Do (wo)men talk too much in films?

Gabriel Y. Arteaga
Uppsala University
`gabrielyanci.arteaga.6822@student.uu.se`

Jakob Nyström
Uppsala University
`jakob.nystrom.5563@student.uu.se`

Inga Wohlert
Uppsala University
`inga.wohlert.8507@student.uu.se`

Alexander Sabelström
Uppsala University
`alexander.sabelstrom.1040@student.uu.se`

Abstract

Gender imbalance in the movie industry has received significant attention in recent years. In this report, we quantitatively answer some key questions related to this topic, using a dataset of Hollywood movies. We also aim to classify the lead actor gender based on movie attributes. The performance of four model families is evaluated. Among these, QDA performs best overall, with an average accuracy of 0.925 in cross-validation. It also has the most balanced accuracy between genders.

1 Introduction

We have access to a dataset of 1,039 Hollywood movies from 1939 to 2015, based on script analysis and IMDB data [1]. We begin by investigating what gender dominates speaking roles and how this has developed over time. For the classification task, if the lead actor in the data is male, the co-lead is female, and vice versa. Hence it boils down to binary classification of the gender of the main lead. We explore and evaluate the performance of the following classification methods on the problem: *logistic regression*; *discriminant analysis* (LDA and QDA); *k nearest neighbors* (kNN); and *boosting*.

2 Data analysis

To make sure the data is in good order, we perform a few quality checks before doing any analysis or modeling. This includes checking for NaNs, zeros and negative values, which reveals no issues with the data. Before 1972 the data is not continuous over years and there are only 12 observations. For consistency and relevance, we therefore restrict all analysis in section 2.1 to the period 1972-2015.

2.1 Key questions about gender imbalance in movies

Do men or women dominate speaking roles in Hollywood movies? Overall, males dominate when looking across all years in scope. Males constituted 68% of all actors with speaking roles (see Figure 1, 1st plot). They also held about 76% of the lead roles. Females only accounted for 21% of words spoken (Figure 1, 2nd plot).

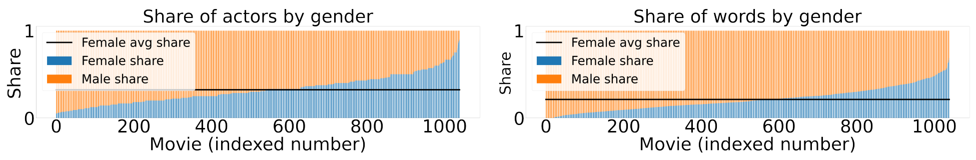

Figure 1: Share of actors and words by gender

Has gender balance in speaking roles changed over time? Balance has increased for share of actors (female share from 22% in 1972 to 45% in 2015; see Figure 2, 1st plot) and lead roles (from 0% in 1972 and 25% in 1973 to 53% in 2015; see Figure 2, 2nd plot). The share of female words has increased from 16% to 24%, but females are still highly underrepresented (Figure 2, 3rd plot).

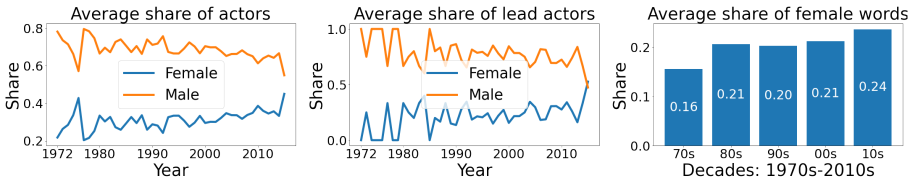

Figure 2: Share of actors, leads and words by gender over time

Do films in which men do more speaking make a lot more money than films in which women speak more? Average earnings were 39% higher for movies with a majority of male words (see Figure 3, 1st plot). Movies with male leads earned about 23% more (see Figure 3, 2nd plot).

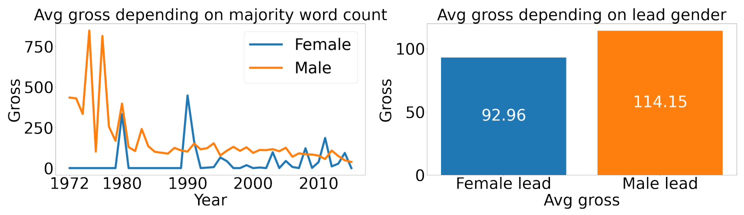

Figure 3: Earnings (gross) by gender that had majority of words, and lead role

2.2 Feature distributions by class to identify important features for modeling

Note that for the rest of the report, data from all years is included, to match the expected distribution of the external validation set. For the classification task, we want to use the features that best discriminate between the two classes. This involves identifying features whose class-dependent distributions are significantly different. First, we compare distributions using histograms and box plots (examples in Figure 4, 1st and 2nd plot).

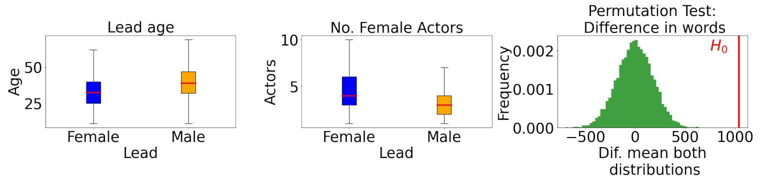

Figure 4: Examples of class-dependent distributions that are different

To substantiate this analysis we perform permutation tests; one example in Figure 4 (3rd plot). This is a hypothesis test with the null hypothesis, H_0, that class-dependent features are drawn from the same distribution. H_0, calculated through our test statistics, is rejected at a 95% confidence level [2] B.2. A summary of suggested feature importances is listed below. However, this does not need to imply that these specific sets of features will lead to the best performance, as this looks at features one-by-one, not taken together.

- **Manual inspection:** Number words lead; Difference in words lead & co-lead; Number of male actors; Number of female actors; Age lead; Age co-lead
- **Permutation test:** Number words lead; Difference in words lead & co-lead; Number of male actors; Number of female actors; Age lead; Age co-lead; Total words; Year

3 General modeling approach: feature selection, tuning and evaluation

The approach to finding the best model within and between model families is illustrated in Figure 5. We start with two benchmark feature selections for all model families: one with all original features (called "basic" in tables further down) and one with the features suggested by the permutation test ("select"). From that starting point we try to improve within each family. In feature engineering, around 40 new features are created to extract more information from the data. For consistency across families, we build a common Python class for this, that includes shares (e.g. *Female word share*), aggregations (e.g. *Decade*) and differences (e.g. *Age lead vs co-lead*). We then prune our feature sets, to avoid overfitting and unnecessary complexity. Feature selection is done individually per family, since our aim is to find the best model in terms of performance, rather than benchmarking all models using exactly the same features. Manual and iterative feature selection is complemented by feature selection packages [3; 4; 5; 6; 7]. This is especially useful when selecting among a large number of features. A detailed explanation of each selection process is found in the respective model sections.

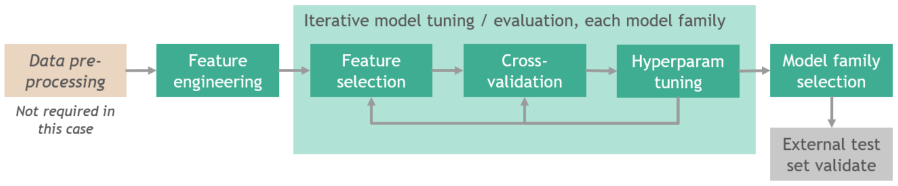

Figure 5: Sequential and iterative modeling approach

We leverage shared cross-validation (CV) code for comparison of models, using the entire dataset as input. Following the standard CV procedure, for each of the five folds indexed by k, we train on the other four folds, and use the k^{th} fold as the test set.[1] Data is scaled or normalized before fitting, to help computations and improve results where relative size of inputs matters (e.g. kNN) [8; 9]. Since the problem is symmetric – we want to classify both genders as accurately as possible – overall test accuracy is the most important metric. This includes both the average and min-max accuracy across the five test folds. Accuracy is also calculated for each gender, to detect gender bias in the model, enabled by the custom-built CV code. We further output training accuracy; if this is substantially higher than test accuracy, it indicates overfitting.

4 Application of different classification methods

4.1 Naive base classifier

To provide a benchmark, we create a naive base classifier, that always predicts *Male* as lead. It has an accuracy of 0.756 on our data set, which is simply the proportion of male lead roles in the data.

4.2 Logistic regression

In logistic regression, we attempt to learn the parameters θ in the linear expression $x^T\theta$, that minimize the log loss of the training data points. A sigmoid function is applied to transform the linear expression to a probability between 0 and 1, with a standard choice being [10]:

$$\frac{e^{x^T\theta}}{1+e^{x^T\theta}} \tag{1}$$

which has a range between 0 and 1. In binary logistic regression, we classify all data points with probability greater than 0.5 to one class and the rest to the other. The log loss is given by [10]:

$$\hat{\theta} = arg\,\underset{\theta}{min}\,\frac{1}{n}\sum_{i=1}^{n} ln(1+e^{-y_i\theta^T x_i}) \tag{2}$$

where a regularization term could be added to the loss function that we are minimizing. To learn the parameters θ it is possible to use different numerical solvers, e.g. gradient descent.

[1] Five folds is a reasonable choice for our medium-sized data set. We get roughly 208 observations in each fold, so each test fold should be representative of the underlying distribution.

4.2.1 Application to the data and results

There are several ways of finding the best model which have the parameters that minimises the loss, and the performance of these methods can vary. Feature engineering was realized to be important for logistic regression due to the performance not changing much depending on which hyperparameters used. Recursive feature elimination [7] was tested in order to select the best features to use for the best model. When testing different sizes of subsets, 14 features selected by recursive feature elimination were found to be most optimal.

Hyperparameter optimization was then implemented with `sklearn's` `GridSearchCV`. It permutes the parameters to be optimized, and find the combination that results in the greatest training accuracy. There are three key hyperparameters in logistic regression; the `C` hyperparameter, the `penalty` hyperparameter and which solver to use. `C` is the inverse of the regularization strength, which means that the use of a large `C` value results in weaker regularization. The `penalty` parameter is which kind of regularization to use in order to avoid overfitting the model. Weights were also tested in order to hyper-parameterize the model because of the over representation of males in the data, but was realized to not improve the overall accuracy. Many of the parameters were performing equally well, but for the solvers, 'newton-cg' and 'liblinear' were on average performing better, along with lasso and ridge for penalties and a low `C`-value.[2]

Table 1: Logistic Regression, cross-validation performance

Metric	LR basic	LR select	LR best
Avg. accuracy	.870	.807	.889
Acc. fem / male	.618 / .954	.437 / .927	.681 / .957
Train accuracy	.880	.804	.890

4.3 Discriminant analysis

Most classification models try to model the conditional $p(y \mid \mathbf{x})$. In discriminant analysis, we instead model the joint distribution $p(\mathbf{x}, y)$, using the factorization $p(\mathbf{x}, y) = p(\mathbf{x} \mid y)\, p(y)$. That is, the probability of observing a set of features $\mathbf{x}$ for a class y, and the probability of y. In our binary classification task, the marginal probability of y is given by $p(y = 1) = \pi_1$, $p(y = 2) = \pi_2$. For the class-conditional distribution we have the key assumption that $p(\mathbf{x} \mid y)$ is Gaussian:

$$p(\mathbf{x} \mid y) \sim \mathcal{N}(\mathbf{x} \mid \mu_{\mathbf{y}}, \mathbf{\Sigma_y}) \tag{3}$$

where $\mu_{\mathbf{y}}$ is a class-dependent vector of mean values for $\mathbf{x}$ and $\mathbf{\Sigma_y}$ is the class-dependent covariance matrix of $\mathbf{x}$. To learn $\theta = \{\mu_{\mathbf{y}}, \mathbf{\Sigma_y}, \pi_{\mathbf{y}}\}$ we maximize the log-likelihood function. This optimization problem has a closed-form solution, where $\mu_{\mathbf{y}}$ and $\mathbf{\Sigma_y}$ take values that best fit the data $\mathbf{x}$ for each class. For prediction, we use the definition of conditional probability to derive the classifier [10].

In quadratic discriminant analysis (QDA), the decision boundary that separates the classes is quadratic. If we make the simplifying assumption that the covariance matrix $\mathbf{\Sigma_y}$ is the same for all classes, we can derive linear discriminant analysis (LDA), which has a linear boundary. QDA has higher flexibility and capacity, whereas LDA is more robust to overfitting [10].

4.3.1 Application to the data and results

The assumption in (3) is evaluated using the Henze-Zirkler test [11; 12], which tests if the class-conditional $\mathbf{x}$ matrices follow a multivariate Gaussian distribution. The null hypothesis of normality is rejected at 95% confidence for both classes. This result urges some caution, but in practice LDA / QDA is useful even if the strict Gaussian assumption does not hold [10], as we will also see below.

