Do (wo)men talk too much in films? Project in Machine Learning

Aniruddha Kshirsagar

Namika Maharjan

Yanhe zhou

Abstract

The aim of the project was to find whether the lead role in a movie is played by a Male or a Female given a set of variables. Thirteen such variables were analysed and different machine learning methods were used to train and predict the outcome. It was found after optimising the models that LDA, Logistic regression, Random forest and classification tree gave the least mean errors. LDA was chosen as the best model out of these four, and shall be used as a predictor for the project case. Number of group members: 3.

1 Introduction

When we talk about film industry the dominance of one particular gender is often discussed and we often hear about the claims prevailing on male domination in the industry. But these are often generalized claims without any data backing up. For an informed discussion and prediction of male or female lead dominance, it is required to check upon various categories affecting the lead dominance. No. of movies released, dialogues for each gender, profits made by the movie, age, era of movie release, etc. are some of the categories that helps to back up the dominance claim.

In this project, various machine learning models have been explored to predict the lead's gender for a movie based on the various properties of the film and put forward the model that predicts the gender with least errors. For this report, four models namely, Logistic Regression, Discriminant analysis (LDA), Tree-based methods (classification trees, random forests, bagging), Boosting and K-nearest neighbors have been considered for the prediction and model comparison.The overall aim of this project is to find a best training model to predict the lead gender for the test dataset.

2 Data analysis

The given data file was downloaded and uploaded to python DataFrame in a '.csv' format. After general screening of data for outliers, null value, data type and data formatting, the data was considered to be clean for further analysis. Basic statistical analysis was carried out by grouping the data by date and three specific points were analyzed:

a) Gender balance in speaking roles (change in time)

b) Which gender dominates speaking roles

c) Collection of the film based on the number of words by Male and Female

Furthermore, before training the model, the correlation of each variable with the response variable was checked to have a general overview of the influence of individual variables in the data set on the response variable. For this, Pearson's correlation coefficient was used.

Male actors account for 75.55% of all films while female actors account for only 24.45% as the lead, hence there is male dominance in speaking roles in Hollywood films. For the gender balance, the proportion of male is significantly higher than that of females, see figure 1. For the given research time, based on the gross data, the films with male lead make a total profit of 10237 but films with female lead make only 5400 implying to the result that the films where male do more speaking make more money. But this result maybe biased since there are less films.

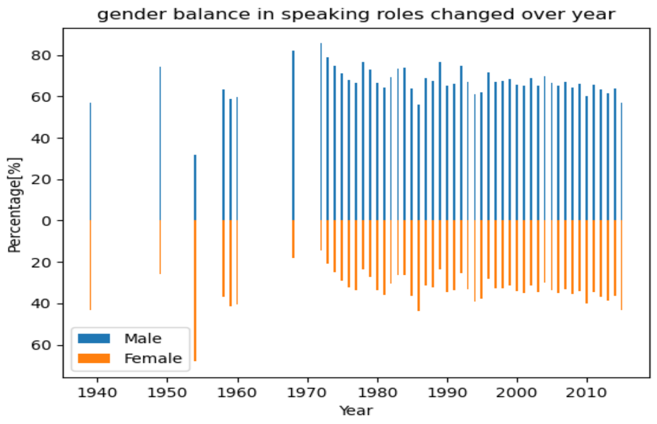

Figure 1: Gender balance in the speaking roles change over years from 1940 to 2015.

3 Methods

3.1 Working with the Data

3.1.1 Separating data

The data was separated into training data and test data. The separation method used was a stratified k fold method (package imported from sklearn). This method was selected because it separates the data according to the percentage split that already exists in the data. For example, if there were 30 males and 20 females. The stratified k fold would split the training and the test data into 60 and 40 percent. We found that this method would nullify the bias toward male, as there were more data points with male as the lead actors. To use this package males and females were given a dummy value of 0 and 1 respectively. The lead column was dropped from the data and using a 'for' loop the data was separated into training and test data, uploading it to a Dataframe. The function created was named 'StKfold' which takes in two values, given data (Dataframe) and the name of the column to be dropped(string). The function will return ten lists of four Dataframes each, two will represent a series of randomly separated 13 columns of data and the other two will represent the outputs for Lead actors.

3.1.2 Models used in project

A. Random Forest

Random forest is an ensemble method similar to bagging. The method injects additional randomness while constructing decision trees. This helps in reducing the correlation of the base models. At a node where the trees are split in this method all the input variables are not considered. Each time the inputs are selected randomly such that the number of selected inputs are less than or equal to the total input variables. Random forests decrease the correlation but increase the variance if compared to bagging.

B. Boost

Boosting is also an ensemble method which is primarily used to reduce bias. Like bagging boosting also uses multiple models to predict the result. Boosting creates a series of sequential models. Each model tries to correct the mistake made by the previous model. The training data set is modified in each iteration in such a way that the modified data will put more emphasis on the points that the model was unable to or could poorly capture. The final value is taken by the weighted average or a weighted majority vote if the model is a classification model.

