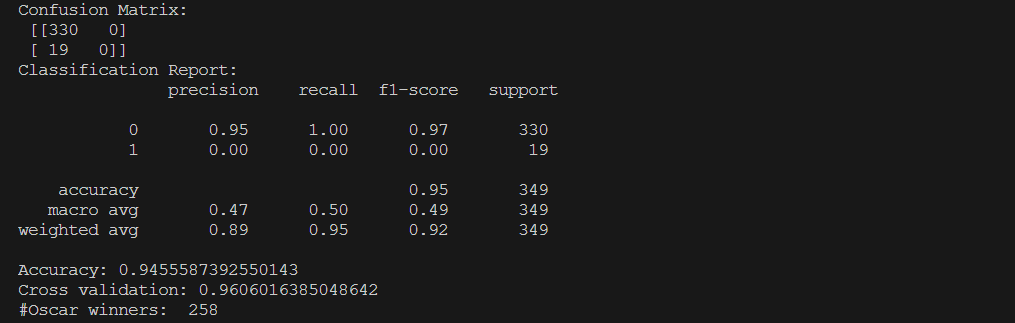
#### **GRADIENT BOOSTER**





#### **LOGISTIC REGRESSION**

Linear Regression and other algorithms like SVC are not suitable for our dataset as long as they are ill defined and they don’t return any useful results.



As we can see the True Negative is 0 which means 0 oscar winners and that is why it is ill-defined, so not suitable.

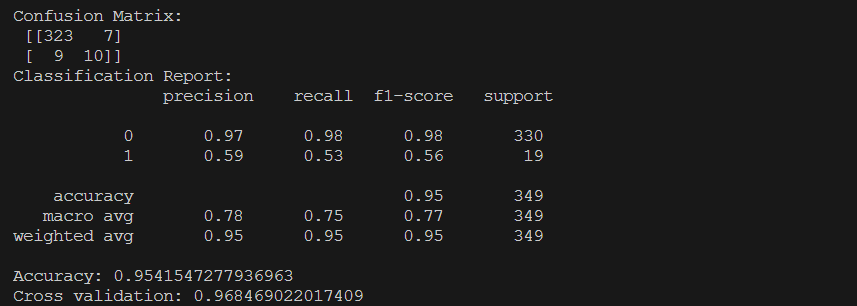
**Note:**

As long as the training models’ parameters is concerned, the models were tested in a variance of parameters. Different *random\_states* were giving vey similar results but huge importance had the splitting the dataset to train and test data. In our case training the model with *test\_size=0.25* had better results than training the model with other *test\_sizes*.

# **B2.PREDICTIONS**

In the models above we can see pretty similar results between some algorithms but at the end the model that is more useful for us is **gradient booster** and that is because it returns a high f-score a high recall, as long as a reasonable amount of Oscar winners. So **Gradient Booster** model is our selection to predict the oscar winners to unknown movies dataset.

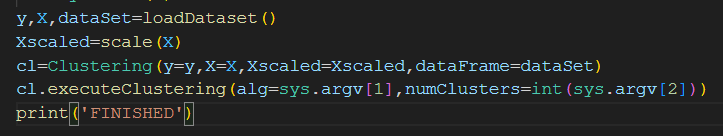
f-score=0.56 - recall=0,42 – Accuracy=95 – Cross Validation=96



The predictions are located in **‘Data/predictions.csv’** or **‘Data/predictions.xlsx’**

# **C.CLUSTERING**

## **STEPS**

1. Load Dataset
2. Scale
3. Cluster
4. Plot Results

### **Load Dataset and Scale**

def scale(X):

    if sys.argv[3] == 'ss':

        scaler = StandardScaler()

    elif sys.argv[3] == 'rs':

        scaler = RobustScaler()

    else:

        scaler = MinMaxScaler()

    Xscaled = scaler.fit\_transform(X)

    return Xscaled

def loadDataset():

    dataset = pd.read\_excel(TRAIN\_PATH\_PROCESSED, sheet\_name='Sheet1')

    y, X = separateData(dataset)

    print(type(X))

    return y,X,dataset

Before clustering is very important our data to be scaled. The type of scaling is very important for the results of our clustering.

### **Clustering**

def executeClustering(self,alg='KM',numClusters=2):

        # Perform PCA on the scaled data

        pca = PCA(n\_components=2)

        Xnew = pca.fit\_transform(self.Xscaled)

        print("PCA Components:")

        print(pca.components\_)

        y=pd.DataFrame(self.y)

        print('EXECUTING: ')

        if alg=='KM':

            clusteringAlgorithm = KMeans(n\_clusters=numClusters, n\_init='auto', random\_state=42)

            print('KMeans')

        elif alg=='HAC':

            clusteringAlgorithm = AgglomerativeClustering(n\_clusters=numClusters)

            print('HAC')

        elif alg=='DBSCAN':

            eps\_value = 0.5

            clusteringAlgorithm = DBSCAN(eps=eps\_value, min\_samples=numClusters)

            print('DBSCAN')

        else:

            clusteringAlgorithm = Birch(n\_clusters=numClusters)

            print('Birch')

        clusterLabels = clusteringAlgorithm.fit\_predict(Xnew)

        # Add cluster labels in the dataframe

        dfCluster = pd.DataFrame(clusterLabels, columns=['cluster'])

        dfPCA = pd.DataFrame(Xnew, columns=["PC1", "PC2"])

        dfClass = pd.DataFrame(self.y, columns=['oscar winners'])

        dfAllClusters = pd.concat([dfPCA, dfClass, dfCluster], axis=1)

        pc1Osc,pc2Osc=self.printStats(y,dfAllClusters,Xnew,self.X,clusteringAlgorithm,numClusters,pca,alg)

        self.plotDiagrams(dfAllClusters,Xnew,pc1Osc,pc2Osc,numClusters)

Before doing clustering another important step is to find the two most important principal components. After that we do clustering according to them and our target feature (oscar winners).