LDA and QDA estimators from `sklearn` [13; 14] are used. Results are in Table 2. With the given features ("basic"), LDA performs quite well, while QDA is struggling with co-linearity and

[2]An example of a top performing model had the parameters: $C \approx 0.62$, $penalty = l2(ridgeregression)$, $solver = newton - cg$.

performs much worse. This underlines the need for careful feature selection in QDA. Using only the permutation testing features ("select") improves performance significantly with QDA (LDA performs the same as before). Interestingly, including *Number female words*, which was not indicated as significant by the permutation test, increases accuracy to 0.893. This shows that such tests for individual features can be too simplified.

Next, we include some engineered features, e.g. *Share of female words* and *Decade*. This raises accuracy to 0.925 for QDA ("best" column), with relatively good results also for the *Female* class. Training accuracy is only slightly higher, so the model does not seem to overfit. Overall, once we start tuning, QDA performs better than LDA due to its higher flexibility.[3] This is true as long as there are not too many features and they are not highly co-linear. The only hyperparameter in the `sklearn` QDA implementation is regularization of the covariance matrix. Applying this only leads to worse test performance. Finally, we run a Sequential Feature Selection (SFS) algorithm [6] to compare with the current best features. This performs well, although not quite as good, but does so with only 5 features, compared to 13. The main issue here is the lower *Female* accuracy.

Table 2: Discriminant analysis, CV performance

Metric	LDA basic	QDA basic	QDA select	QDA SFS	QDA best
Avg. accuracy	.869	.711	.801	.908	.925
Acc. fem / male	.59 / .959	.76 / .696	.461 / .911	.748 / .959	.831 / .955
Train accuracy	.868	.727	.812	.913	.935

4.4 K-nearest neighbors

Unlike the other methods, kNN is a non-parametric and lazy learning model. Classification is based on the distance of a new data point to points in the training data, under the assumption that points that are close are also most similar. Classification is determined by majority vote among these nearest neighbours. The amount of neighbours in each neighborhood is determined by the hyperparameter *k*. There is a high risk that kNN has too much flexibility and thus overfits, when choosing a too small *k* for the problem. Since classification is based on distance between points, the relative size of features also plays an important role. Thus scaling or normalization of the data is crucial. In addition, with increasing dimensionality the neighbor proximity assumption can also break, as data becomes sparse. It also becomes computationally more expensive to compute the distances [10].

For *k* we usually choose an odd number for an even number of classes and vice versa. This is necessary for majority voting to be guaranteed to work. However, we can alternatively add the distance as a weight in the voting to accomplish the same thing. Further, we have to choose a distance measure. The most commonly used ones are Manhattan, Euclidean and Chebychev. All of these can be derived from the Minkowski distance ($D(X,Y) = (\sum_{i=1}^{n} |x_i - y_i|^p)^{\frac{1}{p}}$), by changing the *p* parameter [10; 15].

4.4.1 Application to the data and results

We use the `sklearn` kNN classifier [16]. As a first step we chose standard scaling, while keeping all features.[3] We can see that it is mainly predicting *male* for the lead, ("basic") in Table 3. Next we tweaked the hyperparameters (*k*, distance measure and weights). For *k* we tested a range from 0-30, adding weights and the different distance measures of the kNN classifier. Since this has to be redone for each new feature selection, we wrote a script that tests different combinations of the hyperparameters and then uses our cross validation to estimate the average accuracy. Given that kNN works better with fewer dimensions, we then proceeded with our feature selection by seeing which attributes affected the performance the most, like in recursive feature selection or backward elimination, but manually. We evaluated different methods of scaling, where standard scaling and normalizing turned out to increase accuracy the most. However, our model was still overfitting.

We then created features with our feature engineering. Often they managed to portray all the relevant information but in a scaled manner. We then used `mlxtend's` SFS, with an established good

[3]We tested every model specification using both LDA and QDA, but not all LDA results are reported here

dimension [3; 5]. Reducing the amount of features led to the final selection with *k* = 6, Chebyshev distance, no weights (uniform) and standard scaling. Surprisingly an even *k* produces the best outcome.[4] It seems that the added randomness helps with classifying female lead compared to an uneven *k* (*k*=7) and therefore is our best model.

Table 3: kNN, cross-validation performance

Metric	kNN basic (k=13)	kNN select (k=14)	kNN best (k=7)	kNN best (k=6)
Avg. Accuracy	.784	.793	.880	.885
Acc fem/male	.205/ .972	.315/.948	.611/.967	.682/.952
Train accuracy	1.	1.	.900	.912

4.5 Boosting

The main idea behind boosting is to train weak learners and combine them to create a strong learner that can accurately predict the target variable. Adaptive boosting (AB) uses an exponential loss function (ELF):

$$L(y \cdot f(x)) = e^{-y \cdot f(x)}$$

which is used to calculate the weights for the next instance in the boosting sequence [10]. The final prediction which is the output from AB is the weighted majority vote:

$$\hat{y}^{(B)}_{boost}(X) = sign\left\{\sum_{b=1}^{B} \alpha^{(b)} \hat{y}^{(b)}(X)\right.$$

where α is the coefficient which describes the degree of confidence for the b^{th} ensemble member. Since AB uses ELF it is sensitive to noise and data with high uncertainty in the true input-output relationship [10]. To mitigate the potential drawback of ELF we use gradient boosting (GB). The idea behind GB is to compute the gradient of a differentiable cost function, e.g. a logistic loss function (2). We want to reduce the value of that cost function without requiring a greedy minimization. Each ensemble member is then trained with the objective that its predictions should be close to the negative gradient, were closeness can be evaluated by a suitable distance function e.g. *squared distance* [10].

4.5.1 Application to the data and results

We use tree classifiers for both boosting approaches. The adaptive and gradient boosters chosen are AdaBoost (Ada) [8; 18; 19] and XGBoost (XGB) [20; 21] respectively. In Table 4 we can conclude that the initial select of features barely improves Ada and decreases the performance of XGB compared to the "basic" feature set. Further, we notice that XGB with default parameters under "Basic" and "Select" in Table 4 are particularly prone to overfit.

We keep the default hyperparameters for Ada since it does not overfit. However, for XGB an initial tuning is required to prevent overfitting. We then proceed with our general modelling approach, were we apply our feature engineering techniques.[3] Scaling does neither improve nor deteriorate the algorithms performance, hence, the boosters seem to be insensitive to distance. We proceed with applying our feature engineering techniques.[3] Feature selection is done through `mlxtend's` SFS for both ADA and XGB [3; 5]. For XGB we use SFS to first prune the features and exhaustive feature selection to then replace features with ones that increase performance [3; 5; 4]. The feature selection gives us feature sets consisting of 12 features for Ada and XGB respectively.

Lastly, we perform hyperparameter tuning again to improve performance. Both Ada and XGB are prone to overfitting, thus, we are faced with a trade-off between overfitting and increase of accuracy. For Ada we optimize the tree depth, number of estimators and learning rate. Since XGB have a wider selection of hyperparameters which allows regularization to reduce overfitting, we decide to put more effort into it. Due to computational constrains we use `RandomizedSearchCV` [8; 22], where we tune hyperparameters which increase accuracy e.g. (*learning rate, max depth, number of estimators*) and reduce overfitting e.g. (*gamma, subsample, colsample bytree*). We gradually concentrate the range of parameter values to search through which allows us to find a good combination.

[4]In case of a tie, the sklearn classifier will pick one, based on skipy.stat mode. [16] [17]

Table 4: Boosting, cross-validation performance

Metric	Ada Basic	Ada Select	Ada Best	XGB Basic	XGB Select	XGB Best
Avg. Accuracy	.858	.859	.888	.879	.873	.911
Acc fem/male	.607/.939	.622/.936	.685/.954	.646/.954	.658/.943	.709/.977
Train accuracy	.917	.911	.987	1	1	.984

5 Discussion and conclusions

Our task is to find the model that generalizes best to new data, assuming this comes from the same distribution as the training data, i.e. the one with lowest $\mathbb{E}_*[\mathbb{1}\{y_* \neq f(x_*;\theta)\}]$. A strong candidate model should have **1)** high average accuracy in cross-validation; **2)** small spread (the difference in performance between tests on different folds); **3)** balanced accuracy for both classes; **4)** low difference between train and test accuracy; **5)** simplicity in terms of number of features. Such a model can even be robust to small distribution shifts. Table 5 compares the best models across families.

Table 5: Best models from each family, and naive classifier

Metric	Log. reg.	QDA	kNN	XGB	Naive
Avg. accuracy	.889	.925	.885	.911	.756
Acc., min-max	.861 - 899	.885 - .955	.875 - .908	.885 - .947	n/a
Acc. fem / male	.681 / .957	.831 / .952	.682 / .952	.709 / .977	1. / .0
Train accuracy	.890	.935	.912	.984	n/a
N features	14	13	5	12	n/a

When evaluating the results in light the criteria we can see that

1. More flexible, non-linear models (QDA, boosting) produce better overall results than simpler, linear ones (logistic regression, LDA) and kNN. QDA has the best family member overall. All models perform significantly better than the naive classifier.
2. There is little spread for all families, indicating that results are robust for different samples.
3. All models are better at predicting male leads. This is natural since this class makes up about 3/4 of observations. In terms of female accuracy, QDA performs the best.
4. XGB is overfitting to training data, despite generalizing well, potentially due to the small dataset. Logistic regression, QDA and kNN exhibit very little overfitting.
5. QDA and XGB are of similar complexity in terms of features, whereas kNN delivers good performance with only 5 features, which is an advantage of that model family.

Based on this evaluation we decide to use QDA as the "production" model (all features found in [A.1]). We would like to provide some confidence interval for its test accuracy. But since the folds are not independent, we cannot use concentration inequalities for this. As a proxy, we run 1,000 random train-test splits (20% of data in test), from which we get an interval of (.865, .966) [A.2].

Some common themes can also be noted. First, there is a set of features that were important in several models, e.g. *Number words female, Number of words lead, Difference in words lead and co-lead*. It is not surprising that there are such core attributes that represent something fundamental about the problem. On the contrary *Gross* is not an informative feature. However, the marginal value of additional features varies by model family. Second, feature engineering and selection generally seems to add more value than hyperparameter tuning. While working with the problem, there were some interesting insights. It seems that our inspection of class-dependent distributions was as good or better than using permutation tests. However, both pointed out attributes which contained valued information, further improved with some engineered features. Similarly for feature selection, automated packages can be of help, but there is also a great deal of handcrafting needed to get the best possible performance. Due to computational constrains, it is impossible to explore every combination to find the *"best"* possible model. However, we have been able to identify a sufficiently high-performing model that can be put in production.

References

[1] "Film dialogue from 2,000 screenplays, broken down by gender and age," https://pudding.cool/2017/03/film-dialogue/, accessed: 2022-12-20.

[2] Haneul Kim, "Monte Carlo method: permutation test," https://haneulkim.medium.com/monte-carlo-method-permutation-test-99e2a9c46f52, 2021, 08.12.2022.

[3] S. Raschka, "Mlxtend: Providing machine learning and data science utilities and extensions to python's scientific computing stack," *The Journal of Open Source Software*, vol. 3, no. 24, apr 2018. [Online]. Available: http://joss.theoj.org/papers/10.21105/joss.00638

[4] "ExhaustiveFeatureSelector," http://rasbt.github.io/mlxtend/user_guide/feature_selection/ExhaustiveFeatureSelector/, accessed: 2022-12-08.

[5] "SequentialFeatureSelector," http://rasbt.github.io/mlxtend/user_guide/feature_selection/SequentialFeatureSelector/, accessed: 2022-12-08.

[6] "sklearn.feature_selection.SequentialFeatureSelector," https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SequentialFeatureSelector.html, accessed: 2022-12-09.

[7] "sklearn.feature_selection.RFE," https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html, accessed: 2022-12-08.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[9] "sklearn.preprocessing," https://scikit-learn.org/stable/modules/preprocessing.html, accessed: 2022-12-21.

[10] A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön, *Machine Learning: A First Course for Engineers and Scientists*. Cambridge University Press, 2022.

[11] "Henze-Zirkler test," https://pingouin-stats.org/generated/pingouin.multivariate_normality.html#pingouin.multivariate_normality, accessed: 2022-12-22.

[12] "How to Perform Multivariate Normality Tests in Python," https://www.statology.org/multivariate-normality-test-python/, accessed: 2022-12-22.

[13] "sklearn.discriminant_analysis.LinearDiscriminantAnalysis," https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html, accessed: 2022-12-08.

[14] "sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis," https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis.html, accessed: 2022-12-08.

[15] "Minkowski distance," https://en.wikipedia.org/wiki/Minkowski_distance, 08.12.2022.

[16] "sklearn.neighbors.KNeighborsClassifier," https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html, accessed: 2022-12-09.

[17] "scipy.stats.mode," https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mode.html#scipy.stats.mode, accessed: 2022-12-09.