C. Classification tree

The classification tree essentially splits the dataset into many disjointed regions. Each region has a specific number of similar data points. The number of data points may vary from region to region. A single predictor is used for all the data points in each region. The predictor is calculated by a majority vote.

D. Bagging

Bagging creates multiple training datasets from the given datasets by bootstrapping the data. Then it creates multiple models for each of the subsets. Bagging averages over all the base models it creates and returns the final value. It is an ensemble method and helps to reduce variance. The risk of overfitting also decreases with the help of this model.

E. K-NN

K-NN is short for k nearest neighbours, it is a non-parametric method. Where k belongs to integers. This model looks at k nearest neighbours for every point and calculates the value of the point by taking the average of all the points. If it is a classification model then the model takes a majority vote.

F. Linear Discriminant Analysis (LDA)

LDA is a linear classifier like logistic regression. It addresses some of the drawbacks of logistic regression and is better than logistic regression for multi class classification. LDA assumes that data is uniformly distributed and hence it works best when the outliers in the data are checked and skewness is removed. The model projects the features in a higher dimensional space. This method have been proven to work well in practice without the need of hyperparameter tuning.

G. Logistic regression

Logistic regression is an algorithm for classification problems yielding discrete outcome. Basically, logistic regression is a model that uses logistic function on linear regression to give categorical result. This method predicts the probability of each event occurring based on the independent variables and then the function uses the maximum likelihood for each variable and reiterates to find the best for each variable. However, the drawback of this method is that it can become unstable when the classes are well separated and if the parameter estimates are based on a limited sample size.

3.1.3 Creating functions for Machine learning Models

All the functions mentioned below take in a list as an argument produced by the StKfold function and return a list which includes tuples consisting of a model and its mean error. All the model packages used were imported from sklearn. The function used are as follow:

A. $classification_tree$(lst)

Using the sklearn package for this function. The following command is used to import this package ‘from sklearn.tree import DecisionTreeClassifier’. The package uses a Decision tree classification method which classifies the space by giving a specific definition or a rule. The complexity of the model is based on its depth. The following function uses a range of depths from 2 to 10.

B. $random_forest$(lst)

The following command is used to import the random forest package used in the function, ‘from sklearn.ensemble import RandomForestClassifier’. Random forest package essentially makes a large number of decision trees and chooses the output democratically for classification problems.

C. $bagging$(lst)

The following command is used to import the bagging classifier, ‘from sklearn.ensemble import BaggingClassifier’. Bagging essentially creates an ensemble of models using many subsets of the training data and reports the combined output of the models.

D. $bosting$(lst)

The Adaboost package is used in this function with the help of the command, ‘from sklearn.ensemble import AdaBoostClassifier’. The package uses an algorithm which selects a random part of the data and creates a model, and repeats this many times. Essentially each models learns from the one that is created before it.

E. KNN(lst)

The K nearest neighbors package is used in the function. The following command is used to import the package, ‘import sklearn.neighbors as sknn’. Depths from 2 to hundred are reiterated in the function and each model and error is appended to the list. This package essentially looks at the k nearest neighbors and classifies the point democratically.

F. $LogS$(lst)

In this function the Logistic regression package is used with the command ‘import sklearn.linear_model as skl’. Logistic regression is an algorithm for discrete outcome, it predicts the probability of each event occurring based on the independent variables and then the function uses the maximum likelihood for each variable and reiterates to find the best for each variable. However, the algorithm depends on the solver and method chosen. In this case, liblinear solver has been used with L2 regularization technique that follows ridge regression.

G. $Lda(lst)$

In this function linear discriminant analysis packs is used by importing sklearn.discriminant_analysis. The LDA classifier assumes all input variables to have normal distribution and same variance. Like logistic regression, LDA also follows linear classification algorithm.

3.1.4 Errors and Weights

Models and mean errors were obtained using the above functions. To further understand the behavior of these models, a figure was produced ,see Figure 2, which indicated the weight of each variable for our models. To do this, each variable was removed from the data set one by one and the models were again trained on a new train and test data set. The errors were obtained and Figure 3. Through the plots, specific variables can remove from the data set to make chosen models more efficient.

3.2 Model tuning and training

GridsearchCV function was used to exhaustive search over specified parameter values for estimators. GeidsearchCV will do the grid search and cross validation. GridsearchCV is to adjust the parameters sequentially according to the step size within the specified parameter range and use the adjusted parameters to train the learner to find the parameter with the highest accuracy on the verification set from all the parameters. RepeatedStratifiedKFold was used as cross validation method. Afterwards, the best parameters were used to train the model and selected the best performing model as the used in produce model. Comparing the prediction results difference between chosen model and a naive classifier which only predicts male. A naive classifier model was built i.e a model that has a single output, male. This model is then used to compare the prediction results of the chosen models. For the hyperparameter adjustment of different models, see the appendix.