### **Plot Results**

def plotDiagrams(self,dfAllClusters,Xnew,pc1Osc,pc2Osc,numClusters):

        fig, axs = plt.subplots(1, 2, figsize=(15, 5))

        # Plot the silhouette score comparing to Fowlkes-Mallows score

        xs = range(2, 50)

        sils = []

        fms = []

        for i in xs:

            clusteringAlgorithm = AgglomerativeClustering(n\_clusters=i)

            clusterLabels = clusteringAlgorithm.fit\_predict(Xnew)

            sils.append(metrics.silhouette\_score(Xnew, clusterLabels, metric='euclidean'))

            fms.append(metrics.fowlkes\_mallows\_score(dfAllClusters['oscar winners'], clusterLabels))

        axs[0].plot(xs, sils)

        axs[0].plot(xs, fms)

        axs[0].set\_xlabel('Number of clusters (k)')

        axs[0].set\_ylabel('Score')

        axs[0].set\_title('Silhouette and Fowlkes-Mallows Scores')

        axs[0].legend(['Silhouette', 'Fowlkes-Mallows'])

        # Plot the Clusters PC1=x PC2=y

        for i in range(numClusters):

            clusterDF = dfAllClusters[dfAllClusters['cluster'] == i]

            axs[1].scatter(clusterDF['PC1'], clusterDF['PC2'], label=f'Cluster {i + 1}')

        axs[1].scatter(pc1Osc, pc2Osc, label=f'Oscars', color='red')

        axs[1].set\_xlabel("PC1")

        axs[1].set\_ylabel("PC2")

        axs[1].set\_title("Clusters in PC1-PC2 Space")

        axs[1].legend()

        plt.tight\_layout()

        plt.show()

    def printStats(self,y,dfAllClusters,Xnew,X,clusteringAlgorithm,numClusters,pca,alg):

        # Print clustering stats

        print("Confusion Matrix:")

        print(confusion\_matrix(dfAllClusters['oscar winners'], dfAllClusters['cluster']))

        print("Calinski-Harabasz Score:", metrics.calinski\_harabasz\_score(Xnew, dfAllClusters['cluster']))

        print("Silhouette Score:", metrics.silhouette\_score(Xnew, dfAllClusters['cluster'], metric='euclidean'))

        # Explore cluster characteristics (centroid values)

        if alg!='KM':

            centroids = dfAllClusters.groupby('cluster').mean().values

        else:

            centroids = clusteringAlgorithm.cluster\_centers\_

        for i in range(numClusters):

            print(f"Cluster {i + 1} Centroid Values:")

            print("PC1:", centroids[i, 0])

            print("PC2:", centroids[i, 1])

            print()

        featureImportancePC1 = pca.components\_[0]

        featureImportancePC2 = pca.components\_[1]

        print("Top features contributing to PC1:")

        topFeaturesPC1 = X.columns[np.argsort(featureImportancePC1)[::-1]][:5].tolist()

        print(topFeaturesPC1)

        print("\nTop features contributing to PC2:")

        topFeaturesPC2 = X.columns[np.argsort(featureImportancePC2)[::-1]][:5].tolist()

        print(topFeaturesPC2)

        # Loop through the clusters and print information for points with prediction equal to 1

        predictionOneIds = getRowsWithPredictionOne(y)

        pc1Osc = []

        pc2Osc = []

        for index, row in dfAllClusters.iterrows():

            pointId = index

            pc1Value = row['PC1']

            pc2Value = row['PC2']

            clusterLabel = row['cluster'] + 1

            if pointId in predictionOneIds:

                pc1Osc.append(pc1Value)

                pc2Osc.append(pc2Value)

        # centroids = clusteringAlgorithm.cluster\_centers\_

        # Display top features contributing to each cluster

        print("Top features contributing to each cluster:")

        for i in range(numClusters):

            print(f"Cluster {i + 1}:")

            topFeaturesIndices = np.argsort(centroids[i])[::-1][:7]

            topFeatures = X.columns[topFeaturesIndices].tolist()

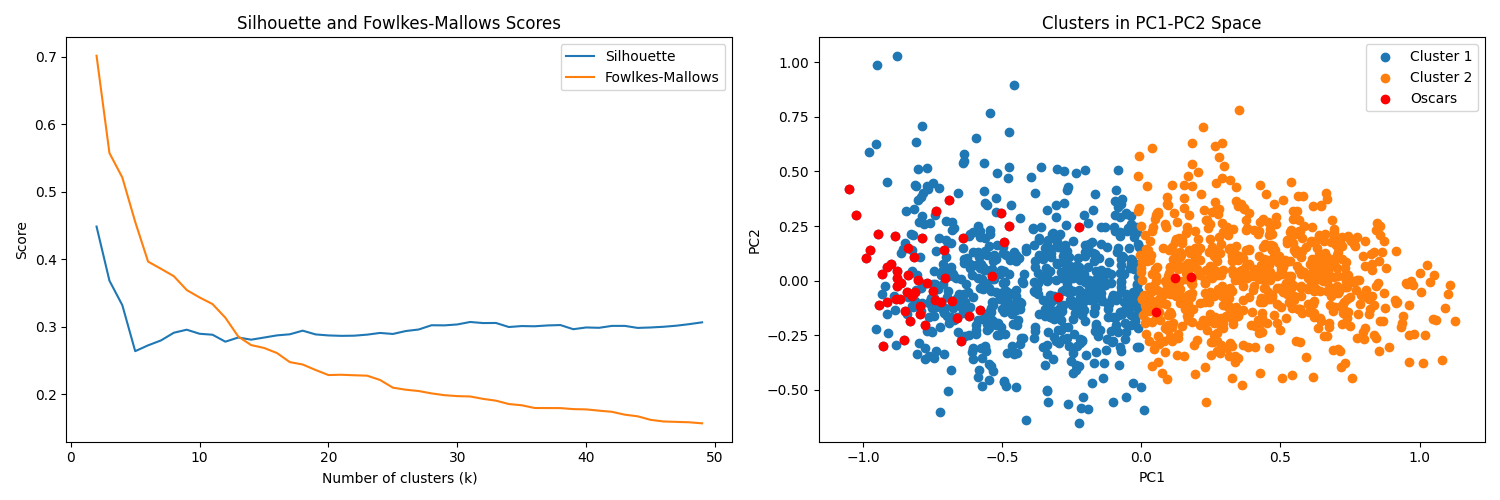
            print(topFeatures)

        return pc1Osc,pc2Osc

Different algorithms with different scaling will provide different results.