[18] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[19] "sklearn.ensemble.AdaBoostClassifier," https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html, accessed: 2022-12-09.

[20] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[21] “XGBoost Documentation,” https://xgboost.readthedocs.io/en/stable/, accessed: 2022-12-09.

[22] “sklearn.model_selection.RandomizedSearchCV,” https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html, accessed: 2022-12-09.

A Supplementary exhibits for Discussion and conclusions

A.1 Selected features of best models

Table 6: Features of best model from each model family

Logistic regression	QDA
’Difference in words lead and co-lead	’Total words’
’Number of female actors’	’Number words female’
’Number words male	’Female word share’
’Mean Age Male’	’Number of words lead’
’Age Lead’	’Lead word share’
’Age Co-Lead’	’Difference in words lead and cc-lead’
’Female word share’	’Total actors’
’Male word share’	’Female actor share’
’Lead word share’	’Age Lead’
’Female actor share’	’Age Co-Lead’
’Male actor share’	’Mean Age Male’
’Difference Age Lead’	’Mean Age Female’
’Yearly mean diff Number of words lead’	’Decade numeric’
’Yearly mean diff Difference in words lead and co-lead’	
’Number of words lead’	
kNN	**XGB**
’Difference in words lead and co-lead’	’Number of words lead’
’Female word share’	’Difference in words lead and co-lead’
’Male actor share’	’Number of female actors’
’Yearly mean diff Number of words lead’	’Age Co-Lead’
’Yearly mean diff Difference in words lead and co-lead’	’Female word share’
	’Male word share’
	’Lead word share’
	’Female actor share’
	’Difference Actors’
	’Difference Age Lead’
	’Difference Mean Age’
	’Yearly mean diff Number of male actors’

A.2 Test accuracy distribution for QDA

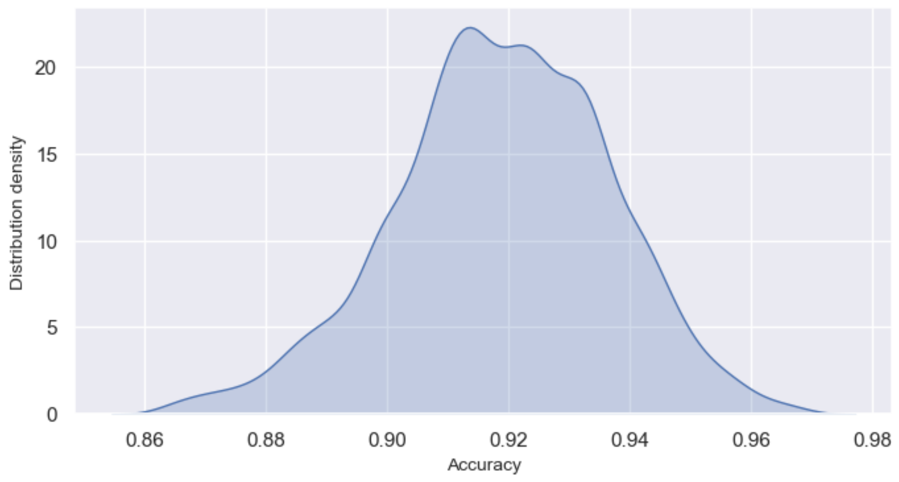

Figure 6: QDA model: Distribution of test accuracy across 1,000 train-test samples

B External packages and other methods

In this section we will go in to further detail of what the functions we have used does.

B.1 Feature selection

We used several types of feature selection functions which will be explained more detailed in the following section.

B.1.1 Sequential feature selection

Sequential feature selection is a greedy algorithm which attempts to iteratively reduce a d-dimensional to a k-dimensional feature subspace [3]. The number of k dimensions is a hyperparameter which is determined by the engineer. We chose to use both forward selection and backward elimination.

In forward selection, we start with an empty set of features then we sequentially add a feature at a time. Which feature to add is chosen through cross validation with 5 folds were accuracy is the scoring metric chosen to evaluate performance. The feature selection algorithm stops once k features have been selected.

In backward elimination we instead start with a full feature set and then sequentially remove the least important feature until we reach the desired k feature set.

B.1.2 Exhaustive feature selection

Exhaustive feature selection is a greedy method which tries out all combinations of features until the best feature set has been found. It does this through cross validation together with a scoring metric e.g. accuracy. It is a very computationally heavy task since the formula for all features which needs to be evaluated looks like this:

$$\frac{n!}{k!(n-k)!}$$

where n is the total number of features to choose from and k is the desired amount of features to be left with once the feature selection has been executed.

Since exhaustive feature selection is such a computationally heavy task we only used it after first pruning the feature set with forward feature selection. We then manually performed backward elimination to then use exhaustive feature selection as a forward feature selector. We fix a set of features and exhaustively search for a feature with more importance than the one which had been removed previously.

B.1.3 Recursive feature eliminiation

The goal of recursive feature elimination or RFE is to find the features best fit for the model. It achieves that by starting with all the features given, and then recursively looking through subsets of the features, removing undesired features until the number of features wished to be used is reached. A feature is seen as undesired if the fitted model is worse with the feature in question, and that is usually decided by obtaining the model's "coef" or "feature importance" attributes [7]. When a feature is removed, the model is refitted and tested again.

B.2 Permutation testing

In our Data Analysis we are using Permutation Testing in order to test whether the two different distributions "male" and "female" lead are coming from the same underlying distribution for each variable. Our null hypothesis is that both distributions are actually coming from the same distribution. First, a test statistic is calculated from the original data.In our case we used the mean between the two different distributions as the test statistics. The data is then shuffled and the test statistic is calculated with the shuffled data. According to the Monte Carlo method, this process is repeated many times, in our case 10 000 times. Finally, the test statistics from the shuffled data are compared to the original. If the original is outside the confidence interval, the null hypothesis is rejected. As our sample distribution is normally distributed, due it being means, we can use the empirical rule to calculate our confidence interval.

B.3 Henze-Zirkler

The Henze-Zirkler test is a test of normality for multivariate distributions [11; 12]. That makes it different from most normality tests, which can only test the normality assumption one feature at a time. This means that it is suitable for evaluating the following assumption in discriminant analysis:

$$p(\mathbf{x} \mid y) \sim \mathcal{N}(\mathbf{x} \mid \mu_{\mathbf{y}}, \mathbf{\Sigma}_{\mathbf{y}})$$

In our case, we are testing the multivariate feature distribution for each of the two classes (genders). The null hypothesis of the test is that the class-dependent data sample comes from a normal distribution. The null hypothesis is rejected if the p-value is lower than the chosen alpha (significance level). That means there is evidence that the sampled data does not come from a normal distribution.

B.4 Hyperparameter tuning

B.4.1 GridSearchCV

Grid search permutes the parameters wished to be optimized, and find the combination that results in the greatest training accuracy, it is therefore an exhaustive search alternative. For numerical variables, a search range is given, and for different methods such as which penalty to choose, the different penalties are given. GridSearchCV then applies cross validation on every possible combination, and the combination with the highest accuracy is chosen as the optimal one.

B.4.2 RandomizedSearchCV

RandomizedSearchCV is an alternative to grid search, which instead of exhaustively search through a predefined set of hyperparameters, it will randomly sample sets of hyperparameters to search through [22]. This is particularly useful when the search space is large or the model is computationally heavy to train. The drawback with RandomizedSearchCV is that you probably will not find a global minima, instead you will get stuck in a local minima.

To use RandomizedSearchCV, you need to specify the estimator which is the model you wish to optimize, the parameter distribution were you specify which range of parameters you want to search through, and the number of iterations you want to run. A scoring function is used for evaluating the performance of the model, the scoring functions used were *"accuracy"* and *"roc_auc"*. The accuracy scoring function evaluates the performance of the model in regards to the accuracy metric. Meanwhile, roc_auc which stands for *receiver operating charateristic* (ROC) and *area under curve* (AUC). It is derived through first plotting the true positive rate versus false positive rate curve, then you integrate that curve to retrieve a score. Finally, we choose a cross validation strategy to evaluate the model with different subsets of training data.

C Data analysis

C.1 Data Analysis

Data Preparation for Analysis

```
year1973 = df_actors.copy()
year1973 = df_actors.loc[df_actors['Year']>= 1972]

year1973["Total actors"] = (
    year1973["Number of female actors"] + year1973["Number of male
                                                  actors"]
)
print("Share of female actors > 1973: ", (year1973["Number of female
                                              actors"] / year1973["Total actors"]
                                              ).mean())
# display occurrences of a particular column
grossMaleSpeakMore = 0
moviesMaleSpeakMore = 0
for index, row in year1973.iterrows():
     if(row["Number words male"] > row["Number words female"]):
            grossMaleSpeakMore += row["Gross"]
            moviesMaleSpeakMore += 1

print(moviesMaleSpeakMore)
avgGrossMale = grossMaleSpeakMore/moviesMaleSpeakMore
print("Average Gross male: ", avgGrossMale)
# ---
grossFemaleSpeakMore = 0
moviesFemaleSpeakMore = 0
for index, row in year1973.iterrows():
     if(row["Number words male"] < row["Number words female"]):
            grossFemaleSpeakMore += row["Gross"]
            moviesFemaleSpeakMore += 1

print(moviesFemaleSpeakMore)
avgGrossFemale = grossFemaleSpeakMore/moviesFemaleSpeakMore
print("Average Gross female: ", avgGrossFemale)
print("Percentage More: ", (avgGrossMale-avgGrossFemale)/
                                              avgGrossFemale)
```

Question 1: Do men or women dominate speaking roles in Hollywood movies?

```
# Additional variables used for analysis

# Calculate gender word shares
df_actors["Female word share"] = (
    df_actors["Number words female"] / df_actors["Total words"]
)
df_actors["Male word share"] = 1 - df_actors["Female word share"]

# Calculate gender actor shares
df_actors["Total actors"] = (
    df_actors["Number of female actors"] + df_actors["Number of male
                                                  actors"]
)
df_actors["Female actor share"] = (
    df_actors["Number of female actors"] / df_actors["Total actors"]
)
df_actors["Male actor share"] = 1 - df_actors["Female actor share"]

# Create numeric class column
df_actors["Lead"] = df_actors["Lead"].astype("category")
df_actors["Lead numeric"] = df_actors["Lead"].cat.codes
```

```
# Share of male and female lead roles

# Calculate share across all movies
# Sort values and create list of movie indices
df_gender_share = df_actors.copy()
df_gender_share = df_gender_share.sort_values(by="Female word share",
                                   ascending=True)
movie_index = df_actors.index.values.tolist()

df_actor_share = df_actors.copy()
df_actor_share = df_actor_share.sort_values(by="Female actor share",
                                   ascending=True)
movie_index = df_actors.index.values.tolist()
fig, (ax2, ax3) = plt.subplots(1,2, figsize = (100,15)) # fig

# data 1
maleShare = df_actors["Lead numeric"].sum() / df_actors["Lead numeric"
                                   ].count()
femaleShare = 1 - maleShare
data1 = [maleShare, femaleShare]
#---------------------

ax2.bar(movie_index, df_actor_share["Female actor share"], width=0.5,
                                   linewidth=0)
ax2.bar(
    movie_index,
    df_actor_share["Male actor share"],
    bottom=df_actor_share["Female actor share"],
    width=0.5,
    linewidth=0,
)


# Plot average female share line
avg_female_actor_share = df_actor_share["Female actor share"].mean()

#-----------------------

# ax1
ax1.bar(["Female", "Male"],
    [femaleShare, maleShare],
    color=["tab:blue", "tab:orange"])
ax1.set_xlabel("Share")
ax1.set_ylabel("Share of female and male lead roles")
#ax1.set_title(
#    "Share of female and male lead roles",
#    fontsize=10,
#)
# ax2
ax2.plot(
    movie_index,
    [avg_female_actor_share for i in range(len(movie_index))],
    color="black",
    linewidth = 10
)


ax2.set_xlabel("Movie (indexed number)", fontsize=130)
ax2.set_ylabel("Share", fontsize=130)
ax2.set_title("Share of actors by gender", fontsize=130)
ax2.legend(["Female avg share", "Female share", "Male share"],
                                   fontsize = 90)
ax2.tick_params(axis='both', which='major', labelsize=130)
# ax3
# PLot the results

ax3.bar(movie_index, df_gender_share["Female word share"], width=0.5,
                                   linewidth=0)
ax3.bar(
    movie_index,
    df_gender_share["Male word share"],
    bottom=df_gender_share["Female word share"],
    width=0.5,
    linewidth=0,
)


# Plot average female share line
avg_female_share = df_gender_share["Female word share"].mean()
ax3.plot(
    movie_index, [avg_female_share for i in range(len(movie_index))],
                                       color="black", linewidth = 10)

# Format the chart
ax3.set_xlabel("Movie (indexed number)", fontsize=130)
ax3.set_ylabel("Share", fontsize=110)
ax3.set_title("Share of words by gender", fontsize=130)
ax3.legend(["Female avg share", "Female share", "Male share"],
                                   fontsize=90)
ax3.tick_params(axis='both', which='major', labelsize=130)
fig.tight_layout()


plt.savefig("question1")
```

