SML Project: Do (wo)men talk too much in films?

Derrick Nii Adjei Adjei, Gabrielle Fidelis de Castilho,
Kalugala Muhandiramge Chathura Jayasanka
Uppsala University
adjeiderrick12@gmail.com, gabriellecastilho@gmail.com, cjayget@gmail.com

Abstract

In this report, we present an analysis of Hollywood movies based on a given data set. We use multiple machine-learning methods to predict the gender of the lead actor based on 13 features using python programming language. Multiple parameters are tuned in each model to achieve the highest accuracy. The chosen model, deep neural networks, is then used to predict the gender of the lead actor from an unknown test data set.

1 Introduction

There have been allegations in Hollywood of sexism and racism in movie roles, in which white men dominate movie roles. A motivated team gathered data from eight thousand (8,000) screenplays segregated into gender and age, then analyzed it to confirm the allegations [1]. In this project, we set out to produce machine learning models that can predict the gender of the lead role using data such as the year the movie was released, the number of female actors, profits made by the film, the number of words for each gender, among others. The algorithms used were logistic regression, discriminant analysis, K-nearest neighbors, random forests (tree-based methods), boosting, and deep neural networks. The highest-performing algorithm will be used to classify an unknown test set.

2 Exploratory data analysis

2.1 Do men or women dominate speaking roles in Hollywood movies?

According to Fig. 1, males dominate speaking roles in the lead and supporting roles, representing 70.87% of the total words spoken.

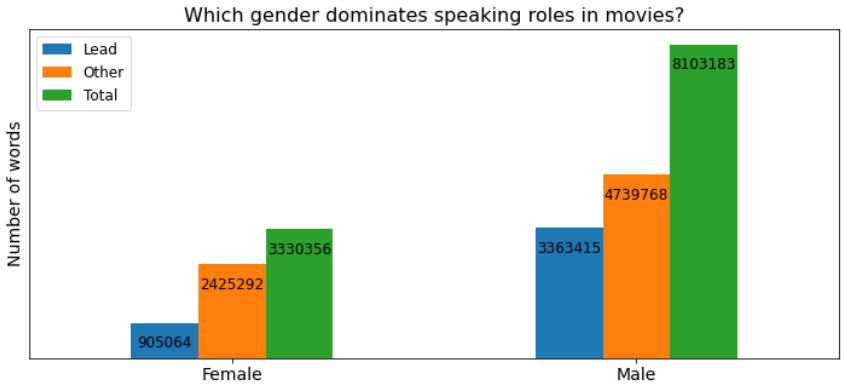

Figure 1: Number of words per gender and type role

2.2 Has gender balance in speaking roles changed over time (i.e.,, years)?

According to Fig. 2, men have had more leading roles than women, with the difference being on a steep rise until the 90s and declining abruptly by the end of the decade. The gap increased again in the 2000s, but the current trend is of a decline.

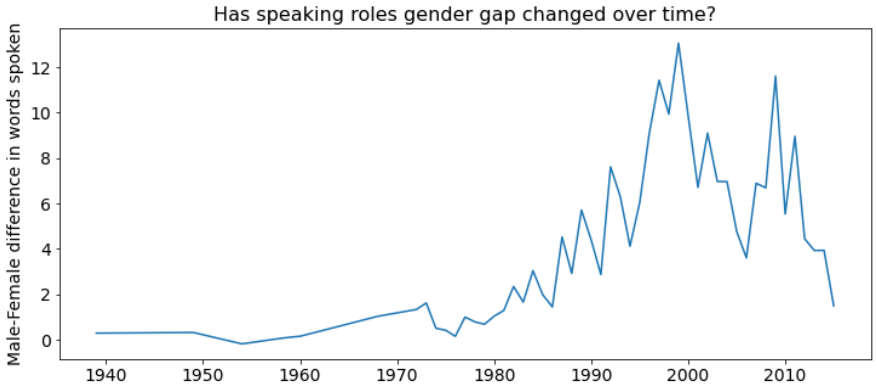

Figure 2: Difference in leading role number between genders

2.3 Do films in which men do more speaking make a lot more money than films in which women speak more?

As observed in Fig. 3, films in which men speak more than women profit more on average, with a difference of 35 units, which exceeds its counterpart by approximately 42%.