#### **KMEANS**

##### **MinMaxScaler**

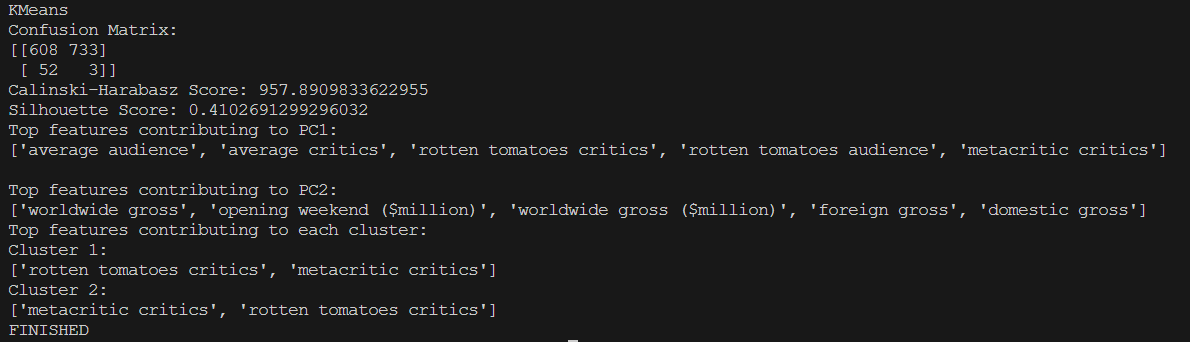
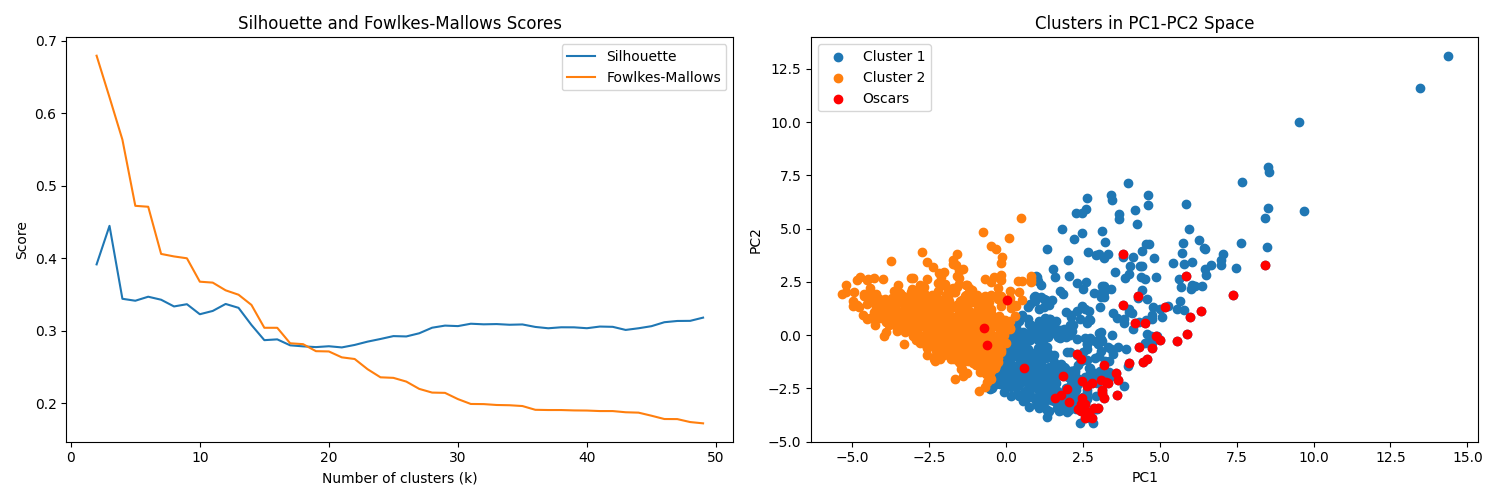


##### 

**Comment:**

Clustering here seems to work well as long as most of the oscar winners except from 3 are on the same cluster. Shilhouette score is good.

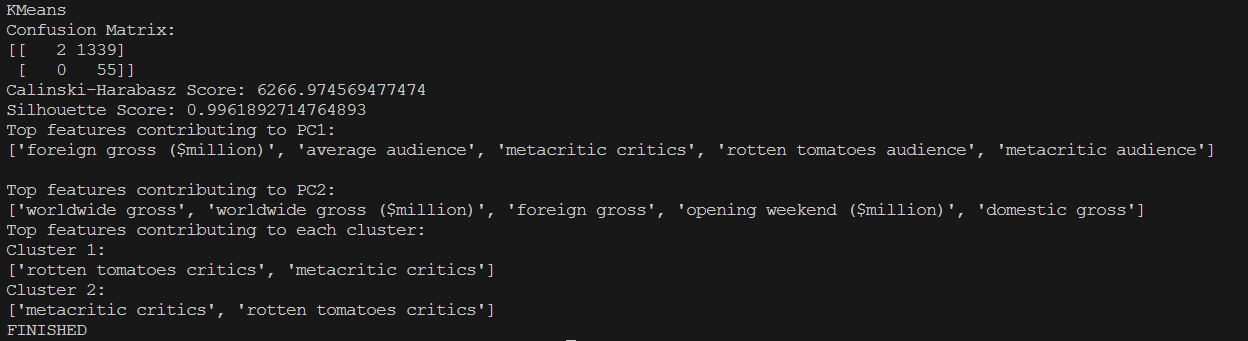
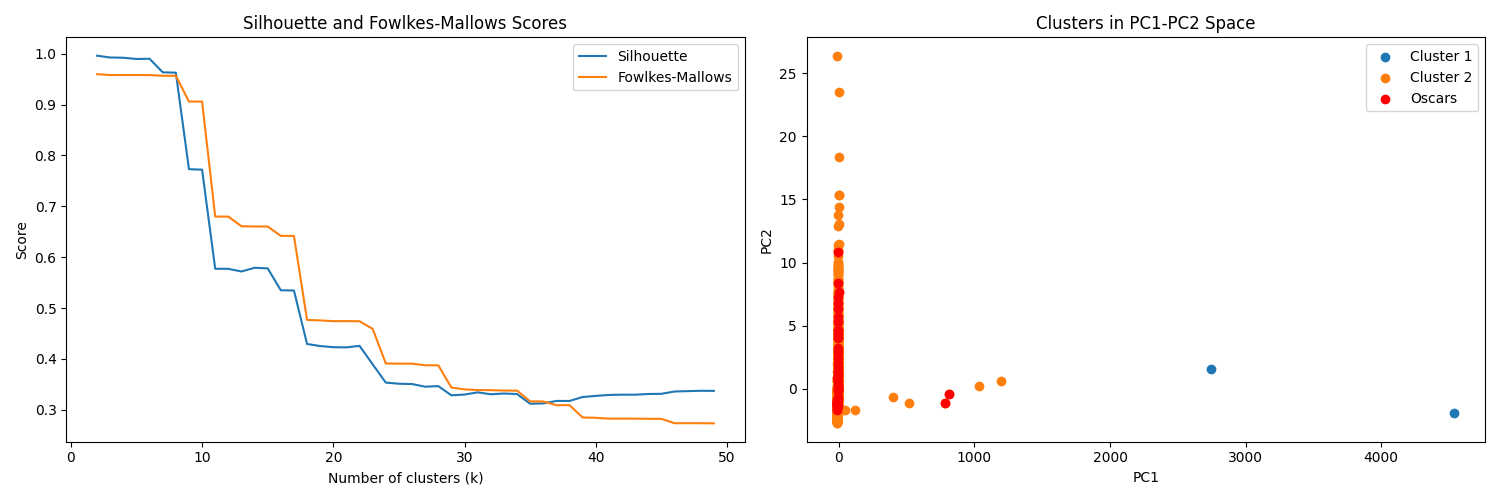
**StandardScaler**



**Comment:**

Clustering here seems to work also well as long as most of the oscar winners except from 3 are on the same cluster. Shilhouette score is good.