Question 2: Has gender balance in speaking roles changed over time (i.e. years)?

```
# Filter out years until we have a continuous time series
df_share_by_year_filtered = df_share_by_year.loc[df_share_by_year["
                                   Year"] >= 1972]

fig, ([ax2, ax3, ax4]) = plt.subplots(1,3, figsize = (30, 6))
import matplotlib.gridspec as gridspec
#gs = gridspec.GridSpec(2, 2)
#ax2 = plt.subplot(gs[0, 0])
#ax3 = plt.subplot(gs[0, 1])
#ax4 = plt.subplot(gs[1, :])
# ax1
ax1.plot(
    df_share_by_year_filtered["Year"],
    df_share_by_year_filtered["Female word share"],
)
ax1.plot(
    df_share_by_year_filtered["Year"],
    df_share_by_year_filtered["Male word share"],
)


ax1.set_xlabel("Year", fontsize=40)
ax1.set_ylabel("Share", fontsize=40)
ax1.set_title("Share of words", fontsize=40)
ax1.legend(["Female", "Male"], fontsize=40)
ax1.tick_params(axis='both', which='major', labelsize=30)
#-----------------------
df_share_by_year_filtered = df_share_by_year.loc[df_share_by_year["
                                   Year"] >= 1972]
```

```
# ax2
ax2.plot(
    df_share_by_year_filtered["Year"],
    df_share_by_year_filtered["Female actor share"], linewidth = 5
)
ax2.plot(
    df_share_by_year_filtered["Year"],
    df_share_by_year_filtered["Male actor share"], linewidth = 5
)

#specify x-axis locations
x_ticks2 = [1972, 1980, 1990, 2000, 2010]
#specify x-axis labels
x_labels2 = ['1972', '1980', '1990', '2000', '2010']
ax2.set_xticks(ticks=x_ticks2, labels=x_labels2)
ax2.set_xlabel("Year", fontsize=40)
ax2.set_ylabel("Share", fontsize=40)
ax2.set_title("Average share of actors", fontsize=40)
ax2.legend(["Female", "Male"], fontsize=35)
ax2.tick_params(axis='both', which='major', labelsize=30)
# --------------------
df_share_by_year_filtered = df_share_by_year.loc[df_share_by_year["
                                   Year"] >= 1972]
# ax3
ax3.plot(
    df_share_by_year_filtered["Year"],
    df_share_by_year_filtered["Female lead share"], linewidth = 5
)
ax3.plot(
    df_share_by_year_filtered["Year"],
    df_share_by_year_filtered["Male lead share"], linewidth = 5
)


#specify x-axis locations
x_ticks3 = [1972, 1980, 1990, 2000, 2010]
#specify x-axis labels
x_labels3 = ['1972', '1980', '1990', '2000', '2010']
ax3.set_xticks(ticks=x_ticks3, labels=x_labels3)
ax3.set_xlabel("Year", fontsize=40)
ax3.set_ylabel("Share", fontsize=40)
ax3.set_title("Average share of lead actors", fontsize=40)
ax3.legend(["Female", "Male"], fontsize=35)
ax3.tick_params(axis='both', which='major', labelsize=30)
#-----------------

df_word_share_by_year = pd.DataFrame(
    df_share_by_year.groupby("Year")["Female word share"].mean().
                                       reset_index()
)
df_word_share_by_year["Decade"] = (
    df_word_share_by_year["Year"].astype(str).str[:3] + "0s"
)
df_word_share_by_year

df_word_share_by_decade = pd.DataFrame(
    df_word_share_by_year.groupby("Decade")["Female word share"].mean
                                       ().reset_index()
)
# Calculate average word share by year
df_share_by_year = pd.DataFrame(
    df_actors.groupby("Year")[
        [
            "Female word share",
            "Male word share",
            "Female actor share",
            "Male actor share",
        ]
    ]
    .mean()
    .reset_index()
)

df_lead_by_year = pd.DataFrame(
    df_actors.groupby("Year")["Lead numeric"]
    .mean()
    .reset_index()
    .rename(columns={"Lead numeric": "Male lead share"})
)

df_share_by_year = df_share_by_year.join(df_lead_by_year.set_index("
                                    Year"), on="Year")
df_share_by_year["Female lead share"] = 1 - df_share_by_year["Male
                                    lead share"]
plt.bar(['70s', '80s', '90s','00s', '10s'], df_word_share_by_decade["
                                    Female word share"][4:9])
#df_word_share_by_decade["Decade"]
# ax4

ax4.set_xlabel("Decades: 1970s-2010s", fontsize=40)
ax4.set_ylabel("Share", fontsize=40)
ax4.set_title("Average share of female words", fontsize=40)
ax4.tick_params(axis='both', which='major', labelsize=30)
plt.tight_layout()
plt.rcParams["figure.figsize"] = (30,12)
plt.savefig("question2")
```

Question 3: Do films in which men do more speaking make a lot more money than films in which women speak more?

```
# Calculate average word share by year
after1972 = df_actors.loc[df_actors['Year']>= 1972]

#occur = before1972.groupby(['Lead'])
df_share_by_year = pd.DataFrame(
        after1972.groupby("Year")[
            [
                "Female word share",
                "Male word share",
                "Female actor share",
                "Male actor share",
            ]
        ]
        .mean()
        .reset_index()
)

df_lead_by_year = pd.DataFrame(
        after1972.groupby("Year")["Lead numeric"]
        .mean()
        .reset_index()
        .rename(columns={"Lead numeric": "Male lead share"})
)

df_share_by_year = df_share_by_year.join(df_lead_by_year.set_index("
                                    Year"), on="Year")
df_share_by_year["Female lead share"] = 1 - df_share_by_year["Male
                                    lead share"]
```

```
df_avg_gross_male_year = pd.DataFrame(
    after1972.loc[df_actors["Male word share"] > df_actors["Female
                                       word share"]]
    .groupby("Year")["Gross"]
    .mean()
    .reset_index()
    .rename(columns={"Gross": "Male avg gross"})
)

df_avg_gross_female_year = pd.DataFrame(
    after1972.loc[df_actors["Male word share"] < df_actors["Female
                                       word share"]]
    .groupby("Year")["Gross"]
    .mean()
    .reset_index()
    .rename(columns={"Gross": "Female avg gross"})
)

df_avg_gross_gender_year = df_avg_gross_male_year.join(
    df_avg_gross_female_year.set_index("Year"), on="Year"
).fillna(0)

# Filter out years until we have a continuous time series
df_share_by_year_filtered = df_avg_gross_gender_year.loc[
    df_share_by_year["Year"] >= 1972
]

# plot data
fig, (ax1,ax2) = plt.subplots(1,2, figsize = (40,12))
ax1.plot(
    df_avg_gross_gender_year["Year"],
    df_avg_gross_gender_year["Female avg gross"], linewidth = 8
)
ax1.plot(
    df_avg_gross_gender_year["Year"],
    df_avg_gross_gender_year["Male avg gross"], linewidth = 8
)

# Format the chart
ax1.set_title(
    "Avg gross depending on majority word count",
    fontsize=60
)
ax1.set_xlabel("Year", fontsize=60)
ax1.set_ylabel("Gross", fontsize=60)
ax1.legend(["Female", "Male"], fontsize=60)
#specify x-axis locations
x_ticks = [1972, 1980, 1990, 2000, 2010]
#specify x-axis labels
x_labels = ['1972', '1980', '1990', '2000', '2010']
ax1.set_xticks(ticks=x_ticks, labels=x_labels)
ax1.tick_params(axis='both', which='major', labelsize=60)

#------------------

after1972 = df_actors.loc[df_actors['Year']>= 1972]

avg_gross_male = after1972[
    after1972["Male word share"] > after1972["Female word share"]
]["Gross"].mean()
avg_gross_female = after1972.loc[
    after1972["Male word share"] < after1972["Female word share"]
]["Gross"].mean()
```

```
print("Avg gross male speak more: ", avg_gross_male)
print("Avg gross female speak more: ", avg_gross_female)

# Lead
avg_gross_male_lead = after1972.loc[after1972['Lead']== 'Male']["Gross
                                    "].mean()
avg_gross_female_lead = after1972.loc[after1972['Lead']== 'Female']["
                                    Gross"].mean()
print("avg gross male lead: ", avg_gross_male_lead)
print("avg gross female lead: ", avg_gross_female_lead)

# ax2
ax2.bar(
    ["Female lead", "Male lead"],
    [avg_gross_female_lead, avg_gross_male_lead],
    color=["tab:blue", "tab:orange"],
)
ax2.set_ylabel("Gross", fontsize = 60)
ax2.set_title("Avg gross depending on lead gender", fontsize = 60)
ax2.set_xlabel("Avg gross", fontsize = 60)
ax2.tick_params(axis='both', which='major', labelsize=60)
#--------------------

fig.tight_layout(pad = 5)
plt.savefig("question3")
print(1-(avg_gross_female_lead/avg_gross_male_lead))

print((avg_gross_male_lead-avg_gross_female_lead)/
                                    avg_gross_female_lead)
```