4 Results

The error for each model before tuning are shown in table 1. Table 2 shows the mean errors and F1 score for every method with all parameters and deleting one specific parameter. Based on the error, model methods LDA, LogS, random forest and boost were chosen as the model for tuning to get a better performance. As the change in error is not significant for Lda and LogS, no parameter was deleted before the tuning. According to correlation coefficient shown in Figure 2, gross was deleted for boost and number of words lead for random forest. The error for Knn is close to the error for the naive classifier, so Knn will not be used for this project.The parameter tuning gave the best parameters for each method as follows:

a)AdaBoostClassifier(DecisionTreeClassifier(max_depth=4),n_estimators=900,learning_rate=1.3)

b)RandomForestClassifier(n_estimators=450,max_depth=d,max_features='auto',min_samples_leaf=1, min_samples_split=10,random_state=0)

c)skl_da.LinearDiscriminantAnalysis(solver='lsqr')

d)skl.LogisticRegression(solver ='liblinear',penalty='l2')

The error for the training is shown in Table 2. The LDA was chosen, due to it has the lowest error and high F1 score which means it has a more robust performance,as the one will use in produce. The differences show 0.185, 0.104, 0.153 and 0.175 for LogS, random forest, boost and LDA, which means the chosen models are not the same as a naive classifier and can be used.

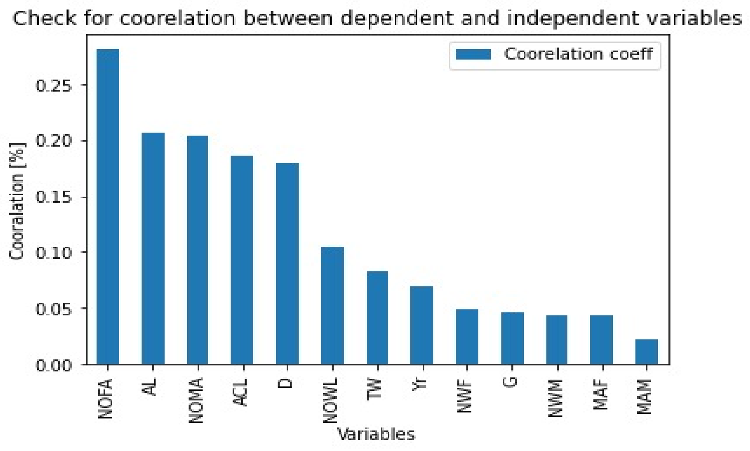

Figure 2: The correlation coefficient for parameters to lead gender.(NOFA= Number of female actors,AL=age lead,NOMA=number of male actors,ACL=age co-lead),D = different in words lead and co-lead, NOWL= number of word lead, TW=total word,Yr=year,NWF= number of word female, G= gross,NWM=number of words male,MAF=mean age female,MAM=mean age male

Table 1: The error for chosen models before tuning

Methods	Error
Boost	0.174757
LDA	0.0893786
LogS	0.135922
Random forest	0.211538
Knn	0.230769
Naive classifier	0.244464

Table 2: The final error and F1 score for chosen models.

Methods	Error	F1 score
Boost	0.116504	0.8889
LDA	0.087374	0.9074
LogS	0.097087	0.8981
Random forest	0.145631	0.834
Naive classifier	0.244464	/