##### **RobustScaler**

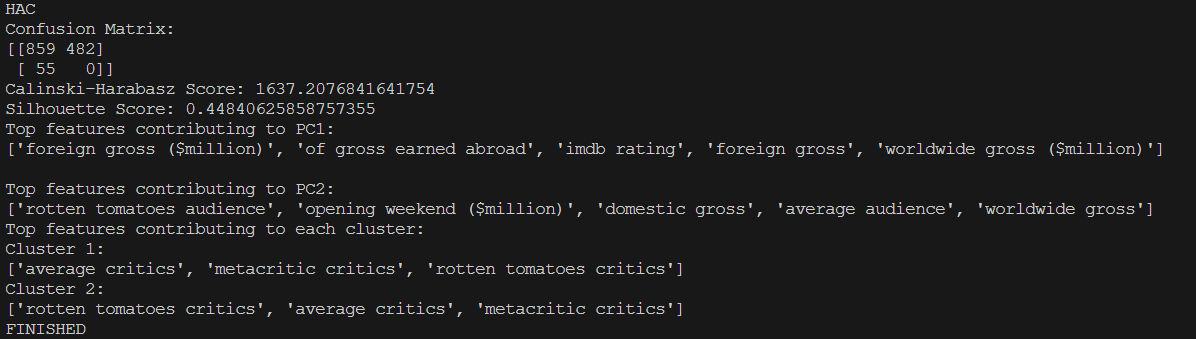
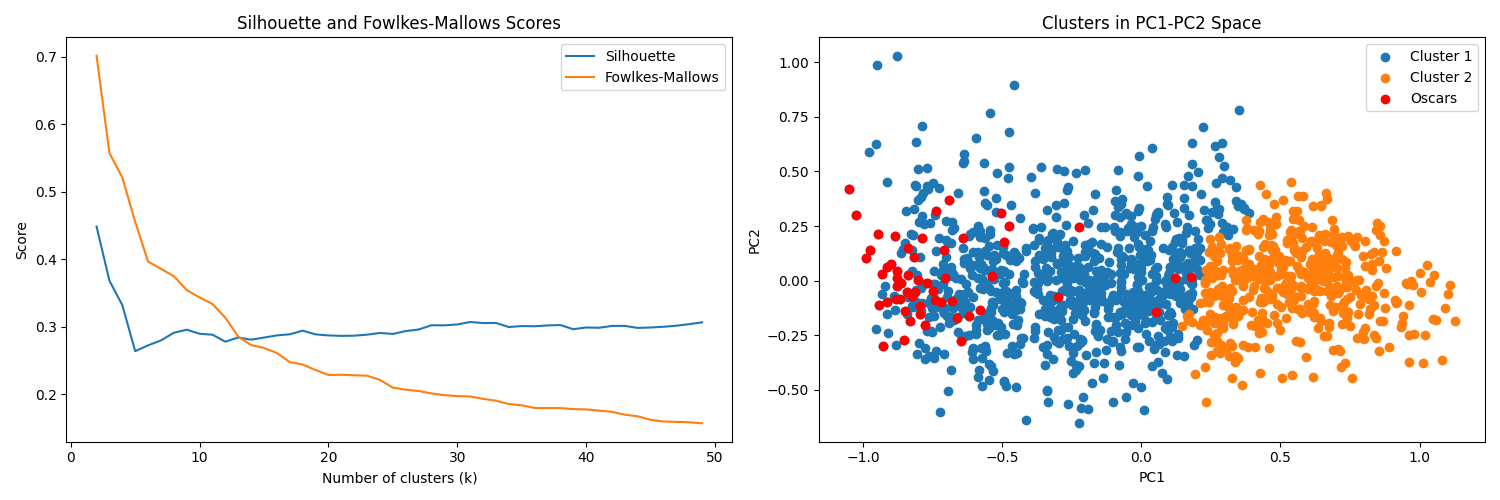


**Comment:**

Clustering here is not working well despite the fact that Silhouette score is high.

#### **AGGLOMERATIVE CLUSTERING (HAC)**

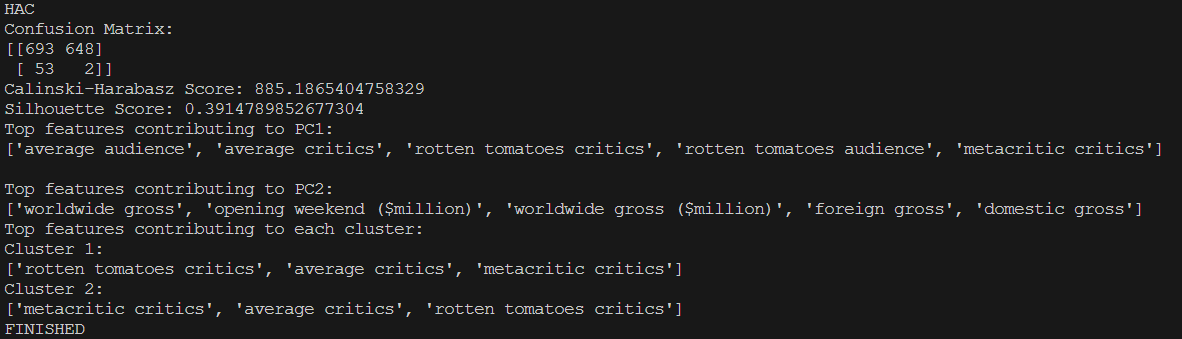
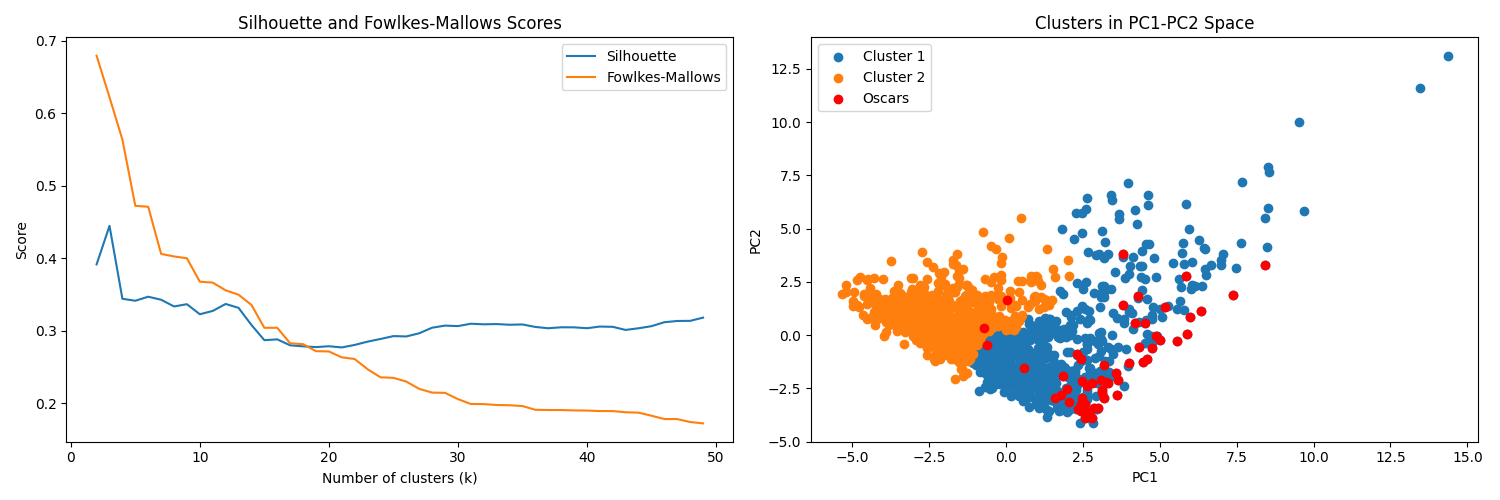
##### **MinMaxScaler**