C.2 Permutation Test and Analysis of relationship between Features in classes

Analysis of relationship between features and classes

Analysis conducted as input to the modeling. Goal is to understand distribution of different features and which ones can be significant for modeling. Class-dependent feature distribution: Histograms and boxplots

```
df_female = df_actors.loc[df_actors["Lead numeric"] == 0]
df_male = df_actors.loc[df_actors["Lead numeric"] == 1]
features = [
    col for col in df_actors.columns if col not in ("Lead", "Lead
                                        numeric", "Year")
]

fig, axes = plt.subplots(len(features), 2, figsize=(20, 100))
fig.tight_layout(pad=4)

row_index = 0
col_index = 0

for col in features:

    sns.histplot(
        data=df_female[col],
        bins=20,
        color="tab:blue",
        stat="probability",
        ax=axes[row_index, col_index],
    )
    axes[row_index, col_index].set_title(
        f"{col} (Female lead character)", fontsize=16, pad=16
    )
    y_lim_1 = axes[row_index, col_index].get_ylim()
    x_lim_1 = axes[row_index, col_index].get_xlim()

    col_index += 1

    sns.histplot(
        data=df_male[col],
        bins=20,
        color="tab:orange",
        stat="probability",
        ax=axes[row_index, col_index],
    )
    axes[row_index, col_index].set_title(
        f"{col} (Male lead character)", fontsize=16, pad=16
    )

    y_lim_2 = axes[row_index, col_index].get_ylim()
    x_lim_2 = axes[row_index, col_index].get_xlim()

    y_max = np.max(np.array(y_lim_1, y_lim_2))
    x_max = np.max(np.array(x_lim_1, x_lim_2))

    # Format plot 1
    axes[row_index, col_index - 1].xaxis.set_tick_params(labelsize=16)
    axes[row_index, col_index - 1].yaxis.set_tick_params(labelsize=16)
    axes[row_index, col_index - 1].set_ylabel("Count of movies",
                                       fontsize=16)
    axes[row_index, col_index - 1].set_xlim([0, x_max])
    axes[row_index, col_index - 1].set_ylim([0, y_max * 1.2])

    # Format plot 2
    axes[row_index, col_index].xaxis.set_tick_params(labelsize=16)
    axes[row_index, col_index].yaxis.set_tick_params(labelsize=16)
    axes[row_index, col_index].set_ylabel("Count of movies", fontsize=
                                       16)
    axes[row_index, col_index].set_xlim([0, x_max])
    axes[row_index, col_index].set_ylim([0, y_max * 1.2])

    col_index = 0
    row_index += 1

fig.suptitle(
    "Distribution of features depending on gender of lead actor",
                                       fontsize=20, y=1
)

print()
plt.show()
```

```
df_female = df_actors.loc[df_actors["Lead numeric"] == 0]
df_male = df_actors.loc[df_actors["Lead numeric"] == 1]
features = [col for col in df_actors.columns if col not in ("Lead", "
                                   Lead numeric")]

fig, axes = plt.subplots(len(features) // 2, 2, figsize=(20, 60))
fig.tight_layout(pad=4)

row_index = 0
col_index = 0

for col in features:

    sns.boxplot(
        data=df_actors, x="Lead", y=col, showfliers=False, ax=axes[
                                               row_index, col_index]
    )
    axes[row_index, col_index].set_title(f"{col}", fontsize=16, pad=16
                                        )

    col_index += 1

    if col_index == 2:
        col_index = 0
        row_index += 1

fig.suptitle(
    "Boxplots of features by gender of lead actor (outliers excl. for
                                        readability)",
    fontsize=20,
    y=1,
)

print()
plt.show()
```

```
is_female = df_actors["Lead"]=="Female"
is_male = df_actors["Lead"]=="Male"
#split the data into female lead and male lead
df_movies_female = df_actors[is_female]
df_movies_male = df_actors[is_male]

#------

femaleLead = data.loc[data['Lead'] == 'Female']
maleLead = data.loc[data['Lead'] == 'Male']
# data 1
ageFemaleLead = femaleLead['Age Lead']
ageMaleLead = maleLead['Age Lead']
boxData = [ageFemaleLead, ageMaleLead]
# data2
numberOfFemaleFemaleLead = femaleLead['Number of female actors']
numberOfMaleMaleLead = maleLead['Number of female actors']
boxData2 = [numberOfFemaleFemaleLead, numberOfMaleMaleLead]
# data 3
differenceFemaleLead = femaleLead['Difference in words lead and co-
                                        lead']
differenceMaleLead = maleLead['Difference in words lead and co-lead']
boxData3 = [differenceFemaleLead, differenceMaleLead]
# data 4
numberWordsMaleFemaleLead = femaleLead['Number words male']
numberWordsMaleMaleLead = maleLead['Number words male']
boxData4 = [numberWordsMaleFemaleLead, numberWordsMaleMaleLead]
# data5
meanAgeFemaleLead = femaleLead['Mean Age Female']
meanAgeFemaleMaleLead = maleLead['Mean Age Female']
boxData5 = [meanAgeFemaleLead, meanAgeFemaleMaleLead]
# data 6
totalWordsFemaleLead = femaleLead['Total words']
totalWordsMaleLead = maleLead['Total words']
boxData6 = [totalWordsFemaleLead, totalWordsMaleLead]

#fig, ([ax1, ax2, ax7], [ax4, ax5, ax8]) = plt.subplots(2,3, figsize =
                                        (23,15)) # fig
fig, ([ax1, ax2, ax7]) = plt.subplots(1,3, figsize = (25,6)) # fig

# ax1
bp = ax1.boxplot(boxData, showfliers=False, patch_artist = True)
```

```
ax1.set_ylabel("Age", fontsize = 35)
ax1.set_xticks([1,2],['Female', 'Male'], fontsize = 7)
ax1.set_title("Lead age", fontsize = 35)
ax1.set_xlabel("Lead", fontsize = 35)
ax1.tick_params(axis='both', which='major', labelsize=35)
colors = ['blue', 'orange']
for patch, color in zip(bp['boxes'], colors):
    patch.set_facecolor(color)
for median in bp['medians']:
    median.set(color ='red',
               linewidth = 3)
# ax2
bp2 = ax2.boxplot(boxData2, showfliers=False, patch_artist = True)
ax2.set_xticks([1,2],['Female', 'Male'], fontsize = 7)
ax2.set_ylabel("Actors", fontsize = 35)
ax2.set_title("No. Female Actors", fontsize = 35)
ax2.set_xlabel("Lead", fontsize = 35)
ax2.tick_params(axis='both', which='major', labelsize=35)
for patch, color in zip(bp2['boxes'], colors):
    patch.set_facecolor(color)
for median in bp2['medians']:
    median.set(color ='red',
               linewidth = 3)

# DIFFERENCE WORDS = COLUM 3
column = 3

#print which column we are using
print(df_movies_male.columns.values[column])
#calculate the true difference of mean between the two distributions
true_diff = df_movies_male.iloc[:,column].mean()-df_movies_female.iloc
                                    [:,column].mean()

    #now we randomly combine the different distributions into two new
                                        groups and sample from then to
                                        calculate their difference
    #multiple times
a_n, b_n = len(df_movies_male.iloc[:,column]), len(df_movies_female.
                                    iloc[:,column])
    #how many times we are sampling
r = 10000
combined = np.concatenate([df_movies_male.iloc[:,column],
                                    df_movies_female.iloc[:,column]])
diff_list = list()

for i in range(r):
    np.random.shuffle(combined)
    a_sample = combined[:a_n]
    b_sample = combined[a_n:]

    diff = a_sample.mean() - b_sample.mean()
    diff_list.append(diff)
    #compute mean of the different list and the standard deviation
mean = np.mean(diff_list)
std_dev = np.std(diff_list)

    #create a histrogram to visualize the output
n, bins, patches = ax7.hist(diff_list, 50, density=True, facecolor='g'
                                    , alpha=0.75)
ax7.axvline(true_diff, color='r', linewidth=4)
ax7.set_ylabel("Frequency", fontsize = 35)
ax7.set_title("Permutation Test:\n Difference in words", fontsize = 35
                                    )
ax7.set_xlabel("Dif. mean both\n distributions", fontsize = 35)
ax7.tick_params(axis='both', which='major', labelsize=35)
```

```
ax7.text(750, 0.002,'$H_{0}$', fontsize = 35, color = "red")
#---------------------------

#------------------------
# MEAN AGE MALE = COLUMN 9
column = 9

#print which column we are using
print(df_movies_male.columns.values[column])
#calculate the true difference of mean between the two distributions
true_diff = df_movies_male.iloc[:,column].mean()-df_movies_female.iloc
                                    [:,column].mean()

    #now we randomly combine the different distributions into two new
                                        groups and sample from then to
                                        calculate their difference
    #multiple times
a_n, b_n = len(df_movies_male.iloc[:,column]), len(df_movies_female.
                                    iloc[:,column])
    #how many times we are sampling
r = 10000
combined = np.concatenate([df_movies_male.iloc[:,column],
                                    df_movies_female.iloc[:,column]])
diff_list = list()

for i in range(r):
    np.random.shuffle(combined)
    a_sample = combined[:a_n]
    b_sample = combined[a_n:]

    diff = a_sample.mean() - b_sample.mean()
    diff_list.append(diff)
    #compute mean of the different list and the standard deviation
mean = np.mean(diff_list)
std_dev = np.std(diff_list)

    #create a histrogram to visualize the output
#n, bins, patches = ax8.hist(diff_list, 50, density=True, facecolor='g
                                    ', alpha=0.75)
#ax8.axvline(true_diff, color='r')
#ax8.set_ylabel("Count", fontsize = 40)
#ax8.set_title("Permutation Test:\n Mean Age Male", fontsize = 40)
#ax8.set_xlabel("Dif. mean both\n distributions", fontsize = 40)
#ax8.tick_params(axis='both', which='major', labelsize=25)
#---------------------------

fig.tight_layout()
plt.savefig("boxPlots") # save figure
```

Why? We don't know if the data is normally distributed (t-test), so this way, we do not make any assumption about the data.

We are using a Monte Carlo method to do permutation testing

```
 is_female = df_actors["Lead"] == "Female"
is_male = df_actors["Lead"] == "Male"

# split the data into female lead and male lead
df_movies_female = df_actors[is_female]
df_movies_male = df_actors[is_male]

def permutation_test(column):

    # print which column we are using
    print(df_movies_male.columns.values[column])
    # calculate the true difference of mean between the two
                                    distributions
    true_diff = (
        df_movies_male.iloc[:, column].mean() - df_movies_female.iloc[
                                            :, column].mean()
    )

    # now we randomly combine the different distributions into two new
                                     groups and sample from then to
                                     calculate their difference
    # multiple times
    a_n, b_n = len(df_movies_male.iloc[:, column]), len(
        df_movies_female.iloc[:, column]
    )
    # how many times we are sampling
    r = 10000
    combined = np.concatenate(
        [df_movies_male.iloc[:, column], df_movies_female.iloc[:,
                                            column]]
    )
    diff_list = list()

    for i in range(r):
        np.random.shuffle(combined)
        a_sample = combined[:a_n]
        b_sample = combined[a_n:]

        diff = a_sample.mean() - b_sample.mean()
        diff_list.append(diff)
    # compute mean of the different list and the standard deviation
    mean = np.mean(diff_list)
    std_dev = np.std(diff_list)

    # create a histogram to visualize the output
    n, bins, patches = plt.hist(diff_list, 50, density=True, facecolor
                                    ="g", alpha=0.75)
    plt.axvline(true_diff, color="r")
    plt.show()
    print("True diff = ", true_diff)
    # 95% confidence interval
    print("Condidence Interval", [mean - 2 * std_dev, mean + 2 *
                                    std_dev])
    if true_diff > mean -2*std_dev and true_diff < mean + 2 * std_dev:
        print("The value lies WITHIN our Confidence interval.
                                            Therefore we cannot say
                                            anything with certainty,
                                            but the underlying
                                            distributions are not
                                            clearly different.")
    else:
        print("The value lies OUTSIDE our Confidence Interval and we
                                            therefore reject the null
                                            hypothesis, meaning that
                                            the samples most likely
                                            come from different
                                            distributions.")



# calling permutation test for all features

for i in range(0, (len(df_movies_male.columns.values) - 1)):
    permutation_test(i)
```

C.3 Test for Gaussian distribution

Testing if class-dependent features are normally distributed. This is a key assumption for discriminant analysis. p-value < 0.05 means we reject the null hypothesis that the data is normally distributed, with 95% confidence.

```
from pingouin import multivariate_normality
from sklearn.preprocessing import StandardScaler

df_actors = pd.read_csv("train.csv")

df_actors["Lead"] = df_actors["Lead"].astype("category")
df_actors["Lead numeric"] = df_actors["Lead"].cat.codes

df_female = df_actors.loc[df_actors["Lead numeric"] == 0]
df_male = df_actors.loc[df_actors["Lead numeric"] == 1]

df_female = df_female.drop(["Lead", "Lead numeric"], axis="columns")
df_male = df_male.drop(["Lead", "Lead numeric"], axis="columns")

df_female_scale = StandardScaler().fit_transform(df_female)
df_male_scale = StandardScaler().fit_transform(df_male)

male_norm_test = multivariate_normality(df_male, alpha=0.05)
female_norm_test = multivariate_normality(df_female, alpha=0.05)

male_norm_test, female_norm_test

male_norm_test = multivariate_normality(df_male_scale, alpha=0.05)
female_norm_test = multivariate_normality(df_female_scale, alpha=0.05)

male_norm_test, female_norm_test
```