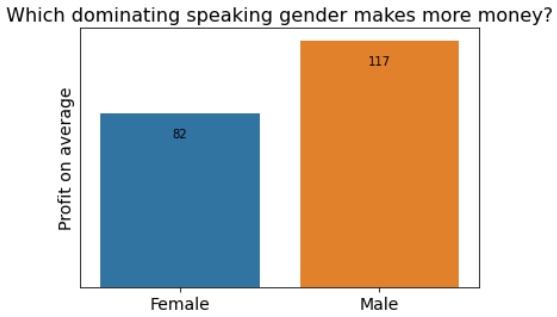

Figure 3: Number of words per gender and type role

3 Data pre-processing

3.1 Feature selection

A fundamental approach to tackling supervised learning problem is feature engineering and selection. One way to remove noise from the data is to identify linear dependencies in the data and drop highly dependent features. To achieve this, we use correlation to determine the linear dependencies and then prune the features.

From Fig. 4, the features "*Total words*", *"Difference in words lead and co-lead"*, and *"Number of words lead"* have a high correlation. After some testing, we decided to drop the *"Total words"* feature.

3.2 Feature scaling

Features may have varying values, affecting how well the model learns from the data. Using *StandardScaler* from *sklearn*, we scale the training data points to help the model understand and learn from the training data.

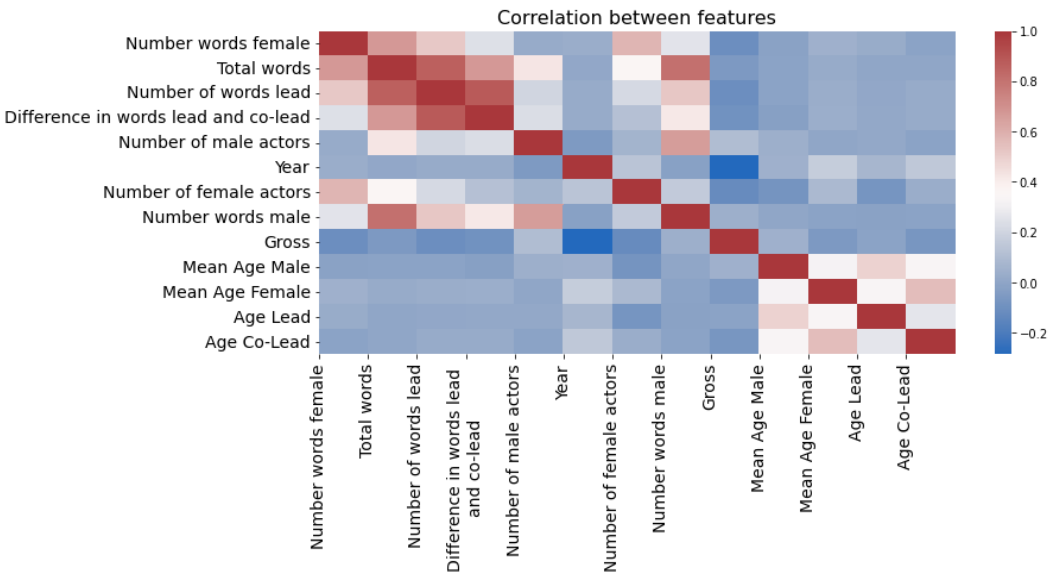

Figure 4: Grid of correlations between input columns

4 Linear and quadratic discriminant analysis

Linear and quadratic discriminant models are generative models derived from the Gaussian mixture model (GMM) [2]. They provide a probabilistic description of how to generate the input and output data. Since the logarithm of the Gaussian probability density function is a quadratic function in x, the decision boundary for this classifier is also quadratic, and the method is referred to as quadratic discriminant analysis (QDA). Making an additional simplifying assumption, we instead obtain the linear discriminant analysis (LDA).

4.1 LDA

4.1.1 Training the model

Here are outlined the steps taken to train and find the best model:

- Create a *train-test-split* from the data with our selected features. The test data size is set to 10%, and the train data size at 90%. We will run 12-fold cross-validation on the training set, so we picked a small size for the final test set.
- Try out multiple parameters using a pipeline and apply grid search cross-validation to run the algorithm for each parameter and return the best estimator model. We experimented with the following:
  - solver = ['lsqr', 'eigen'] and 'svd': Describes the behavior of a matrix on a particular set of vectors. The 'svd' solver does not rely on the covariance matrix and may be preferable when the number of features is extensive. Since it does not support shrinkage, it was done separately. The 'lsqr' solver is an efficient algorithm that only works for classification. The 'eigen' solver is based on optimizing the between-class scatter to the within-class scatter ratio.
  - shrinkage = ['auto']: A shrinkage is a form of regularization used to improve the estimation of covariance matrices. It is useful in situations where the number of training samples is small compared to the number of features.