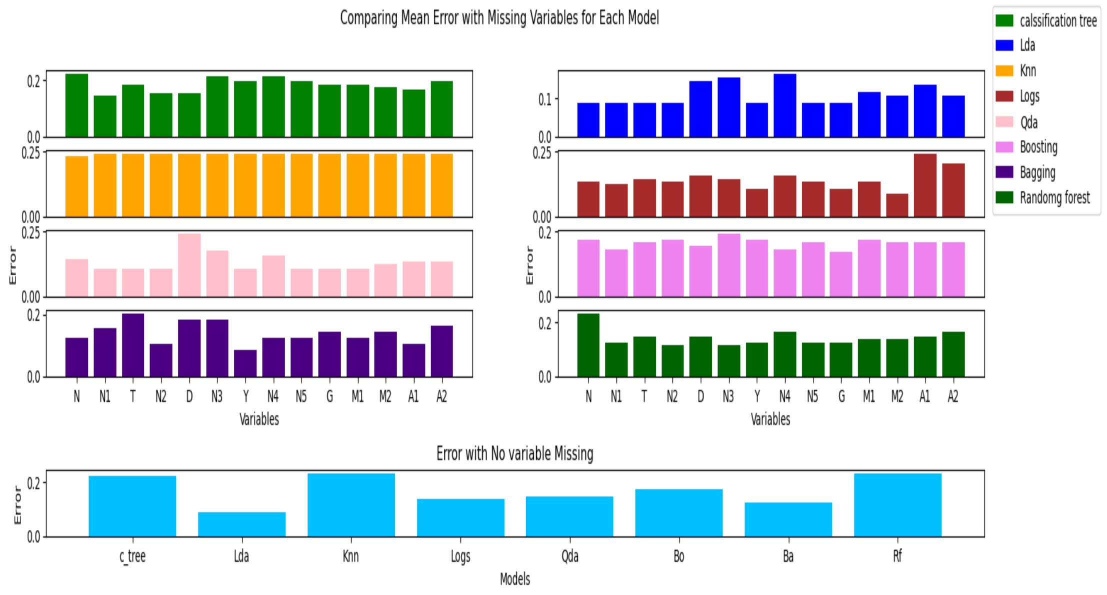

Figure 3: Comparing mean error with missing variables of each model(N1=number of word female, T=total word, N2=nmuber of word lead, D = different in words lead and co-lead, N3=number of male,Yr=year,N4=number of female actors,N5=number of words male, G= gross,M1=mean age male,M2=mean age female,A1=age lead,A2=age co-lead).

5 Discussion and conclusion

The GridsearchCV used in this project only checks the range that was chosen in this project case, it can not contain all the parameters and the parameters were checked have limitation number. Therefore, model is still available for further optimization. LogS, LDA and boost have relatively close error. Logistic Regression may have over fitted problem with high dimensional datasets. Linear Discriminant Analysis makes a normal distribution assumption on the variables, but it will be better when it satisfies the assumption. Boost method is sensitive to outliers. In this project case, the model chosen is the one with the least error namely LDA.