D Joint feature engineering and cross validation

D.1 Feature engineering

```python
from __future__ import annotations
import pandas as pd
import numpy as np
from sklearn.preprocessing import (
    MaxAbsScaler,
    MinMaxScaler,
    RobustScaler,
    StandardScaler,
)


class FeatureEngineering:
    """
    Collection of methods to engineer additional features and
    add to the original dataframe. Note that some methods can
    require previous steps to have run first.
    """

    def __init__(self):
        pass

    @staticmethod
    def numeric_class_column(df: pd.DataFrame) -> pd.DataFrame:
        """
        Add a column with numeric class values.

        Args:
            df: Dataframe with data about actors and movies

        Returns:
            df: Dataframe with column "Lead numeric" appended
        """

        df["Lead"] = df["Lead"].astype("category")
        df["Lead numeric"] = df["Lead"].cat.codes

        return df

    @staticmethod
    def encode_class_column(df: pd.DataFrame):
        """
        Encode column lead to numerical values.

        Args:
            df: Dataframe with data about actors and movies.

        Returns:
            Null: Encodes "Lead" column with numerical values.
        """
        # Encode Lead to numeric values
        df["Lead"] = df["Lead"].astype("category").cat.codes

    @staticmethod
    def calculate_total_actors(df: pd.DataFrame) -> pd.DataFrame:
        """
        Calculate total number of actors.
        """

        # Total number of actors of both genders
        df["Total actors"] = df["Number of male actors"] + df["Number
                                            of female actors"]

        return df

    def calculate_relative_shares(self, df: pd.DataFrame) -> pd.
                                        DataFrame:
        """
        Calculate relative shares for relevant columns and append to
        the original dataframe. Calculation of word shares is
        dependent on first calling calculate_totals().

        Args:
            df: Dataframe with data about actors and movies

        Returns:
            df: Dataframe with the following columns appended:
            "Female word share",
            "Male word share", "Female actor share", "Male actor share
                                                  "
        """

        # Calculate totals first if needed
        if "Total actors" not in df.columns:
            df = self.calculate_total_actors(df)

        # Calculate gender word shares
        # This excludes the lead character to avoid leakage in
                                            training
        df["Female word share"] = (
                df["Number words female"] / df["Total words"]
        )
        df["Male word share"] = 1 - df["Female word share"]

        df["Lead word share"] = (
                df["Number of words lead"] / df["Total words"]
        )

        # Calculate gender actor shares
        df["Total actors"] = (
                df["Number of female actors"] + df["Number of male
                                                    actors"]
        )
        df["Female actor share"] = (
                df["Number of female actors"] / df["Total actors"]
        )
        df["Male actor share"] = 1 - df["Female actor share"]

        return df

    @staticmethod
    def calculate_ratios(df: pd.DataFrame) -> pd.DataFrame:

        df["Female / male word ratio"] = (
                df["Number words female"] / df["Number words male"]
        )
        df["Female / male actor ratio"] = (
                df["Number of female actors"] / df["Number of male
                                                    actors"]
        )

        return df

    @staticmethod
    def log_features(df: pd.DataFrame) -> pd.DataFrame:
        log_columns = [
            "Total words",
            "Number words female",
            "Number words male",
            "Number of words lead",
            "Difference in words lead and co-lead",
            "Gross",
        ]

        for column in log_columns:
            df[column] = np.log(df[column]).replace([np.inf, -np.inf],
                                                    0)

        return df

    @staticmethod
    def decade(df: pd.DataFrame) -> pd.DataFrame:

        df["Decade"] = df["Year"].astype(str).str[:3] + "0s"
        df["Decade"] = df["Decade"].astype("category").cat.codes


        return df

    def gross_decade_ratio(self, df: pd.DataFrame) -> pd.DataFrame:

        if "Decade" not in df.columns:
            df = self.decade(df)

        df_avg_gross_decade = pd.DataFrame(
            df.groupby("Decade")["Gross"]
                .mean()
                .reset_index()
                .rename(columns={"Gross": "Avg gross decade"})
        )

        df = df.join(df_avg_gross_decade.set_index("Decade"), on="
                                            Decade")
        df["Gross ratio vs decade"] = df["Gross"] / df["Avg gross
                                            decade"]

        return df


    @staticmethod
    def calculate_difference(
            df: pd.DataFrame, column_b: str, column_a: str
            ) -> pd.Series:
        """
        Calculates difference between columnB and columnA,
        i.e. columnB-columnA.
        Args:
            df: Dataframe with data about actors and movies
            column_b: String name of column that will become subject
                                                of
            subtraction.
            column_a: String name of column which will be term to
                                                subtract
            column_b by.
        Returns:
            Series: Series with one column contatining the difference
            of the two input columns.
        """
        # Check if column B exists in df
        assert column_b in df.columns, "column_b does not exist in df!
                                   "

        # Check if column A exists in df
        assert column_a in df.columns, "columnA does not exist in df!"

        # Check if columnB contains numerical values
        assert (
            df[column_b].dtypes == np.int64 or df[column_b].dtypes ==
                                               np.float64
        ), "columnB does not contain a numerical value!"

        # Check if columnA contains numerical values
        assert (
            df[column_a].dtypes == np.int64 or df[column_a].dtypes ==
                                               np.float64
        ), "column_a does not contain a numerical value!"

        # Calculates the difference of column B and A
        result = df[column_b] - df[column_a]

        return result

    @staticmethod
    def calculate_mean_yearly(
        df: pd.DataFrame, columns: list | str
    ) -> pd.Series | pd.DataFrame:
        """
        Calculates yearly mean of the desired feature.
        Args:
            df: Dataframe with data about actors and movies
            columns: String name of column or list of string name of
                                               columns
            of which the mean will be calculated.
        Returns:
            If type(columns) == list:
                DataFrame: A DataFrame with several columns containing
                those feature's yearly mean
            else:
                Series: Series with one column containing desired
                                                   feature's
                yearly mean.
        """
        # Calculate the yearly value of column
        if isinstance(columns, list):
            # Check if the columns exist within the DataFrame
            assert all(
                column in df.columns.tolist() for column in columns
            ), "One or several columns does not exist in df!"

            # Check if columns contains numerical value
            assert all(
                [dtype in (np.int64, np.float64) for dtype in df[
                                                   columns].dtypes.
                                                   tolist()]
            ), "One or several columns does not contatin numerical
                                               values!"

            df_columns_by_year = pd.DataFrame(df.groupby("Year")[
                                               columns].mean())
            return df_columns_by_year

        else:
            # Check if column exists in df
            assert columns in df.columns, "column does not exist in df
                                               !"

            # Check if column contains numerical values
            assert (
                df[columns].dtypes == np.int64 or df[columns].dtypes =
                                                   = np.float64
            ), "column does not contain a numerical value!"
            # Calculate the yearly value of column
            total_yearly = df.groupby("Year")[columns].sum()

            # Count total movies by year
            movies_yearly = df.Year.value_counts()

            # Calculate yearly mean of column for each row in df
            mean_list = []
            for index, value in df["Year"].iteritems():
                mean_list.append(total_yearly[value] / movies_yearly[
                                                   value])

            return pd.Series(mean_list)

    @staticmethod
    def scaling(df: pd.DataFrame, method: str, columns: list | str =
                                      None):
        """
        A function which scales an entire dataframe or specified
                                           features in the dataframe.

        Methods:

        MaxAbs Scaling: This method scales and translates each feature
        individually such that the maximal absolute value of each
                                           feature
        in the training set will be 1.0. It does not shift/center the
                                           data,
        and thus does not destroy any sparsity.

        MinMax Scaling: Rescaling of all values in a feature in the
                                           range 0 to 1.
        The min value in the original range will take the value 0,
        the max value will take 1 and the rest of the values in
                                           between the two will be
                                           appropriately scaled.

        Robust Scaling: Scale features using statistics that are
                                           robust to outliers.
        This Scaler removes the median and scales the data according
                                           to the quantile range (
                                           Interquartile range).
        The IQR is the range between the 1st quartile (25th quantile)
                                           and the 3rd quartile (75th
                                           quantile).

        Standard Scaling: Standardize features by removing the mean
                                           and scaling to unit
                                           variance.

        Normalize scaling: [To add description]
        Args:
            df: Dataframe with data about actors and movies
            Method: String name of scaling method, either MaxAbs,
                                               MinMax, Robust or
                                               Standard.
            columns: list or str of features to be scaled, default is
                                               None,
                     if no argument is passed the entire dataframe
                                                          will be scaled
                                                          .
        Returns:
            void: Changes the desired dataframe internally, returns
                                               nothing.

        """
        assert method.lower() in [
            "maxabs",
            "minmax",
            "robust",
            "standard",
            "normalize",
        ], "Method should contain a string with name of scaling method
                                          , either MaxAbs, MinMax,
                                          Robust, Standard or
                                          Normalize"

        # Check if using assigning specific columns
        if columns != None:
            # Check if columns is a list
            if isinstance(columns, list):
                # Check if the columns exist within the DataFrame
                assert all(
                    column in df.columns.tolist() for column in
                                                       columns
                ), "One or several columns does not exist in df!"

                # Check if columns contains numerical value
                assert all(
                    [
                        dtype in (np.int64, np.float64)
                        for dtype in df[columns].dtypes.tolist()
                    ]
                ), "One or several columns does not contatin numerical
                                                   values!"

                # Transform features to fit scaler
                features = [np.array(df[feature]).reshape(-1, 1) for
                                                     feature in columns]

            else:
                # Check if column exists
                assert columns in df.columns, "column does not exist
                                                     in df!"

                # Check if numerical value
                assert (
                    df[columns].dtypes == np.int64 or df[columns].
                                                       dtypes == np.
                                                       float64
                ), "column does not contain a numerical value!"

                # Transform features
                features = [np.array(df[column]).reshape(-1, 1)]

                # Create a list containing column
                columns = list(columns)

        else:
            if "Lead" in df.columns.tolist():
                # Remove lead from features and reshape all features
                features = [
                    np.array(df[feature]).reshape(-1, 1)
                    for feature in df.columns.drop("Lead")
                ]
                columns = df.columns.drop("Lead")

            else:
                # Reshape all features to fit scaler
                features = [
                    np.array(df[feature]).reshape(-1, 1) for feature
                                                     in df.columns
                ]

                # Create a local variable with columns
                columns = df.columns

        for feature, column_name in zip(features, columns):
            # If using MaxAbsScaling method
            if method.lower() == "maxabs":
                df[column_name] = MaxAbsScaler().fit_transform(feature
                                                    )

            # If using MinMaxScaling method
            elif method.lower() == "minmax":
                df[column_name] = MinMaxScaler().fit_transform(feature
                                                    )

            # If using RobustScaling method
            elif method.lower() == "robust":
                df[column_name] = RobustScaler().fit_transform(feature
                                                    )

            # If using StandarScaling method
            elif method.lower() == "standard":
                try:
                    df[column_name] = StandardScaler().fit_transform(
                                                     feature)
                except ValueError:
                    print("Column name:", column_name, "\nFeature:",
                                                     feature)

            # If using normalizer method
            elif method.lower() == "normalize":
                df[column_name] = Normalizer().fit_transform(feature)

    def run_feature_engineering(
        self,
        df: pd.DataFrame,
        scaling_method: str = "standard",
        scaling: boolean = True,
        add_numeric_class_column: boolean = False,
        encode_class: boolean = True,
        total_actors: boolean = True,
        relative_shares: boolean = True,
        ratios: boolean = True,
        decade: boolean = True,
        differences: boolean = True,
        abs_diff: boolean = False,
        yearly_mean: boolean = True,
        yearly_mean_diff: boolean = True,
    ) -> pd.DataFrame:
        """Run desired methods in this class in sequence
        Args:
            df: A dataframe with data about actors and movies.
        scaling_method: A string with scaling method one out of "
                                    standard",
        "minmax", "maxabs", "robust" or "normalize"
        scaling: boolean, if True scales data with desired
                                    scaling_method
        add_numeric_class_column: boolean, if True adds a new
                                    numeric 'Lead' column
        encode_class: boolean, if True encodes existing 'Lead'
                                    column to numeric
        total_actors: boolean, if True adds column with total
                                    actors
        relative_shares: boolean, if True, adds columns with
                                    relative shares
        ratios: boolean, if True, adds columns with ratios
        decade: boolean, if True adds categorical decade and
                                    numeric decade column
        differences: boolean, if True adds differences of some
                                    columns
        abs_diff: boolean, if True adds columns of the absolute
                                    value from difference
        taken of 'Age Lead' & 'Age Co-Lead' and 'Mean Age Male' &
                                    'Mean Age Female'.
        yearly_mean: boolean, if True adds columns with the yearly
                                     mean of the original
        features.
        yearly_mean_diff: boolean, if True takes the difference of
                                     yearly mean and the
        original features.
    Returns:
            df: A dataframe with data about actors and movies.
    """

    # Create numeric class column
    if add_numeric_class_column:
        self.numeric_class_column(df)

    # Encode 'Lead' column to numeric values
    if encode_class:
        self.encode_class_column(df)

    # Calculate new aggregated totals columns
    if total_actors:
        self.calculate_total_actors(df)

    # Calculate relative shares between some columns
    if relative_shares:
        self.calculate_relative_shares(df)

    # Add ratios between features
    if ratios:
        self.calculate_ratios(df)

    # Create decade feature
    if decade:
        self.decade(df)

    # Calculate differences between some columns
    if differences and abs_diff:
        # Calculate difference in words by gender
        df["Difference Words Gender"] = self.calculate_difference(
            df, "Number words male", "Number words female"
        )

        # Calculdate difference of actors by gender
        df["Difference Actors"] = self.calculate_difference(
            df, "Number of male actors", "Number of female actors"
        )

        df["Difference Age Lead"] = self.calculate_difference(
            df, "Age Lead", "Age Co-Lead"
        )
        df["Difference Age Lead Abs"] = abs(
            self.calculate_difference(df, "Age Lead", "Age Co-Lead
                                                ")
        )
        df["Difference Mean Age"] = self.calculate_difference(
            df, "Mean Age Male", "Mean Age Female"
        )
        df["Difference Mean Age Abs"] = abs(
            self.calculate_difference(df, "Mean Age Male", "Mean
                                                Age Female")
        )

    elif differences == True and abs_diff == False:
        # Calculate difference in words by gender
        df["Difference Words Gender"] = self.calculate_difference(
            df, "Number words male", "Number words female"
        )

        # Calculdate difference of actors by gender
        df["Difference Actors"] = self.calculate_difference(
            df, "Number of male actors", "Number of female actors"
        )

        # Calculate age lead difference
        df["Difference Age Lead"] = self.calculate_difference(
            df, "Age Lead", "Age Co-Lead"
        )

        # Calculate mean age difference
        df["Difference Mean Age"] = self.calculate_difference(
            df, "Mean Age Male", "Mean Age Female"
        )

    if yearly_mean and yearly_mean_diff:
        original_features = [
            "Number words female",
            "Total words",
            "Number of words lead",
            "Difference in words lead and co-lead",
            "Number of male actors",
            "Number of female actors",
            "Number words male",
            "Gross",
            "Mean Age Male",
            "Mean Age Female",
            "Age Lead",
            "Age Co-Lead",
        ]
        mean_yearly = [
            "Yearly mean Number words female",
            "Yearly mean Total words",
            "Yearly mean Number of words lead",
            "Yearly mean Difference in words lead and co-lead",
            "Yearly mean Number of male actors",
            "Yearly mean Number of female actors",
            "Yearly mean Number words male",
            "Yearly mean Gross",
            "Yearly mean Mean Age Male",
            "Yearly mean Mean Age Female",
            "Yearly mean Age Lead",
            "Yearly mean Age Co-Lead",
        ]
        mean_diff_yearly = [
            "Yearly mean diff Number words female",
            "Yearly mean diff Total words",
            "Yearly mean diff Number of words lead",
            "Yearly mean diff Difference in words lead and co-lead
                                                ",
            "Yearly mean diff Number of male actors",
            "Yearly mean diff Number of female actors",
            "Yearly mean diff Number words male",
            "Yearly mean diff Gross",
            "Yearly mean diff Mean Age Male",
            "Yearly mean diff Mean Age Female",
            "Yearly mean diff Age Lead",
            "Yearly mean diff Age Co-Lead",
        ]
        # Calculate the yearly mean for all features and adds them
                                             to dataframe
        for orig, name in zip(original_features, mean_yearly):
            df[name] = self.calculate_mean_yearly(df, orig)

        # Take the difference of the yearly mean and the original
                                            feature
        for orig, yearl, name in zip(
            original_features, mean_yearly, mean_diff_yearly
        ):
            df[name] = self.calculate_difference(df, yearl, orig)

    elif yearly_mean == True and yearly_mean_diff == False:
        original_features = [
            "Number words female",
            "Total words",
            "Number of words lead",
            "Difference in words lead and co-lead",
            "Number of male actors",
            "Number of female actors",
            "Number words male",
            "Gross",
            "Mean Age Male",
            "Mean Age Female",
            "Age Lead",
            "Age Co-Lead",
        ]
        mean_yearly = [
            "Yearly mean Number words female",
            "Yearly mean Total words",
            "Yearly mean Number of words lead",
            "Yearly mean Difference in words lead and co-lead",
            "Yearly mean Number of male actors",
            "Yearly mean Number of female actors",
            "Yearly mean Number words male",
            "Yearly mean Gross",
            "Yearly mean Mean Age Male",
            "Yearly mean Mean Age Female",
            "Yearly mean Age Lead",
            "Yearly mean Age Co-Lead",
        ]

        # Calculate the yearly mean for all features and adds them
                                             to dataframe
        for orig, name in zip(original_features, mean_yearly):
            df[name] = self.calculate_mean_yearly(df, orig)
    # Perform scaling
    if scaling:
            self.scaling(df, scaling_method)

    return df
```