  Each cross-validation fold uses the same approach so the overall train time is increased.

4.2 QDA

4.2.1 Training the model

- The *train-test-split* step is the same as in Section 4.1
- The pipeline approach was implemented here. We experimented with the following:
  - reg_param = [0.0, 0.1, 0.2, 0.3, 0.5, 0.8, 1.0]: Regularization used to improve the estimation.

  Each cross-validation fold uses the same approach, increasing the overall train time. The only solver available for QDA is 'svd'.

4.3 Best estimator parameters for LDA and QDA

The parameters for the best estimator of *LDA* we found are solver = **'lsqr'** and shrinkage = **'auto'**, with an accuracy of 0.875, i.e., **87.5%**. For *QDA*, the best parameter is reg_param = **0.0**, with an accuracy of 0.904, i.e., **90.4%**. For being more flexible, the QDA model provided a better result than (or at least as good as) the LDA model.

5 K nearest neighbors

The KNN algorithm tries to find only one k value, the number of nearest neighbors with the least error on a given training set. Once k is found, given a test input, a majority vote is carried out for classification problems and the average for regression problems. KNN is known to perform better on classification problems, in which it is recommended to use an odd value for k since a majority vote will always predict one class. In case k is even, we can recalculate the distances and predict the class closest to k or pick one of the classes.

5.1 Training the model

- The *train-test-split* step is the same as in Section 4.1
- The pipeline approach was implemented here. We experimented with various parameters:
  - $\{k : k \in [1, 30]\}$: Values which k can take. For each value, find the error and select the value of k with the least error.
  - weights = ['uniform', 'distance']: For each value of k, apply *"uniform"* weights to all data points such that all points have the same influence on a test point; and *"distance"* weight which is closer points have stronger influence to a test point.
  - metrics = ['minkowski', 'manhattan', 'euclidean']: For each value of k and weights, run the metrics used for the distance calculation between point.. In essence, it tries the different distance calculation methods and returns the best one that minimizes the error in the classification.

  Each cross-validation fold uses the same approach, increasing the overall train time.

5.2 Best estimator parameters for KNN

The parameters for the best estimator of the *KNN* we found are "metric" = **"manhattan"**, "n_neighbors" = **"14"** and weight = **"uniform"**. The accuracy for the model was 0.885, i.e., **88.5%**.

6 Logistic regression

Logistic regression is similar to linear regression but uses an activation function, making it a classifier. In this type of regression, we find the values of $\hat{\theta}$ from Eq. 1, which minimizes the classification errors from the activation Eq. 2.

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_p x_p = \theta^T x \tag{1}$$

$$h(z) = \frac{e^z}{1 + e^z} \tag{2}$$

6.1 Training the model

- The *train-test-split* step is the same as in Section 4.1
- The pipeline approach was implemented here. We experimented the parameters:
  - $\{c = [2.0, 1.0, 0.5, 0.25, 0.0625]\}$: The strength of regularization is increased with smaller values of c. It penalizes features that do not improve the accuracy of the model
  - fit_intercept = [True, False]: This indicates whether a bias (constant) should be added to the *decision function*, i.e., 1

  Each cross-validation fold uses the same approach, increasing the overall train time.

6.2 Best estimator parameters for logistic regression

The parameter for the best estimator of the *Logistic Regression* we found is "C" = **"2"**, "fit_intercept" = **"True"**. The accuracy for the model was also 0.885, i.e., **88.5%**.

7 Random forests (tree-based methods)

Random forests is an algorithm that makes use of multiple trees to make a prediction. It does this by creating multiple sub data sets with repetition from the original one. For each sub data set, random features are selected to guarantee variety and reduce correlation between the trees. Then, a decision tree is built for each sub data set. For predictions, a test data point is applied to each tree and the output is either averaged (for regression) or classified by majority vote (classification).

7.1 Training the model

- The *train-test-split* step is the same as in Section 4.1
- We experimented with the following parameters:
  - criterion = ["gini", "entropy"]: The criteria for a split. Each criterion specifies the values for which a split should occur.
  - min_samples_split = [2, 3, 4, 5, 7, 9]: The minimum number of samples to merit a split. It means that a region in the decision space can only be split if it has the minimum number of samples
  - max_features = ["sqrt", "log2"]: The function to calculate the max number of features per data subset in the bootstrapping stage.