**Comment:**

Clustering here seems to work even better as long as allof the oscar winners in the same cluster. Shilhouette score is good.

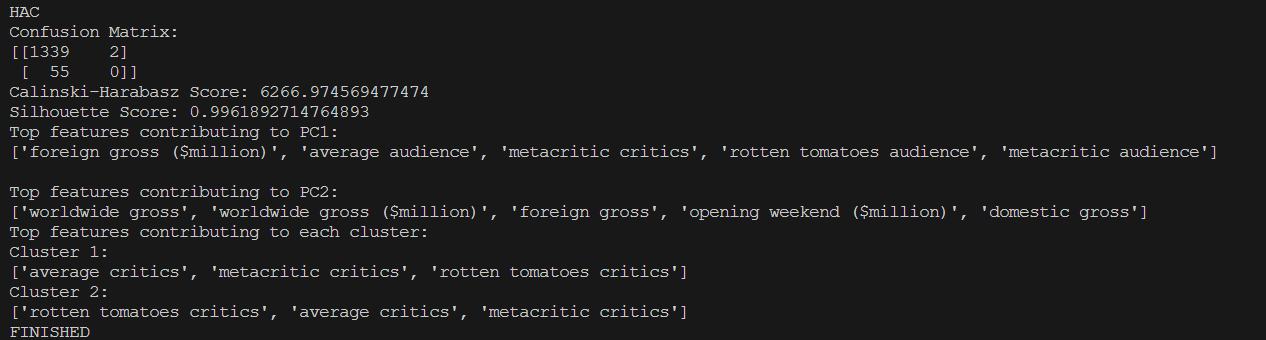
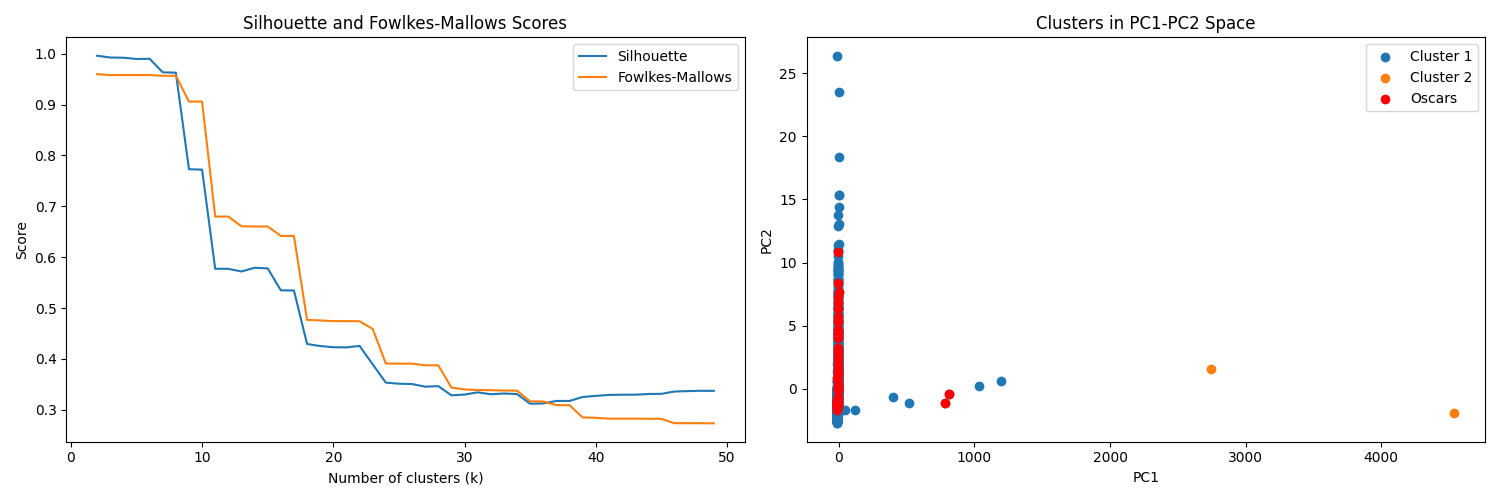
##### **StandardScaler**



**Comment:**

Clustering here seems to work well as long as most of the oscar winners, except from 2, are on the same cluster. Shilhouette score is good.