A Appendix

```
"""
Created on Sun Nov 27 16:27:50 2022

@author: 17103
"""
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
#classifacation tree
from sklearn import tree

#random forest and bagging
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
from sklearn.model_selection import StratifiedKFold as skfold
```

```
import statistics as sta
#bosting
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from sklearn.tree import DecisionTreeClassifier
train=pd.read_csv(r'D:\\Machine Learning\\Project\\train.csv',sep=',')
#knn
import sklearn.neighbors as sknn
#logistic regression
import sklearn.linear_model as skl
#LAD QDA
import sklearn.discriminant_analysis as skl_da
## statistic part
import random
from statistics import mean
train.head()
train.columns=['NWF','TW','NOWL','D','NOMA','Yr','NOFA','NWM','G','MAM','MAF',
               'AL','ACL','LEAD']
#Has gender balance in speaking roles changed over time (i.e. years)?
labels = train.loc[:,'Yr']
Yr = list(labels)
Yr = [*set(Yr)]
M = pd.DataFrame(Yr)
M['NWM'] = np.nan
M['NWF'] = np.nan
M['MP'] = np.nan
M['FP'] = np.nan

for i in range(len(M['NWM'])):
    r = train[Yr[i] == train['Yr']].index.values
    r1 = train[Yr[i] == train['Yr']].index.values
    M['NWM'][i]= np.sum(train.iloc[r,7])
    M['NWF'][i]= np.sum(train.iloc[r1,0])


for i in range(len(M['NWM'])):
    M['MP'][i] = (M['NWM'][i])/(M['NWM'][i]+ M['NWF'][i])
    M['FP'][i] = -(M['NWF'][i])/(M['NWM'][i]+ M['NWF'][i])

figure_1=plt.figure()
Female = M['FP']
Male = M['MP']
width=0.3
#plt.plot(Yr, Male, color = 'black')
#plt.plot(Yr,Female, color = 'black')
plt.bar(Yr, Male, width, label = 'Male')
plt.bar(Yr, Female, width, label ='Female')
plt.yticks([0.8,0.6,0.4,0.2,0,-0.2,-0.4,-0.6],[r'80',r'60',r'40',r'20',0,
                                             r'20',r'40',r'60'])
plt.ylabel('Percentage[%]')
plt.xlabel('Year')
plt.title('gender balance in speaking roles changed over year')
plt.legend()

#Do men or women dominate speaking roles in Hollywood movies?
#train['LEAD'].replace('Female',0,inplace=True)
#train['LEAD'].replace('Male',1,inplace=True)
#train['LEAD']=pd.to_numeric(train['LEAD'])
male_dominate=train.LEAD.value_counts().Male /len(train['LEAD'])
female_dominate=train.LEAD.value_counts().Female/len(train['LEAD'])
```

```
dominate=[male_dominate,female_dominate]

def make_autopct(values):
    def my_autopct(pct):
        total = sum(values)
        val = int(round(pct*total/100.0))
        return '{p:.2f}%'.format(p=pct,v=val)
    return my_autopct

figure_2=plt.figure()
plt.pie(dominate,labels=['Male','Female'],autopct=make_autopct(dominate))
plt.title('Dominate speaking roles in Hollywood movies')


#Do films in which men do more speaking make a lot more money than films in
#which women speak more?
M['MP']=np.nan
M['FP']=np.nan
#train['LEAD'].replace('Female',0,inplace=True)
#train['LEAD'].replace('Male',1,inplace=True)
#train['LEAD']=pd.to_numeric=(train['LEAD'])
for i in range(len(M['NWM'])):
    r = train[Yr[i] == train['Yr']].index.values
    for n in r:
        if train['NWF'][n]<train['NWM'][n] :
            M['MP'][i]= np.sum(train.iloc[n,8])
        else:
           M['FP'][i]= np.sum(train.iloc[n,8])
M['FP']=M['FP'].fillna(0)
M['MP']=M['MP'].fillna(0)

#by year use the code below
#plt.bar(M[0],M['MP'],color='red')
#plt.bar(M[0],M['FP'],color='blue')
#plt.scatter(M[0],M['MP'],color='red')
#plt.scatter(M[0],M['FP'],color='blue')
figure_3=plt.figure()
plt.bar('Male',M['MP'].sum(),width=0.5)
plt.bar('Female',M['FP'].sum(),width=0.5)
plt.title('Profit by gender')
plt.ylabel('Profit sum')
plt.xlabel('Gender')
plt.legend()


## for the Stratified K-Fold Cross Validation
train=pd.read_csv(r'D:\\Machine Learning\\Project\\train.csv',sep=',')
def StKfold(t , c):
    X = t.drop(columns = [c])
    y = t[c].replace({'Male' : 0 , 'Female' : 1})
    sk = skfold(n_splits = 10, shuffle = True, random_state = 1)
    lst = []
    for train_index, test_index in sk.split(X, y) :
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]
        lst.append((X_train, X_test, y_train, y_test))
    return lst
list1= StKfold(train, 'Lead')
## classification tree
```

```
def classification_tree(lst):
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        for z in range(2,10):
            model_ct=tree.DecisionTreeClassifier(max_depth=z)
            model_ct.fit(xtr,ytr)
            y_predict=model_ct.predict(xt)
            error=np.mean(y_predict !=yt)
            l.append((model_ct, error))
    return l
#error_classification_tree= classification_tree (2,list)


##random forest
def random_forest(lst):
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        for d in range(2,10):
            model_rf=RandomForestClassifier(max_depth= d,random_state=0)
            model_rf.fit(xtr,ytr)
            y_predict=model_rf.predict(xt)
            error=np.mean(y_predict !=yt)
            l.append((model_rf, error))