7.2 Best estimator parameters for Random forests

The parameters for the best estimator of the *Random forests* is "criterion" = **"entropy"**, "max_features"=**"sqrt"** and "min_samples_split" = **"4"**. The accuracy for the model was also 0.894, i.e **89.4%**.

8 Boosting

Boosting is a variant of bagging where it tries to improve not well performing weak learners. Boosting converts multiple weak learners to a single robust model using decision trees. By using incorrect weak learners and prioritizing them over others, the next iteration is improved iteratively to obtain the final result. This final result is a combination of weak learners that can be used for predictions more accurately. One of the disadvantages of boosting is that multiple models need to be trained sequentially and it takes more time on average compared to some other methods on same data set. This is different in XGBoost as it can utilize techniques such as parallelization, distributed computing and cache optimization.

8.1 Training the model

- The *train-test-split* step is the same as in Section 4.1

- The *feature scaling* is not applied as boosting methods are not affected by it. Trees are based on conditions, and the range of value does not influence these tree splits.
- Different parameters are used in the boosting methods explored below.
- Optimal parameter values are listed in Section 8.2

Three boosting methods explored:

1. Gradient boosting
    - Trying to iteratively reduce prediction error using multiple trees. Output from previous tree is taken and all outputs are scaled with *learning rate* parameter so that resulting model after many iterations will have low variance. Tree depth can be higher than *Adaptive boosting*.
2. Adaptive boosting (AdaBoost)
    - Rather than scaling output as *Gradient boosting*, *Adaptive boosting* assigns weights to weak learners so that least performing trees contribute more to next model creation. Typically tree depth is set to 1.
3. Extreme gradient boosting (XGBoost)
    - Version of *Gradient boosting* which is optimized for speed and scale of input data. Unlike both other boosting methods, *XGBoost* can utilize parallel processing and distributed computing which makes it a popular choice for applications with large input data.

8.2 Best estimator parameters for boosting

8.2.1 Gradient boosting

The parameters for the best estimator of the *gradient boosting* we found are "learning_rate" = **"0.03"**, "max_depth" = **"4"**, "n_estimators" = **"500"** and "subsample" = **"1.0"**. The accuracy for the model was also 0.8919, i.e., **89.19%**.

8.2.2 Adaptive boosting

The parameters for the best estimator of the *adaptive boosting* we found are "algorithm" = **"SAMME.R"**, "learning_rate" = **"0.05"** and "n_estimators" = **"2000"**. The accuracy for the model was also 0.8749, i.e., **87.49%**.

8.2.3 Extreme gradient boosting

The parameters for the best estimator of the *Extreme gradient boosting* we found are "booster" = **"gbtree"**, "grow_policy" = **"depthwise"**, "learning_rate" = **"0.03"** and "n_estimators" = **"1000"**. The accuracy for the model was also 0.9135, i.e., **91.35%**.

9 Deep neural networks (DNN)

We also experimented with deep neural networks. A neural network uses a linear model that assigns weights to the input, features, and activation function. This is mathematically represented below:

$$\hat{y} = h(W_1 x_1 + W_2 x_2 + ... + W_p x_p + b) \tag{3}$$

$$h(z) = \frac{1}{1 + e^{-z}} \tag{4}$$

$$h(z) = max(0, z) \tag{5}$$

Activation functions can take many forms, but the common choices are logistic (Eq. 4) and ReLU (Eq. 5). DNNs use multiple layers of perceptrons to learn a model. The first layer is the input layer, and the subsequent layers define the depth of the DNN. A DNN has two or more layers; otherwise, it is a neural network.

9.1 Training the model

- The *train-test-split* step is the same as in Section 4.1
- No pipeline approach was used since this can increase the training time indefinitely. We experimented with the parameters:
  - {solver = ['lbfgs', 'sgd', 'adam']}: The algorithm for weight optimization.
  - activation = ['identity', 'logistic', 'tanh', 'relu']: The activation function for the hidden layer.
  - hidden_layer_sizes = $[[12, 14, 14], [12, 13, 13], [12, 15, 15]]$
- Performed a simple cross-validation using *train-test-split* with test data set of 30% of the original.

9.2 Best estimator parameters for DNN

The parameters for the best estimator of the *DNN* is "solver" = **"sgd"**, "activation"=**"tanh"**, "max_iter" = **"15000"** and hidden_layer_sizes = $[12, 14, 14]$.
The accuracy for the model was 0.910, i.e **91.0%**.