##### **RobustScaler**



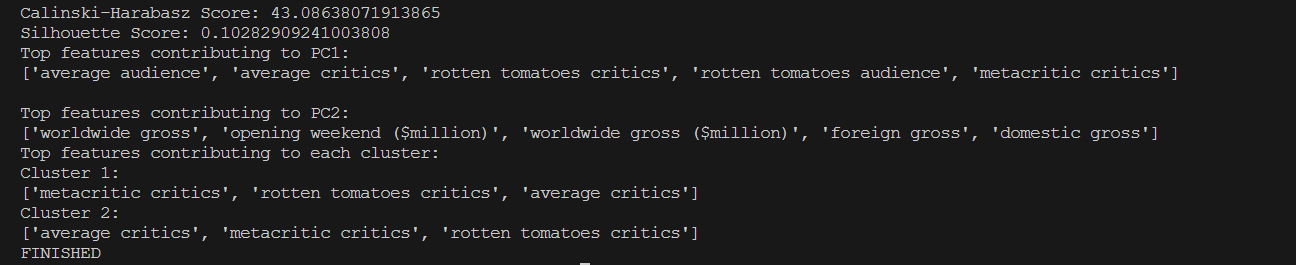
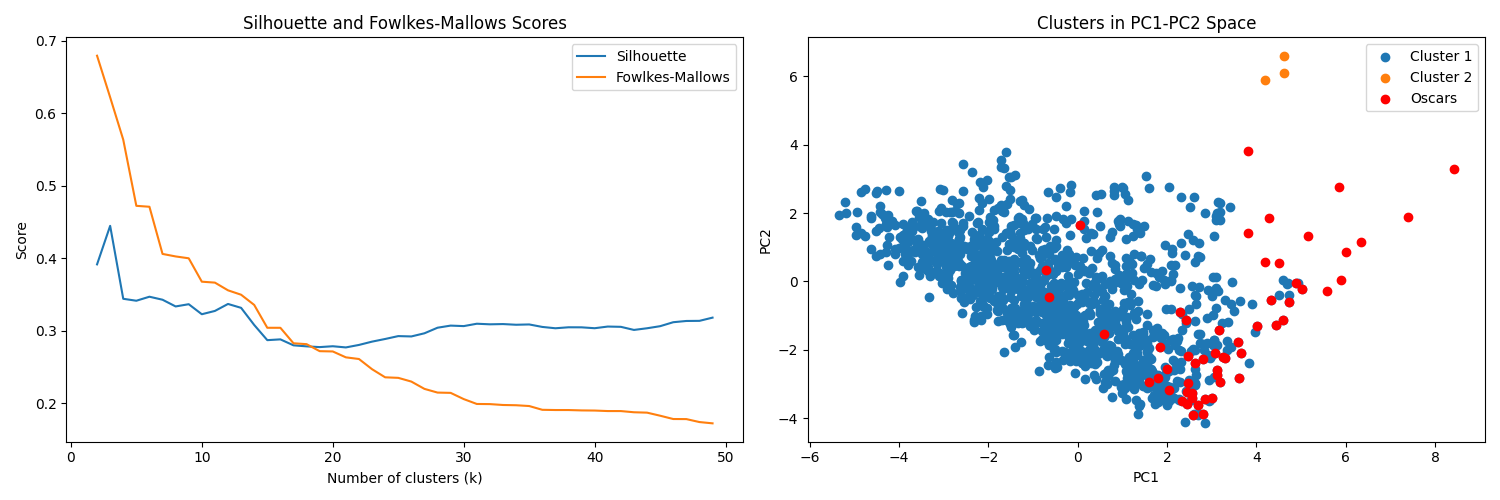
**Comment:**

Clustering here is not working well despite the fact that Silhouette score is high.

#### **DBSCAN**

##### **MinMaxScaler(not applicable in DBSCAN)**

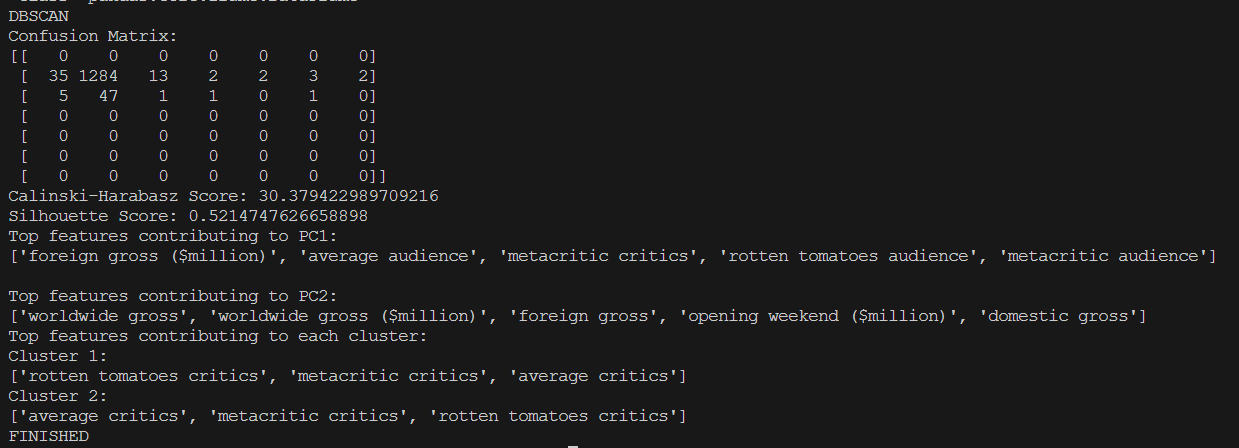
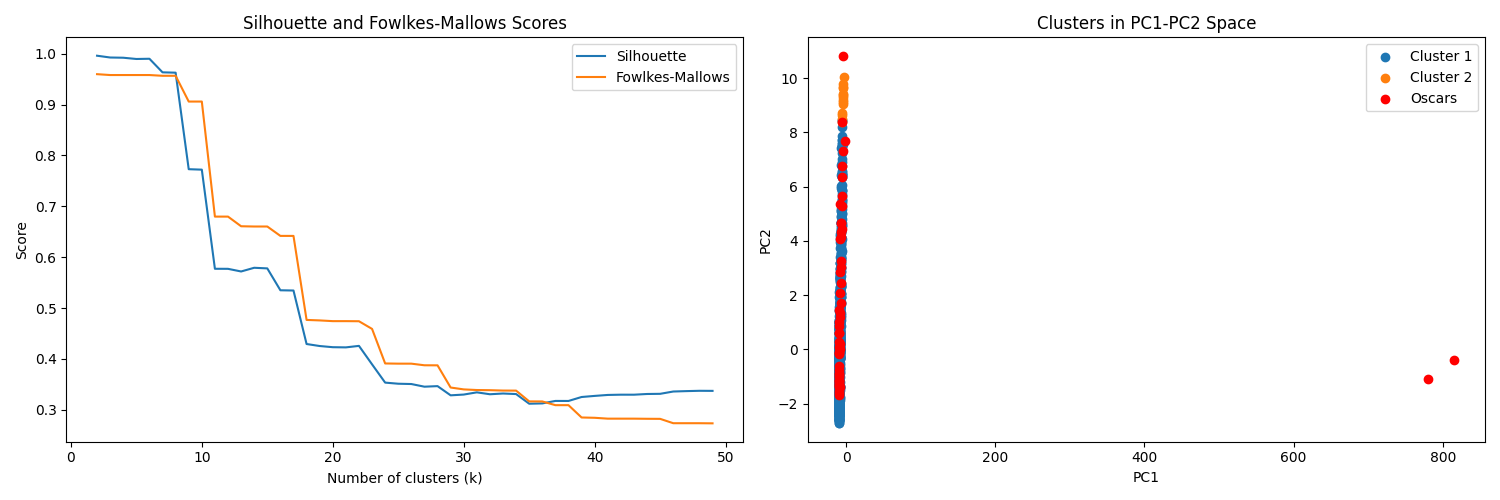
##### **StandardScaler**



**Comment:**

Clustering here is not working well because Silhouette score is low and Oscars seem to be not in the same cluster.