D.2 Cross-validation

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import (
    StandardScaler,
    MaxAbsScaler,
    MinMaxScaler,
    RobustScaler,
    Normalizer,
)
from sklearn.model_selection import cross_validate
from sklearn.model_selection import StratifiedKFold


class ModelEvaluation:
    def __init__(self):
        pass

    def cross_val(
        self,
        estimator: object,
        features: pd.DataFrame,
        target: pd.Series,
        nb_folds: int = 5,
        use_preprocess=False,
        preprocess_option: str = "standard",
    ):
        """
        Perform cross-validation and print model performance report.

        Args:
            estimator: the estimator used for fitting and testing
            features: set of features to train the estimator on
            target: labels in the classification
            nb_folds: number of folds (test sets) to use in the cross-
                                   validation
            use_preprocess: set to true if data is not already scaled
                                   / normalized
            preprocess_option: what scaling / standardization method
                                   to use
        """

        if type(features) == pd.DataFrame:
            feature_columns = list(features.columns)

        elif type(features) == pd.Series:
            feature_columns = features.name
            features = np.array(features).reshape(-1, 1)

        if target.dtype == str:
            target = target.copy()
            target = target.astype("category").cat.codes

        # Perform scaling of features
        if use_preprocess:
            features = features.copy()
            features = self.pre_process(features, preprocess_option)

        # Create k stratified folds
        k_folds = StratifiedKFold(n_splits=nb_folds)

        # Scoring metrics to be used
        scoring = {
            "overall_acc": [],
            "female_acc": [],
            "male_acc": [],
            "train_acc": [],
        }

        test_sets = pd.DataFrame()
        for train_indices, test_indices in k_folds.split(features,
                                                           target):
            x_train, y_train = features[train_indices], target[
                                                           train_indices]
            x_test, y_test = features[test_indices], target[
                                                           test_indices]
            test_sets = pd.concat([test_sets, pd.DataFrame(x_test)],
                                                           axis="rows")

            # Fit model and do prediction
            model = estimator.fit(x_train, y_train)
            y_pred = model.predict(x_test)
            y_pred_train = model.predict(x_train)

            # Put results in a dataframe
            df_res = pd.DataFrame({"y_pred": y_pred, "y_test": y_test}
                                                           )

            # Calculate and append metrics for this fold; 1 is male
                                                           and 0 is female
            overall_acc = np.mean(y_pred == y_test)
            train_acc = np.mean(y_pred_train == y_train)

            female_acc = (
                df_res.loc[
                    (df_res["y_test"] == 0) & (df_res["y_pred"] ==
                                                                   df_res["y_test"
                                                                   ])
                ].count()
                / df_res.loc[(df_res["y_test"] == 0)].count()
            )

            male_acc = (
                df_res.loc[
                    (df_res["y_test"] == 1) & (df_res["y_pred"] ==
                                                                   df_res["y_test"
                                                                   ])
                ].count()
                / df_res.loc[(df_res["y_test"] == 1)].count()
            )

            scoring["overall_acc"].append(overall_acc)
            scoring["female_acc"].append(female_acc)
            scoring["male_acc"].append(male_acc)
            scoring["train_acc"].append(train_acc)

        # Calculate key metrics for output
        overall_acc = scoring["overall_acc"]
        female_acc = scoring["female_acc"]
        male_acc = scoring["male_acc"]
        train_acc = scoring["train_acc"]
        # Summary key metrics on test folds
        avg_acc, min_acc, max_acc = (
            np.mean(overall_acc),
            np.min(overall_acc),
            np.max(overall_acc),
        )

        avg_female_acc, min_female_acc, max_female_acc = (
            np.mean(female_acc),
            np.min(female_acc),
            np.max(female_acc),
        )

        avg_male_acc, min_male_acc, max_male_acc = (
            np.mean(male_acc),
            np.min(male_acc),
            np.max(male_acc),
        )

        avg_train_acc, min_train_acc, max_train_acc = (
            np.mean(train_acc),
            np.min(train_acc),
            np.max(train_acc),
        )

        print("----------- Cross-validation report -----------\n")
        print(f"Model: {estimator}\n")
        print(f"Feature set: {feature_columns}\n")
        print(f"Number of folds: {nb_folds}\n")
        print("Performance:")
        print(
            f"- Accuracy: {avg_acc:.3f} (avg), {min_acc:.3f} (min), {
                                   max_acc:.3f} (max)"
        )
        print(
            f"- Accuracy, female: {avg_female_acc:.3f} (avg), {
                                   min_female_acc:.3f} (
                                   min), {max_female_acc:.
                                   3f} (max)"
        )
        print(
            f"- Accuracy, male: {avg_male_acc:.3f} (avg), {
                                   min_male_acc:.3f} (min)
                                   , {max_male_acc:.3f} (
                                   max)"
        )
        print(
            f"- Training accuracy: {avg_train_acc:.3f} (avg), {
                                   min_train_acc:.3f} (min
                                   ), {max_train_acc:.3f}
                                   (max)"
        )
        print("---------------------------------------------\n")

    @staticmethod
    def pre_process(features, method):

        assert method in ["max_abs", "min_max", "robust", "standard",
                                  "normalize"], (
            "Method should contain a string with name of scaling
                                   method, either "
            "max_abs, min_max, robust, standard or normalize"
        )
        if method == "max_abs":
            features = MaxAbsScaler().fit_transform(features)

        elif method == "min_max":
            features = MinMaxScaler().fit_transform(features)

        elif method == "robust":
            features = RobustScaler().fit_transform(features)

        elif method == "standard":
            features = StandardScaler().fit_transform(features)

        else:
            features = Normalizer().fit_transform(features)

        return features
```

E Naive base classifier

```
class NaiveBaseClassifier:
    def init(self):
        pass

    def fit(self, x_train, y_test):
        pass

    def predict(self, x_test):
        return pd.Series([1] * len(x_test))
```

F Logistic regression

An example of a top performing model had the parameters: $C \approx 0.615848211066026$, $penalty = l2(ridgeregression)$, $solver = newton - cg$. The newton-cg solver is a newton method which uses the Hessian matrix which calculates the second derivative in order to minimise the loss.

F.1 Grid search with logistic regression

This alternative of finding the best parameters is most often not preferred for more complex models than logistic regression due to the time complexity.

Three parameters were optimized which were the C-value, the penalty, and the solver. The C value's range were "np.logspace (-4 ,4 , 20)" and is deciding the strength of the regularization. This is because C is the inverse of the regularization strength, which means a large C valuer results in a weaker regularization.

Penalties available for logistic regression were lasso, ridge and elasticnet which is the newest that combines lasso and ridge. A choice of not using any penalty at all is also possible. Some penalties doesn't work with all solvers, for example is Stochastic Average Gradient or SAG, only supported by the ridge regularization or no penalty at all. Main difference between ridge and lasso regularization is that ridge used squared errors meanwhile lasso uses the absolute error. If lambda in ridge regression is set to 0, it will produce the ordinary least squares.

The solvers that were inspected were "lbfgs", "liblinear", "newtong-cg", "sag" and "saga". The solvers is just different methods of minimising the loss, where most of them uses different version of gradient descent. Gradient is used to find a local minimum of the cost function, where the function must be convex and differentiable. If a function is non convex, finding the local minimum can't be ensured. One way of knowing that the function is convex is by inspecting whether the second derivative is positive or not, with a positive second derivative, the function is convex. While performing gradient descent, saddle points could be encountered as well, which could be identified by using the Hessian matrix. Gradient descent works by computing the gradient at the current position, scales it with the chosen learning rate, and then goes the opposite direction in order to decrease the value of the function. These steps of gradient descent is repeated until convergence.

F.2 Import packages

```
import pandas as pd
import numpy as np
import sklearn
import sklearn.linear_model as LM
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_validate
from sklearn.preprocessing import StandardScaler
import os
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.feature_selection import RFE

training_data = pd.read_csv('train.csv')
featureEngineer_data = pd.read_csv('train.csv')
# Get libraries github
cwd = os.getcwd()
os.chdir(cwd+"/statMLgroup16/core")
%load_ext autoreload
%autoreload
from features import FeatureEngineering as FE
from model_eval import ModelEvaluation
modelClass = ModelEvaluation()
FEClass = FE()
```

```
FEClass.run_feature_engineering(featureEngineer_data, ratios=False,
                                    scaling ='standard', decade = False
                                    )
features = featureEngineer_data[featureEngineer_data.columns.drop('
                                    Lead')]
```

F.3 The model with given features

```
# Model Given Features
attributes = training_data.loc[:,:'Age Co-Lead']
lead = training_data['Lead']
modelGivenFeatures = LM.LogisticRegression(max_iter = 5000)
# Cross Validation
lead_binary = lead.replace(["Male", "Female"],[1,0])
modelClass.cross_val(modelGivenFeatures, attributes, lead_binary, 5) #
                                     cross_val function all the same in
                                     the group
```

F.4 The model with given initial selection features

```
# Model Initial Selection Features
attributesIS = training_data.iloc[:,[1,2,3,4,6,11,12]]
lead = training_data['Lead']
modelInitialSelection = LM.LogisticRegression(max_iter = 5000)
# Cross Validation
lead_binary = lead.replace(["Male", "Female"],[1,0])
modelClass.cross_val(modelInitialSelection, attributesIS, lead_binary,
                                     5) # cross_val function all the
                                    same in the group
```

F.5 Find the optimal features using RFE

```
# Feature engineering
modelBest = LM.LogisticRegression(max_iter = 5000)
lead = training_data['Lead']
lead_binary = lead.replace(["Male", "Female"],[1,0])

print("Features Before: ", features.shape[1])
modelClass.cross_val(modelBest, features, lead_binary, 5) # cross_val
                                    function all the same in the group
#RFE
modelRFE = LM.LogisticRegression(max_iter = 5000)
featuresToSelect = 14 # Found through trial and error
rfe = RFE(estimator = modelRFE, n_features_to_select =
                                    featuresToSelect)
rfe = rfe.fit(features, lead_binary)
bestFeatures = []
attributesIndex = range(0,len(list(features)))
for f, i in zip(list(rfe.ranking_), attributesIndex):
    if f == 1: # Add the highest ranked features to the newFeatures
        bestFeatures.append(i)
print("Features After : ", len(bestFeatures))
newFeatures = features.iloc[:,bestFeatures]
modelClass.cross_val(modelRFE, newFeatures, lead_binary, 5) #
                                    cross_val function all the same in
                                    the group
```

F.6 Hyper parameterize with grid Search