10 Conclusion

From all nine compared machine learning models, the *Extreme gradient boosting* model gave the highest accuracy for this task, while *Deep neural network* and *QDA* also achieved comparable performance. Figure 5 shows every model we analyzed with the maximum accuracy achieved by it. The accuracy for each model refers to the average accuracy for each cross validation fold.

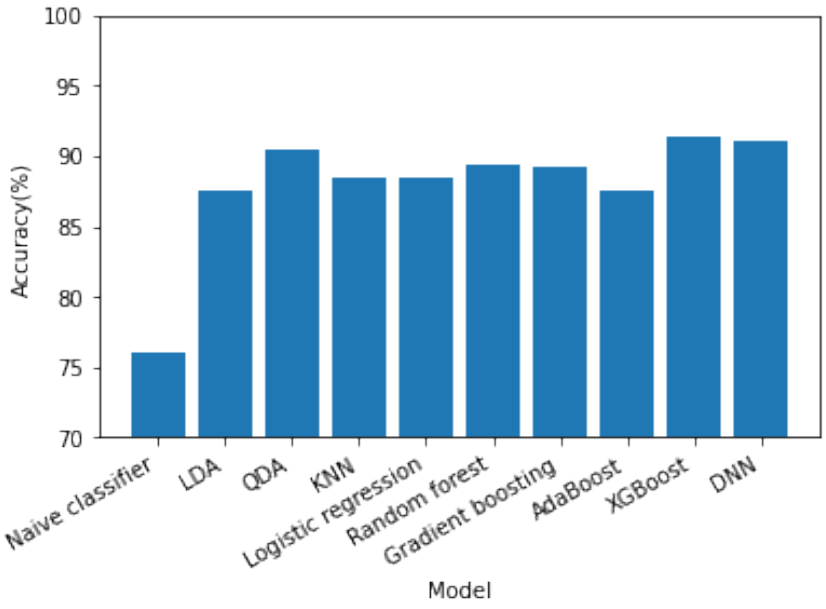

Figure 5: Model accuracy comparison

With a more extensive data set, though, *DNN* would have given more accurate predictions, as neural networks tend to perform better as data points increase. If used with all movie data available rather than a limited data set, a trained *DNN* model would be even more accurate. Also, with the order of the train-test split, boosting gave more variance in prediction accuracy. For these reasons, we chose DNN over the other contenders.
All analyses and predictions in this report were made using a comparable small data set, and all conclusions are based only on available data. Actual gender bias in Hollywood movies could be different.

References

[1] Anderson, H. & Daniells, M. (2017). *Film Dialogue from 2,000 screenplays, Broken Down by Gender and Age.*. url = "https://pudding.cool/2017/03/film-dialogue/". Accessed: 23.11.2022.

[2] Lindholm, A., Wahlström, N., Lidsten, F. & Schön, T. B. (2022). *Machine Learning: A First Course for Engineers and Scientists.* Pre-publication version. Cambridge University Press.

A Appendix

A.1 Preparation

Importing packages and loading data sets.

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

train_url = "https://raw.githubusercontent.com/gabriellecastilho/
              datasets/master/train.csv"

train_data = pd.read_csv(train_url)
```

A.2 Do men or women dominate speaking roles in Hollywood movies?

Calculating number of words per gender per role.

```
female_lead_words = train_data[train_data['Lead'] == 'Female']
              ['Number of words lead'].sum()
male_lead_words = train_data[train_data['Lead'] == 'Male']
              ['Number of words lead'].sum()

female_other_words = train_data['Number words female'].sum()
male_other_words = train_data['Number words male'].sum()

female_words = female_lead_words + female_other_words
male_words = male_lead_words + male_other_words

words = pd.DataFrame()
words['Gender'] = ['Female', 'Male']
words['Lead'] = [round(female_lead_words), round(male_lead_words)]
words['Other'] = [round(female_other_words), round(male_other_words)]
words['Total'] = words['Lead'] + words['Other']
words['Percen'] = round(words['Total'] / words['Total'].sum() * 100, 2)
```

Creating Figure 1.