##### **RobustScaler**

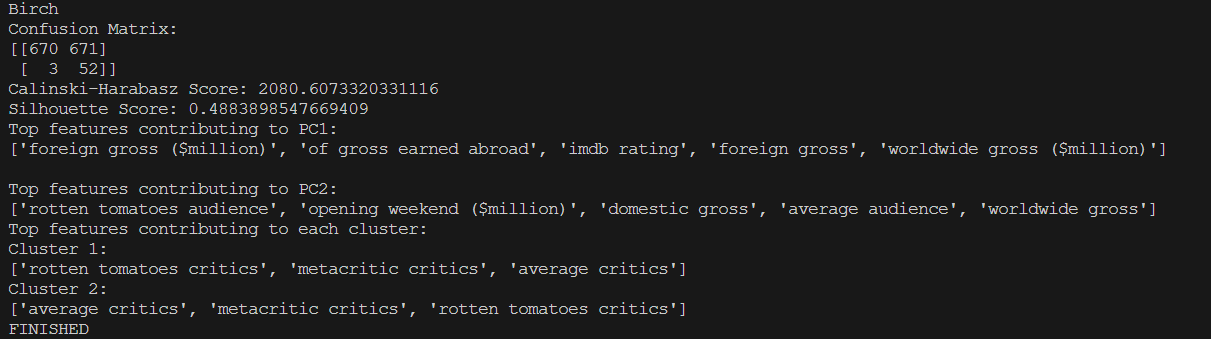
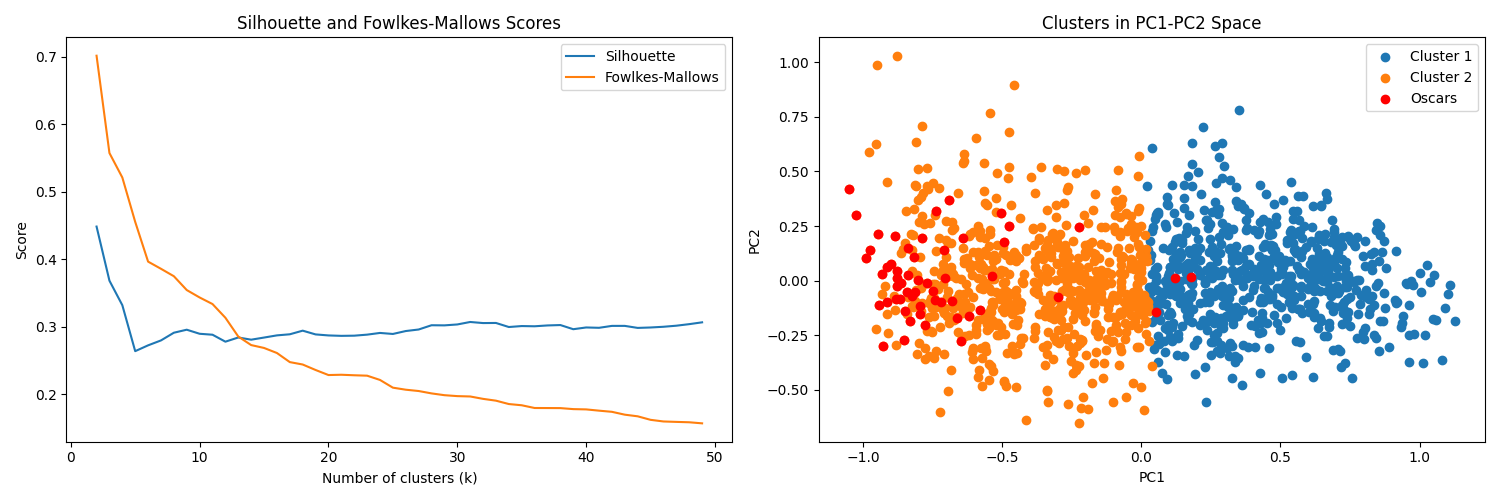


**Comment:**

Clustering here is not working well despite the fact that Silhouette score is good.

#### **BIRCH**

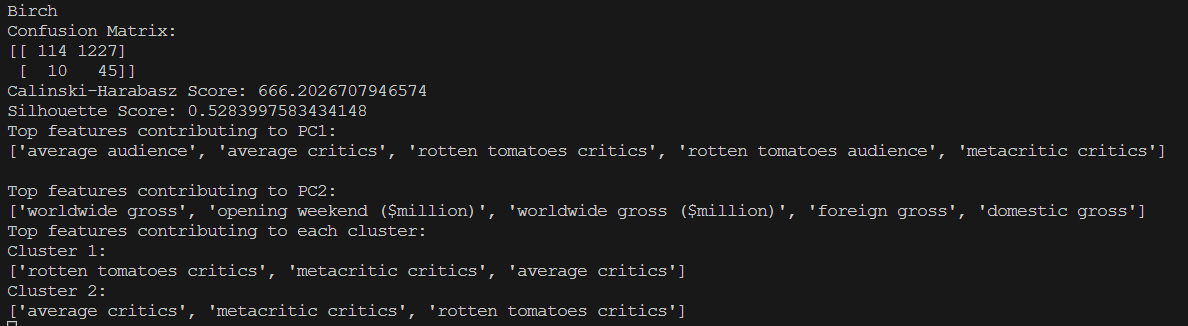
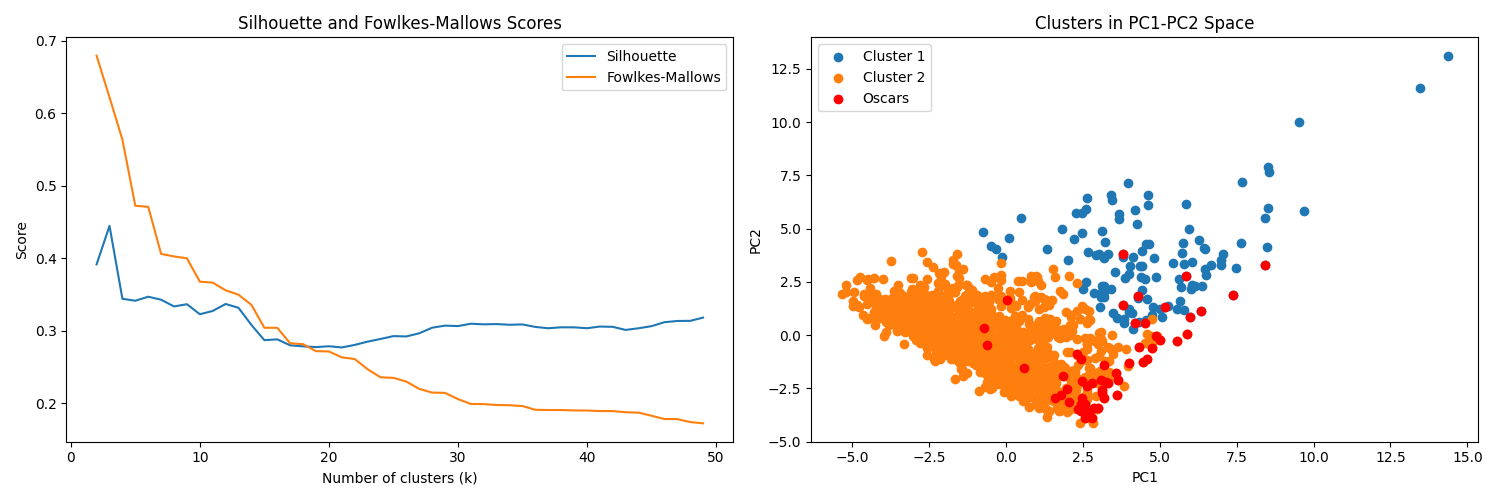
##### **MinMaxScaler**