    return l

# error_Randomforest= random_forest (2,list)

##bagging
def bagging(lst):
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        for d in range(2,10):
            model_b=BaggingClassifier(n_estimators = d)
            model_b.fit(xtr,ytr)
            y_predict=model_b.predict(xt)
            error=np.mean(y_predict !=yt)
            l.append((model_b, error))
    return l
# error_Bagging= bagging (2,list)
##bosting
def bosting(lst):
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        model_ab=AdaBoostClassifier()
        model_ab.fit(xtr,ytr)
        y_predict=model_ab.predict(xt)
        error=np.mean(y_predict !=yt)
        l.append((model_ab, error))
    return l
#error_AdaBoost =bosting(list)

##KNN method
def Knn(lst):
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        for z in range(5, 100):
            modelKnn = sknn.KNeighborsClassifier(n_neighbors=z)
            modelKnn.fit(xtr,ytr)
            pred = modelKnn.predict(xt)
            error = np.mean(pred != yt)
            l.append((modelKnn, error))
    return l
##logistic regression
def LogS(lst):
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        model = skl.LogisticRegression()

        model.fit(xtr, ytr)

        pred = model.predict(xt)
        pred2 = model.predict(xt)
        error1 = np.mean(pred != yt)
        error2 = np.mean(pred2 != yt)
        l.append((model, error1))
    return l
##LDA


def Lda(lst):
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        model = skl_da.LinearDiscriminantAnalysis()
        model.fit(xtr,ytr)
        pred = model.predict(xt)
        error = np.mean(pred != yt)
        l.append((model, error))
    return l
##QDA
def Qda(lst):
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        model = skl_da.QuadraticDiscriminantAnalysis()
        model.fit(xtr,ytr)
        pred = model.predict(xt)
        error = np.mean( pred != yt)
        l.append((model, error))
    return l


##model choose
error_Classificationtree= classification_tree (list)
error_Randomforest= random_forest (list)
error_Bagging= bagging (list)
error_AdaBoost =bosting(list)
error_knn=Knn(list)
error_LogS=LogS(list)
error_LDA=Lda(list)
error_QDA=Qda(list)
for s in range(0,10):
    mean_error_ct=np.mean(error_Classificationtree[s][1])
    mean_error_rf=np.mean(error_Randomforest[s][1])
    mean_error_b=np.mean(error_Bagging[s][1])
    mean_error_ab=np.mean(error_AdaBoost[s][1])
    mean_error_knn=np.mean(error_knn[s][1])
    mean_error_logs=np.mean(error_LogS[s][1])
    mean_error_lda=np.mean(error_LDA[s][1])
    mean_error_qda=np.mean(error_QDA[s][1])
mean_error=[mean_error_ct,mean_error_rf,mean_error_b,mean_error_ab,mean_error_knn,
            mean_error_logs,mean_error_lda,mean_error_qda]
method=['classification tree','random forest','bagging','boost','knn','LogS','LDA','QDA']
print(method[mean_error.index(min(mean_error))], 'has the least error')

#here we want to use the classification model while the least error is LDA
#part need to discuss
#adboost were choosen in below case
e=[]
for s in range(0,10):
    e.append(error_AdaBoost[s][1])
def takeclosest(lst,number):
    idx=(np.abs(lst-number)).argmin()
    return idx
a=takeclosest(e,mean_error_ab)
model=error_AdaBoost[a][0]

#test=pd.read_csv(r'',sep=',')
#test['lead']=model.predict(test)

train.info()
Lp = ['Number words female', 'Total Words', 'Number of Words Lead',
'Difference in Words Lead and Co-Lead', 'Number of Male Actors', 'Year',
'Number of Female Actors', 'Number Words Male', 'Gross', 'Mean Age Male', 'Mean Age Female',
'Age Lead' , 'Age Co-Lead' , 'Lead']

F = ['classification_tree', 'Lda', 'Knn', 'Logs', 'Qda', 'boosting', 'bagging', 'Random Forest']
```

```
N = [1,2,3,4,5,6,7,8,9,10,11,12,13]
Func = []
def Mdf(f):
    Models = pd.DataFrame(N)
    for i in range(len(F)):
         Models[f'{F[i]}'] = np.nan
    Models.drop( columns = 0, inplace= True)
    return Models
Models = Mdf(F)
Models1 = Mdf(F)


clasTL = []
LdaL =[]
knnL =[]
LogsL =[]
QdaL = []
boostL = []
bagL=[]
RandomL=[]

Lp[0:8]
# Series of missing parameters descending order
## Just to see how the model behaves if we remove the parameter in series
for t in range(len(Lp)-1):
    g = Lp[0:t]
    Train = train.drop(columns = g)
    Lstt = StKfold(Train, 'Lead')
    c = 'ClasT' + str(t)
    c = classification_tree(Lstt)
    clasTL.append(c)
    c1 = 'Lda' + str(t)
    c1 = Lda(Lstt)

    LdaL.append(c1)
    c2 = 'Knn' + str(t)
    c2 = Knn(Lstt)
    knnL.append(c2)
    c3 = 'Logs' + str(t)
    c3 = LogS(Lstt)
    LogsL.append(c3)
    c4 = 'Qda' + str(t)
    c4 = Qda(Lstt)
    QdaL.append(c4)
    c5 = 'boosting' + str(t)
    c5 = bosting(Lstt)
    boostL.append(c5)
    c6 = 'bagging' + str(t)
    c6 = bagging(Lstt)
    bagL.append(c6)
    c7 = 'Random' + str(t)
    c7 = random_forest(Lstt)
    RandomL.append(c7)

clasTL1 = []
LdaL1 =[]
knnL1 =[]
LogsL1 =[]