```
words_plot = words.plot(x="Gender", y=['Lead', 'Other', 'Total'], kind="bar", figsize=(12, 5))
plt.title('Which gender dominates speaking roles in movies?', fontsize = 16)

plt.ylabel('Number of words', fontsize =14)
plt.xlabel('')
plt.yticks([], fontsize=14)
plt.xticks(rotation=0, fontsize=14)
plt.legend(loc='upper left', fontsize = 12)

for bar in words_plot.patches:
    words_plot.annotate(round(bar.get_height()), (bar.get_x() + bar.get_width() / 2,
              bar.get_height() - 800000), ha = 'center', va = 'center',
              xytext = (0, 9), textcoords = 'offset points', fontsize = 12)
```

A.3 Has gender balance in speaking roles changed over time (i.e., years)?

Calculating the difference in words spoken per gender.

```
data_years = pd.DataFrame()
data_years['Year'] = train_data['Year']

data_years['Female lead words'] = train_data[train_data['Lead'] == 'Female']['Number of words lead']
data_years['Female lead words'] = data_years['Female lead words'].fillna(0)
data_years['Male lead words'] = train_data[train_data['Lead'] == 'Male']['Number of words lead']
data_years['Male lead words'] = data_years['Male lead words'].fillna(0)

data_years['Female total words'] = train_data['Number words female'] + data_years['Female lead words']
data_years['Male total words'] = train_data['Number words male'] + data_years['Male lead words']

#Difference normalized by total words
data_years['Diff'] = (train_data['Number words male'] - train_data['Number words female']) /
                train_data['Total words']
data_years = data_years.drop(columns=['Female lead words', 'Male lead words']).groupby('Year').sum()
```

Creating Figure 2.

```
data_years['Diff'].plot(figsize=(12,5))
plt.title('Has speaking roles gender gap changed over time?', fontsize = 16)
plt.ylabel('Male-Female difference in words spoken', fontsize = 14)
plt.yticks(fontsize=14)
plt.xticks(fontsize=14)
plt.show()
```

A.4 Do films in which men do more speaking make a lot more money than films in which women speak more?

Calculating gross profit per gender on average.

```
profit = train_data[['Lead','Number of words lead', 'Number words female',
        'Number words male', 'Gross']]
profit['Total words female'] = profit.apply(
                lambda x: x['Number of words lead'] + x['Number words female']
                if x['Lead'] == 'Female' else x['Number words female'] , axis=1)
profit['Total words male'] = profit.apply(
                lambda x: x['Number of words lead'] + x['Number words male']
                if x['Lead'] == 'Male' else x['Number words male'] , axis=1)
profit['Speaks more'] = profit.apply(
                lambda x: 'Female'
                if x['Total words female'] > x['Total words male'] else 'Male' , axis=1)
profit = profit[['Gross', 'Speaks more']]
profit = profit.groupby('Speaks more', as_index=False).mean()
```

Creating Figure 3.

```
profit_plot = sns.barplot(data = profit, x="Speaks more", y='Gross')
plt.title('Which dominating speaking gender makes more money?', fontsize = 16)

plt.xlabel('', fontsize = 14)
plt.ylabel('Profit on average', fontsize = 14)
plt.xticks(rotation=0, fontsize = 14)
plt.yticks([], fontsize = 14)

for bar in profit_plot.patches:
    profit_plot.annotate(round(bar.get_height()), (bar.get_x() + bar.get_width() / 2,
                bar.get_height() - 15), ha = 'center', va = 'center',
                xytext = (0, 9), textcoords = 'offset points')
```

A.5 Data pre-processing

Importing packages.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_validate, GridSearchCV
from sklearn.preprocessing import StandardScaler
import sklearn.discriminant_analysis as skl_da
from sklearn.metrics import accuracy_score, classification_report
from sklearn.pipeline import Pipeline
```

Creating Figure 4 for feature correlation.

```
import re
plt.figure(figsize=(12, 5))
sns.heatmap(train_data.corr(), cmap=sns.color_palette("vlag", as_cmap=True))

plt.title('Correlation between features', fontsize = 16)
xlabels = [re.sub("(.{25})", "\\1\n", label, 0, re.DOTALL) for label in train_data.columns]
plt.xticks(range(13), xlabels, fontsize=14)
plt.yticks(fontsize=14)
```

Train-test split.

```
df = pd.read_csv('train.csv')
X = df.drop(['Total words', 'Lead'], axis=1)
y = df['Lead']
train_X, test_X, train_y, test_y = train_test_split(X, y,
                test_size=0.1, random_state=42, stratify = train_data['Lead'])
```