**Comment:**

Clustering here seems to work well as long as most of the oscar winners, except from 3, are on the same cluster. Shilhouette score is good.

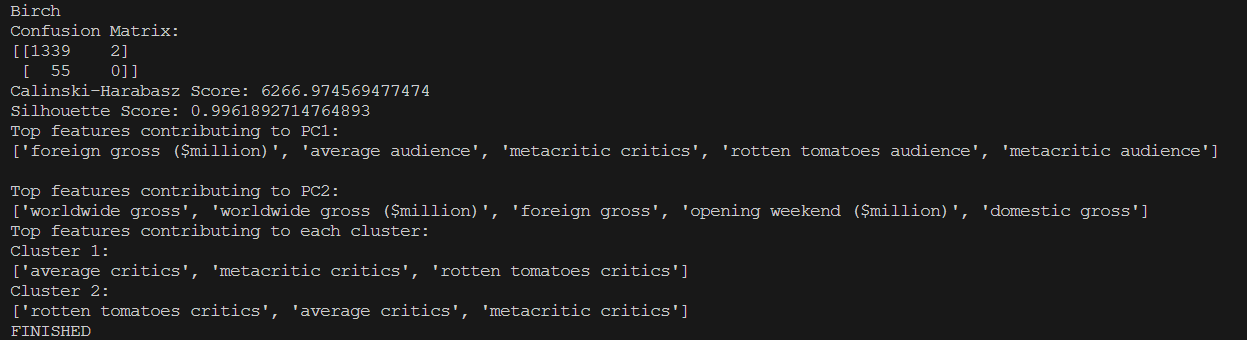
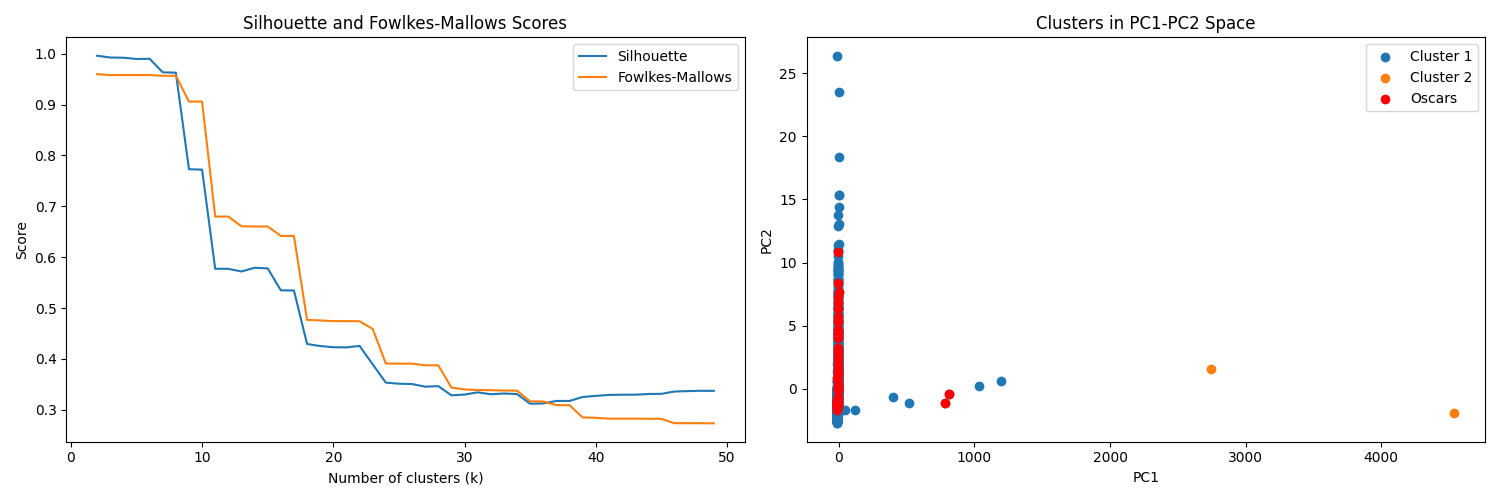
##### **StandardScaler**



**Comment:**

Clustering here is not working so well as long as most of the oscar winners, except from 2, are on the same cluster. Silhouette score is good.

##### **RobustScaler**



Clustering here is not working well despite the fact that Silhouette score is good.

**Conclusion for Clustering:**

As you notice it is very important the way of scaling the dataset. The reason we use 2 clusters in all the algorithms is because silhouette score has the highest value with 2 clusters (see blue line in figures). The clustering seems to be good in HAC and Kmeans as long as most of the Oscars are in the same cluster, but we did not make it to characterize each cluster according to their features, because cluster1 and cluster2 in all cases have the same features as most important.

# **CLOSING**

## **.readme**

**To run preprocessing and classification**

**FORMAT:** python3 main.py <lr/knn/rf/dtc/gb> <stats/nostats> <prepro/noprepro>

python3 main.py gb stats prepro

**To run classification (without doing preprocessing again)**

python3 main.py gb stats noprepro

**To run clustering**

python3 clusteringMain.py <KM/DBSCAN/HAC/Birch > <numClusters> <rs/ss/mm >

python3 clusteringMain.py HAC 2 mm

**To run the final and correct model use this command for the predictions I uploaded:**

python3 main.py gb stats prepro

**NOTE:**

For preprocessing is recommended to process the file **moviesUpdated.xlsx** because it is already filled with the imdb rating missing values.

In the file **movies.xlsx** imdb rating missing values are missing in all the rows so it will take a lot of time to fill all the missing values .

**moviesUpdated.xlsx**  is a copy of **movies.xlsx** with the only difference that **moviesUpdated.xlsx** has also the IMDb ratings column ready.

## **Links**

**GitHub:** <https://github.com/Kokkales/Oscars---Data-Mining>