QdaL1 = []
```

```python
boostL1 = []
bagL1=[]
RandomL1=[]

# Missing variable
## Looking for which variable affects the most in a simplistic way.
for t in range(len(Lp)-1):
    g = Lp[t]
    Train = train.drop(columns = g)
    Lstt = StKfold(Train, 'Lead')
    c = 'ClasT' + str(t)
    c = classification_tree(Lstt)
    clasTL1.append(c)
    c1 = 'Lda' + str(t)
    c1 = Lda(Lstt)

    LdaL1.append(c1)
    c2 = 'Knn' + str(t)
    c2 = Knn(Lstt)
    knnL1.append(c2)
    c3 = 'Logs' + str(t)
    c3 = LogS(Lstt)
    LogsL1.append(c3)
    c4 = 'Qda' + str(t)
    c4 = Qda(Lstt)
    QdaL1.append(c4)
    c5 = 'boosting' + str(t)
    c5 = bosting(Lstt)
    boostL1.append(c5)
    c6 = 'bagging' + str(t)
    c6 = bagging(Lstt)
    bagL1.append(c6)
    c7 = 'Random' + str(t)
    c7 = random_forest(Lstt)
    RandomL1.append(c7)

FuncL = [(clasTL1,0), (LdaL1,1), (knnL1,2), (LogsL1,3), (QdaL1,4), (boostL1,5),(bagL1, 6),
(RandomL1,7) ]
FuncL1 = [(clasTL,0), (LdaL,1), (knnL,2), (LogsL,3), (QdaL,4), (boostL,5),(bagL, 6),(RandomL,7)]



def MeR(f,df):
    for y in f:
        for i in range(13):
            for x in range(len(y[0][i])):

                 Er =[]
                 Em = y[0][i]
                 Er.append(Em[x][1])
                 ErM = mean(Er)
                 df.iloc[i, y[1]] = ErM
    return df
Models = MeR(FuncL,Models)
Models1 = MeR(FuncL1, Models1)

Lp1 = ['NwF','TW','NWL','DW','NMA','Y','NFA','NWM','G','MAM', 'MAF', 'AL' ,'AcL']

Models.plot(y = 'classification_tree', use_index = True,)
```

```
def PLotG(t,df, l,y):
    for i in range(8)  :
       plt.figure(i +y)
       plt.plot(t, df.iloc[:,i])
       plt.title(F[i])
       plt.ylabel('Mean Error')
       plt.xlabel(l)

PLotG(Lp1, Models, 'Missing Parameter', 4)
PLotG(N, Models1,'No.of missing parameters', 12)

#########Optimising the models
#for every model, after parameter truning, choose the best combination then put into training
part
#For each model the optimization code is run separately and the optimal parameters are
handpicked and put in the package.
X = train.drop(columns = ['Lead'])
y = train['Lead'].replace({'Male' : 0 , 'Female' : 1})
ab_paramters=[]

for i in range(1,6):
    model = AdaBoostClassifier(DecisionTreeClassifier(max_depth=i))
    grid = dict()
    grid['n_estimators'] = [50,100,150,200,300,500,600,700,800,900]
    grid['learning_rate'] =[0.01,0.1,0.2,0.3,0.4,0.5,1,1.1,1.2,1.3,1.4]
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3)
    grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1,
                               cv=cv, scoring='accuracy')
    grid_result = grid_search.fit(X, y)
    print("Best: %f using %s" % (grid_result.best_score_,
                                 grid_result.best_params_))
    ab_paramters.append([i,grid_result.best_score_,grid_result.best_params_])

def StKfold(X, y):
    sk = skfold(n_splits = 10, shuffle = True, random_state = 1)
    lst = []
    for train_index, test_index in sk.split(X, y) :
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]
        lst.append((X_train, X_test, y_train, y_test))
    return lst

list_Boost = StKfold(X,y)

def boosting(lst,m):
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        model_ab=m
        model_ab.fit(xtr,ytr)
        y_predict=model_ab.predict(xt)
        error=np.mean(y_predict !=yt)
        l.append((model_ab, error))
    return l
error_Boost=boosting(list_Boost,
                     AdaBoostClassifier(DecisionTreeClassifier(max_depth=4),
                                        n_estimators=900,learning_rate=1.3))

X = train.drop(columns = ['Lead', 'Number of Words Lead'])
y = train['Lead'].replace({'Male' : 0 , 'Female' : 1})
rf_paramters=[]
model = RandomForestClassifier()
grid = dict()
grid['n_estimators'] = [10,50,100,150,200,250,300,350,400,450,500]
grid['max_depth'] =[2,3,4,5,6,7,8,9,10]
grid['max_features'] =['auto','sqrt']
grid['min_samples_split'] =[2,6,10]
grid['min_samples_leaf'] =[1,3,4]
grid['random_state']=[0]
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv,
                           scoring='accuracy')
grid_result = grid_search.fit(X, y)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
rf_paramters.append([grid_result.best_score_, grid_result.best_params_])

list_Rf = StKfold(X,y)

def random_forest(d,lst):
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        model_rf=RandomForestClassifier(n_estimators=450,max_depth=d,max_features='auto',
        min_samples_leaf=1,min_samples_split=10,random_state=0)
        model_rf.fit(xtr,ytr)
        y_predict=model_rf.predict(xt)
        error=np.mean(y_predict !=yt)
        l.append((model_rf, error))
    return l

error_Rf= random_forest (10,list_Rf)


X = train.drop(columns = ['Lead'])
y = train['Lead'].replace({'Male' : 0 , 'Female' : 1})
lda_paramters=[]
model = skl_da.LinearDiscriminantAnalysis()
grid = dict()
grid['solver'] = ['lsqr', 'eigen']