Feature scaling.

```
scaler = StandardScaler()
scaler.fit(train_X)
train_X = scaler.transform(train_X)
test_X = scaler.transform(test_X)
```

A.6 LDA

```
#LDA 'svd' model training
model_lda = skl_da.LinearDiscriminantAnalysis(solver='svd')
model_lda.fit(train_X, train_y)

#Accuracy 'svd' LDA
pred_y_lda = model_lda.predict(test_X)
print('Accuracy "svd":',accuracy_score(test_y, pred_y_lda))

#Pipeline
operations_lda = [('scaler', scaler), ('model_lda', model_lda)]
pipeline_lda = Pipeline(steps = operations_lda)

#GridSeachCV
solver = ['lsqr', 'eigen']
shrinkage = ['auto']

param_grid_lda = {'model_lda__solver':solver, 'model_lda__shrinkage':shrinkage}
cv_classifier_lda = GridSearchCV(pipeline_lda, param_grid_lda, cv=12, scoring='accuracy')
cv_classifier_lda.fit(train_X, train_y)
cv_classifier_lda.best_estimator_.get_params()
```

```
pred_y_lda_cv = cv_classifier_lda.predict(test_X)

#Accuracy GridSearchCV LDA
print(classification_report(test_y, pred_y_lda_cv))
print('Accuracy GridSearchSV:', accuracy_score(test_y, pred_y_lda_cv))
```

A.7 QDA

```
#QDA model training
model_qda = skl_da.QuadraticDiscriminantAnalysis()
model_qda.fit(train_X, train_y)

#Pipeline
operations_qda = [('scaler', scaler), ('model_qda', model_qda)]
pipeline_qda = Pipeline(steps = operations_qda)

#GridSeachCV
reg_param = [0.0, 0.1, 0.2, 0.3, 0.5, 0.8, 1.0]

param_grid_qda = {'model_qda__reg_param':reg_param}
cv_classifier_qda = GridSearchCV(pipeline_qda, param_grid_qda, cv=12, scoring='accuracy')
cv_classifier_qda.fit(train_X, train_y)
cv_classifier_qda.best_estimator_.get_params()
pred_y_qda_cv = cv_classifier_qda.predict(test_X)

#Acuuracy GridSearchCV QDA
print(classification_report(test_y, pred_y_qda_cv))
print('Accuracy GridSearchSV:', accuracy_score(test_y, pred_y_qda_cv))
```

A.8 KNN

```
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier

X_new = df.drop(['Total words', 'Lead'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(X_new, y, test_size=0.1, random_state=57)
scaler = StandardScaler()
knn = KNeighborsClassifier()

operations = [('scaler', scaler), ('knn', knn)]
pipeline = Pipeline(steps=operations)
k_values = range(1, 30)
weights = ['uniform', 'distance']
metrics = ['minkowski', 'manhattan', 'euclidean']

param_grid = {'knn__n_neighbors':k_values, 'knn__metric':metrics, 'knn__weights': weights }

# Create a classifier with 12 fold cross-validation using the pipeline and param_grid
cv_classifier = GridSearchCV(pipeline, param_grid, cv=12, scoring='accuracy')

#Fit the model to the training data, i.e learn a model from the data
cv_classifier.fit(X_train, y_train)

from sklearn.metrics import confusion_matrix

#Predict X_test with the best parameters for the model and display the accuracy and confusion matrix
y_pred = cv_classifier.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(f"Accuracy: {np.mean(y_pred == y_test):.3f}")
```

A.9 Logistic Regression

```
# Create train-test-split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

X_new = df.drop(['Total words', 'Lead'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(X_new, y, test_size=0.1, random_state=57)

#Parameters for logistic regressor
c = [2.0, 1.0, 0.5, 0.25, 0.0625]
fit_intercept = [True, False]

param_grid = {"lr__C": c, "lr__fit_intercept": fit_intercept}

# Create a classifier with 10 fold cross-validation using the pipeline and param_grid
cv_classifier = GridSearchCV(pipeline, param_grid, scoring='accuracy', cv=10)

cv_classifier.fit(X_train, y_train)

y_pred = cv_classifier.predict(X_test)

print(classification_report(y_test, y_pred))
print(f"Accuracy: {np.mean(y_pred == y_test):.3f}")
```

A.10 Random forest

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=57)

rf = RandomForestClassifier()

operations = [('scaler', scaler), ('rf', rf)]
pipeline = Pipeline(steps=operations)

criterion = ["gini", "entropy"]
min_samples_split = [2, 3, 4, 5, 7, 9]
max_features = ["sqrt", "log2"]

param_grid = {"rf__criterion": criterion,
              "rf__min_samples_split": min_samples_split,
              "rf__max_features": max_features}

rf_cv_classifier = GridSearchCV(pipeline, param_grid,
                            scoring='accuracy', cv=10)

y_pred = rf_cv_classifier.predict(X_test)
print(classification_report(y_test, y_pred))
```

A.11 Gradient boosting

```
GBC = GradientBoostingClassifier()

#GridSearchCV parameters
parameters = {
    'learning_rate': [0.01,0.02,0.03,0.05],
    'subsample' : [1.0, 0.9, 0.5, 0.2],
    'n_estimators' : [100,500,1000,2000],
    'max_depth' : [3,4,6,8]
    }
```

```
grid_GBC = GridSearchCV(estimator=GBC, param_grid = parameters, cv = 12, n_jobs=-1)
grid_GBC.fit(x_train, y_train)

print("Optimal Grid Search values" )
print("\n The best model with searched params:\n",grid_GBC.best_estimator_)
print("\n The best score among all searched params:\n",grid_GBC.best_score_)
print("\n The best parameters from all searched params:\n",grid_GBC.best_params_)

pred_gbc = grid_GBC.predict(x_test)

#Accuracy GridSearchCV GradientBoosting
print("Accuracy GridSearchSV:", accuracy_score(y_test, pred_gbc))
```

A.12 Adaptive boosting

```
ABC = AdaBoostClassifier()
#GridSearchCV parameters
parameters_ada = {
    'n_estimators' : [2000, 5000],
    'learning_rate': [0.03, 0.05, 0.1],
    'algorithm' : ["SAMME", "SAMME.R"]
    }

grid_ABC = GridSearchCV(estimator=ABC, param_grid = parameters_ada, cv = 12, n_jobs=-1)
grid_ABC.fit(x_train, y_train)

print("Optimal Grid Search values" )
print("\n The best model with searched params:\n",grid_ABC.best_estimator_)
print("\n The best score among all searched params:\n",grid_ABC.best_score_)
print("\n The best parameters from all searched params:\n",grid_ABC.best_params_)

pred_abc = grid_ABC.predict(x_test)

#Accuracy GridSearchCV GradientBoosting
print("Accuracy GridSearchSV:", accuracy_score(y_test, pred_abc))
```

A.13 Extreme gradient boosting

```
from xgboost import XGBClassifier
XGBC = XGBClassifier()
#GridSearchCV parameters
parameters_xgb = {
    'n_estimators' : [100,500,1000],
    'learning_rate': [0.01,0.02,0.03],
    'grow_policy' : ["depthwise", "lossguide"],
    'booster ' : ["gbtree", "gblinear", "dart"]
    }

grid_XGBC = GridSearchCV(estimator=XGBC, param_grid = parameters_xgb, cv = 12, n_jobs=-1)
grid_XGBC.fit(x_train, y_train)

print("Optimal Grid Search values" )
print("\n The best model with searched params:\n",grid_XGBC.best_estimator_)
print("\n The best score among all searched params:\n",grid_XGBC.best_score_)
print("\n The best parameters from all searched params:\n",grid_XGBC.best_params_)

pred_XGBC = grid_XGBC.predict(x_test)

#Accuracy GridSearchCV GradientBoosting
print("Accuracy GridSearchSV:", accuracy_score(y_test, pred_XGBC))
```

A.14 Deep neural networks

```
    from sklearn.neural_network import MLPClassifier
    from sklearn.metrics import classification_report
    from sklearn.model_selection import GridSearchCV

    df = pd.read_csv('train.csv')
    X = df.drop(['Total words', 'Lead'], axis=1)
    y = df['Lead']

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=52)
    scaler.fit(X_train)
    X_train = scaler.transform(X_train)
    X_test = scaler.transform(X_test)

    clf = MLPClassifier(solver='sgd', alpha=1e-5,
                  hidden_layer_sizes=(len(X_train[0]), len(X_train[0])+2, len(X_train[0])+2),
                  random_state=1,
                  max_iter=15000,
                  activation='tanh'
                 )
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(classification_report(y_test, y_pred))
```

A.15 Naive classfier

```
    naive_y = np.full((104,),'Male')
    naive_accuracy = ((naive_y == test_y)*1).mean()
```