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv,
                           scoring='accuracy')
grid_result = grid_search.fit(X, y)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
lda_paramters.append([grid_result.best_score_, grid_result.best_params_])

list_Lda = StKfold(X,y)

def Lda(lst,m):
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        model = m
        model.fit(xtr,ytr)
        pred = model.predict(xt)
        error = np.mean(pred != yt)
        l.append((model, error))
    return l
error_LDA=Lda(list_Lda,skl_da.LinearDiscriminantAnalysis(solver='lsqr'))

X = train.drop(columns = ['Lead','Mean Age Female'])
y = train['Lead'].replace({'Male' : 0 , 'Female' : 1})
logs_paramters=[]
model = skl.LogisticRegression()
grid = dict()
grid['penalty'] = ["l1","l2"]
grid['solver']=['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv,
                           scoring='accuracy')
grid_result = grid_search.fit(X, y)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
logs_paramters.append([grid_result.best_score_, grid_result.best_params_])

list_LogS = StKfold(X,y)

def LogS(lst,m):
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        model = m
        model.fit(xtr, ytr)
        pred = model.predict(xt)
        error1 = np.mean(pred != yt)
        l.append((model, error1))
    return l

error_LogS=LogS(list_LogS,skl.LogisticRegression(solver ='liblinear',penalty='l2'))

def naive_model(lst): # comparing model prediction to naive model
    l =[]
    for i in range(0,10):
        xtr = lst[i][0]
        ytr = lst[i][2]
        xt = lst[i][1]
        yt = lst[i][3]
        y_predict=0
        error=np.mean(y_predict !=ytr)
        l.append(error)
    return l
mean_error_nm=np.mean(naive_model(list))

for s in range(0,10):
    mean_error_rf=np.mean(error_Rf[s][1])
    mean_error_b=np.mean(error_Boost[s][1])
    mean_error_logs=np.mean(error_LogS[s][1])
    mean_error_lda=np.mean(error_LDA[s][1])
mean_error=[mean_error_rf,mean_error_b,mean_error_logs,mean_error_lda,
            mean_error_nm]
method=['Random Forest','Boost','LogS','LDA','Naive Model']
print(method[mean_error.index(min(mean_error))], 'has the least error')
###
mean_error_f={'Random Forest':mean_error_rf,
            'Boost':mean_error_b,
            'LogS':mean_error_logs,
            'LDA':mean_error_lda,
            'Naive Model':mean_error_nm}


def naive_model1(lst, lst12): # comparing model prediction to naive model
    l1 = []
    for z in range(len(lst12)):
        l =[]
        for i in range(0,10):
            xtr = lst[i][0]
            ytr = lst[i][2]
            xt = lst[i][1]
            lst12[z].fit(xtr,ytr)
            yt = lst12[z].predict(xt)
            y_predict=0
            error=np.mean(y_predict != yt)
            l.append(error)
        l1.append(np.mean(l))
    return l1
train=pd.read_csv(r'D:\\Machine Learning\\Project\\train.csv',sep=',')
# reading the dataframe again to produce new split
l123 = StKfold(train, 'Lead')

## Comparing the naive model to the four final selected models.
nc = [1,2,3,4,5]
Ncompare = pd.DataFrame(nc)
MLogsF= error_LogS[8][0]
MRfF = error_Randomforest[77][0]
MBoF = error_Boost[9][0]
MLdaF = error_LDA[9][0]

Mf = [MLogsF,MRfF,MBoF,MLdaF]

NaivM = naive_model1(l123, Mf)






Models.to_excel('D:\Machine Learning\Project\Models.xlsx')
# copying the dataframe to excel


##### PLOTS

M = pd.read_csv(r'D:\Machine Learning\Project\Models.csv')
F1 = ['c_tree', 'Lda', 'Knn', 'Logs', 'Qda', 'Bo', 'Ba', 'Rf']
```

```
fig  = plt.figure( figsize = (30,5))
s = gridspec.GridSpec(6,2)

ax0 = fig.add_subplot(s[0,0])
ax0.bar(M.iloc[:,0], M.iloc[:,1], color= 'Green', label = 'calssification tree')
ax0.set_xticks([])
ax1 = fig.add_subplot(s[0,1])
ax1.bar(M.iloc[:,0], M.iloc[:,2], color = 'BLue', label = 'Lda')
ax1.set_xticks([])
ax2 = fig.add_subplot(s[1,0])
ax2.bar(M.iloc[:,0], M.iloc[:,3], color = 'Orange', label = 'Knn')
ax2.set_xticks([])
ax3 = fig.add_subplot(s[1,1])
ax3.bar(M.iloc[:,0], M.iloc[:,4], color = 'Brown', label = 'Logs')
ax3.set_xticks([])
ax4 = fig.add_subplot(s[2,0])
ax4.bar(M.iloc[:,0], M.iloc[:,5], color = 'pink', label = 'Qda')
ax4.set_ylabel('Error')
ax4.set_xticks([])
ax5 = fig.add_subplot(s[2,1])
ax5.bar(M.iloc[:,0], M.iloc[:,6], color = 'Violet', label = 'Boosting')
ax5.set_ylabel('Error')
ax5.set_xticks([])
ax6 = fig.add_subplot(s[3,0])
ax6.set_ylabel("Variables")
ax6.bar(M.iloc[:,0], M.iloc[:,7], color = 'Indigo', label = 'Bagging')
#ax6.set_xticks([])
ax7 = fig.add_subplot(s[3,1])
ax7.bar(M.iloc[:,0], M.iloc[:,8], color = 'darkgreen', label = 'Randomg forest')
ax7.set_xlabel('Variables')
#ax7.set_xticks([])
ax8 = fig.add_subplot(s[5,:])
ax8.bar(F1, M.iloc[0,1:9], color= 'deepskyblue' )
ax8.title.set_text('Error with No variable Missing')
ax8.set_ylabel('Error')
ax8.set_xlabel('Models')
fig.legend()
fig.suptitle('Comparing Mean Error with Missing Variables for Each Model')
```