```
# Grid Search
modelGrid = LM.LogisticRegression(max_iter=5000)
grid_parameters = [
                    {'penalty' : ['l1'], 'C': np.logspace(-4,4,20), '
                                                                   solver' : ['
                                                                   liblinear', '
                                                                   saga'] },
                    {'penalty' : ['l2'], 'C': np.logspace(-4,4,20), '
                                                                   solver' : ['
                                                                   newton-cg', '
                                                                   lbfgs', '
                                                                   liblinear', '
                                                                   sag', 'saga']},
                    {'penalty' :['none'], 'solver' : ['newton-cg', '
                                                                   lbfgs', 'sag',
                                                                   'saga']},
                    {'penalty' :['elasticnet'], 'l1_ratio' : np.
                                                                   linspace(0,1,10
                                                                   ), 'C': np.
                                                                   logspace(-4,4,
                                                                   20), 'solver' :
                                                                    ['saga']}
                   ]

grid_search = GridSearchCV(modelGrid, param_grid = grid_parameters,
                                   n_jobs=-1,scoring = 'accuracy',
                                   verbose = 1)
lead_binary = lead.replace(["Male", "Female"],[1,0])
grid_search.fit(newFeatures, lead_binary) # Fit with new features
print("Best Parameters: ", grid_search.best_params_)
print("Best Score: ", grid_search.best_score_)
modelBestGrid = LM.LogisticRegression(**grid_search.best_params_,
                                   max_iter = 5000)
modelClass.cross_val(modelBestGrid, newFeatures, lead_binary, 5) #
                                   cross_val function all the same in
                                   the group
```

G Discriminant analysis

Extract original features from dataframe and run feature factory to get new features

```
# Extract original features

original_features = list(df_actors.columns)
original_features.append("Lead numeric")
original_features

# Run feature factory to engineer all features

feature_factory = features.FeatureEngineering()
feature_factory.run_feature_engineering(
    df_actors,
    scaling=False,
    add_numeric_class_column=True,
    total_actors=True,
    relative_shares=True,
    ratios=True,
    decade=True,
    decade_gross_ratio=True,
    differences=True,
    abs_diff=True,
    yearly_mean=True,
    yearly_mean_diff=True,
)
```

Models using the "basic" (given) features

```
# Select data to use and perform scaling
target = df_actors["Lead numeric"]
features = df_actors[original_features].drop(["Lead", "Lead numeric"],
                                    axis="columns")

# Instantiate LDA model
lda_model = LDA()

# Run cross-validation
ModelEvaluation().cross_val(
    lda_model, features, target, nb_folds=5, use_preprocess=True
)

# Instantiate QDA model
qda_model = QDA()

# Run cross-validation
ModelEvaluation().cross_val(
    qda_model, features, target, nb_folds=5, use_preprocess=True
)
```

Models with "select" subset of features from permutation testing

```

# Select data to use
target = df_actors["Lead numeric"]
features = df_actors[
    [
        "Total words",
        "Number of words lead",
        "Difference in words lead and co-lead",
        "Number of female actors",
        "Number of male actors",
        "Age Lead",
        "Age Co-Lead",
        "Year",
    ]
]

# Instantiate LDA model
lda_model = LDA()

# Run cross-validation
ModelEvaluation().cross_val(
    lda_model, features, target, nb_folds=5, use_preprocess=True
)

# Instantiate QDA model
qda_model = QDA()

# Run cross-validation
ModelEvaluation().cross_val(
    qda_model, features, target, nb_folds=5, use_preprocess=True
)
```

Features based on manual inspection

Number words female added compared to the previous model.

```
# Select data to use
target = df_actors["Lead numeric"]
features = df_actors[
    [
        "Number words female",
        "Total words",
        "Number of words lead",
        "Difference in words lead and co-lead",
        "Number of female actors",
        "Number of male actors",
        "Age Lead",
        "Age Co-Lead",
        "Year",
    ]
]

# Instantiate LDA model
lda_model = LDA()

# Run cross-validation
ModelEvaluation().cross_val(
    lda_model, features, target, nb_folds=5, use_preprocess=True
)

# Instantiate QDA model
qda_model = QDA()

# Run cross-validation
ModelEvaluation().cross_val(
    qda_model, features, target, nb_folds=5, use_preprocess=True
)
```

Model with the best performance

```

# Select data to use
target = df_actors["Lead numeric"]
features = df_actors[
    [
        "Total words",
        "Number words female",
        "Female word share",
        "Number of words lead",
        "Lead word share",
        "Difference in words lead and co-lead",
        "Total actors",
        "Female actor share",
        "Age Lead",
        "Age Co-Lead",
        "Mean Age Male",
        "Mean Age Female",
        "Decade numeric",
    ]
]

features = features.replace([np.inf, -np.inf, np.nan], 0)

# Instantiate LDA model
lda_model = LDA()

# Run cross-validation
ModelEvaluation().cross_val(
    lda_model, features, target, nb_folds=5, use_preprocess=True
)

# Instantiate QDA model
qda_model = QDA()

# Run cross-validation
ModelEvaluation().cross_val(
    qda_model, features, target, nb_folds=5, use_preprocess=True
)
```

Code to run sequential feature selection

```
from sklearn.feature_selection import SequentialFeatureSelector

# Select data to use
target = df_actors["Lead numeric"]
features = df_actors.drop(["Lead", "Lead numeric", "Decade"], axis="
                                   columns")

qda_model = QDA()

sfs = SequentialFeatureSelector(
    qda_model,
    n_features_to_select="auto",
    tol=0.01,
    direction="forward",
    scoring="accuracy",
    cv=5,
)

features = features.replace([np.inf, -np.inf, np.nan], 0)

sfs.fit(features, target)

feature_select_output = pd.DataFrame(columns=["feature", "support"])

i = 0
for column in features.columns:

    row = {
        "feature": column,
        "support": sfs.support_[i],
    }
    feature_select_output = feature_select_output.append(row,
                                           ignore_index=True)
    i += 1

selected_features = sfs.get_support(1)
features_to_use = features[features.columns[selected_features]]
feature_select_output
```

Model with features from sequential feature selection

```
# Select data to use
target = df_actors["Lead numeric"]
features = features_to_use

# Instantiate LDA model
lda_model = LDA()

# Run cross-validation
ModelEvaluation().cross_val(
    lda_model, features, target, nb_folds=5, use_preprocess=True
)

# Instantiate QDA model
qda_model = QDA()

# Run cross-validation
ModelEvaluation().cross_val(
    qda_model, features, target, nb_folds=5, use_preprocess=True
)
```

Code to generate distribution of test accuracy

```
# Select data to use
target = df_actors["Lead numeric"]
features = df_actors[
    [
        "Total words",
        "Number words female",
        "Female word share",
        "Number of words lead",
        "Lead word share",
        "Difference in words lead and co-lead",
        "Total actors",
        "Female actor share",
        "Age Lead",
        "Age Co-Lead",
        "Mean Age Male",
        "Mean Age Female",
        "Decade numeric",
    ]
]

features = features.replace([np.inf, -np.inf, np.nan], 0)
features = StandardScaler().fit_transform(features)

test_accuracies = []

for i in range(1000):

    x_train, x_test, y_train, y_test = train_test_split(features,
                                       target, test_size=0.2)
    qda_model = QDA()
    qda_model.fit(x_train, y_train)
    score = qda_model.score(x_test, y_test)
    test_accuracies.append(score)

import seaborn as sns
import matplotlib.pyplot as plt

sns.set_theme()

plt.figure(figsize=(8, 4))
sns.kdeplot(test_accuracies, fill=True, bw_adjust=0.8)

# Formatting
plt.title(
    "QDA model: Distribution of test accuracies across 1,000 train-
                                       test samples",
    fontsize=14,
    pad=20,
)
plt.xlabel("Accuracy", fontsize=10)
plt.ylabel("Distribution density", fontsize=10)

print()
plt.show()
```

H K-nearest neighbor

Importing all the libraries

```
import pandas as pd
import sklearn.neighbors as skl_nb
import sklearn.preprocessing as skl_pre
import numpy as np
```

H.1 Hyperparameter tuning

```
best = []
for metrix in ['euclidean', 'manhattan', 'chebyshev']:
    for weights in ['uniform', 'distance']:
        accuracy_list = []
        # k is not restricted as distance can work we even numbers
        for k in range(30):
            model_knn = skl_nb.KNeighborsClassifier(n_neighbors=k+1,
                                                   weights= weights,
                                                   metric = metrix)
            labels = df_movies["Lead"]

            #add your selected features here

            data_cross = df_movies[["Difference in words lead and co-
                                                   lead","Female actor
                                                   share","Difference
                                                   Words Gender","Number
                                                   words female", "Male
                                                   actor share", "Number
                                                   of female actors"]]
            cv_output = cross_validate(model_knn,data_cross,labels,cv=
                                                   5,scoring=["accuracy"]
                                                   ,)

            # Calculate key metrics for output
            accuracy = cv_output["test_accuracy"]
            avg_acc, min_acc, max_acc = np.mean(accuracy), np.min(
                                                   accuracy), np.max(
                                                   accuracy)
            print(metrix, "and", k+1, weights)
            print(avg_acc)
            accuracy_list.append(avg_acc)
        best.append(max(accuracy_list))
        best
        print("bestes", max(best))
        K = np.linspace(1,30,30)
        plt.plot(K, accuracy_list, ".")
        plt.show
```

H.2 Code to produce models

Prep. for all models

```
from model_eval import ModelEvaluation
from features import  FeatureEngineering

engineering = FeatureEngineering()
df_movies = engineering.run_feature_engineering(df_movies, ratios=
                                   False)
labels = df_movies["Lead"]
```

First model

```
df_movies = df_movies[['Lead','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', 'Total actors']]

model_knn = skl_nb.KNeighborsClassifier(n_neighbors=6,weights="
                                   distance", metric="euclidean")
data_cross = df_movies.drop(["Lead numeric"],axis=1)

cv_output = cross_validate(model_knn,data_cross,labels,cv=5,
                                   return_train_score=True)
```

First Selection

```
df_movies = df_movies[['Lead','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', 'Total actors']]

model_knn = skl_nb.KNeighborsClassifier(n_neighbors=10, weights="
                                   distance", metric="chebyshev")
data_cross = df_movies.drop(["Lead", "Number words male", "Gross", "
                                   Mean Age Female", "Mean Age Male",
                                   "Number words female"],axis=1)

cv_output = cross_validate(model_knn,data_cross,labels,cv=5,
                                   return_train_score=True)
```

best Model K= 7

```
from sklearn.neighbors import KNeighborsClassifier as KNN
from mlxtend.feature_selection import SequentialFeatureSelector as SFS

features = df[df.columns.drop("Lead")]
target = df["Lead"]
# Split data
X_train, X_test, y_train, y_test = train_test_split(features, target)
knn = KNN(n_neighbors=7)


sfs5 = SFS(knn, k_features=5).fit(X_train, y_train)

print(sfs5.k_feature_names_)
#chebyshev and 6 uniform
model_knn = skl_nb.KNeighborsClassifier(n_neighbors=6, weights="
                                   uniform", metric="chebyshev")
data_cross = df_movies[["Difference in words lead and co-lead",
        "Female word share",
        "Male actor share",
        "Yearly mean diff Number of words lead",
        "Yearly mean diff Difference in words lead and co-lead"]]

model_evaluation = ModelEvaluation()
```

```
#[["Difference in words lead and co-lead", "Female actor share", "Lead
                                    word share", "Difference Words
                                   Gender", "Difference Age Lead"]]



model_evaluation.cross_val(model_knn,features=data_cross, target=
                                   labels)

\textbf{Best Model}

The process including removing features based on Sequential feature
                                   Selection, And then running the own
                                    written exhaustive hyperparameter
                                   optimization.

\begin{python}
from sklearn.neighbors import KNeighborsClassifier as KNN
from mlxtend.feature_selection import SequentialFeatureSelector as SFS

features = df[df.columns.drop("Lead")]
target = df["Lead"]
# Split data
X_train, X_test, y_train, y_test = train_test_split(features, target)
knn = KNN(n_neighbors=7)


sfs5 = SFS(knn, k_features=5).fit(X_train, y_train)

print(sfs5.k_feature_names_)
#chebyshev and 6 uniform
model_knn = skl_nb.KNeighborsClassifier(n_neighbors=6, weights="
                                   uniform", metric="chebyshev")
data_cross = df_movies[["Difference in words lead and co-lead",
        "Female word share",
        "Male actor share",
        "Yearly mean diff Number of words lead",
        "Yearly mean diff Difference in words lead and co-lead"]]

model_evaluation = ModelEvaluation()

#[["Difference in words lead and co-lead", "Female actor share", "Lead
                                    word share", "Difference Words
                                   Gender", "Difference Age Lead"]]



model_evaluation.cross_val(model_knn,features=data_cross, target=
                                   labels)
```

Previous other models:

```
#euclidean and 14 uniform
#model_knn = skl_nb.KNeighborsClassifier(n_neighbors=8, weights="
                                   distance", metric="euclidean")
#data_cross = df_movies[["Number words female", "Difference in words
                                   lead and co-lead", "Number of
                                   female actors", "Number words male
                                   "]]

#model_knn = skl_nb.KNeighborsClassifier(n_neighbors=7, weights="
                                   distance", metric="manhattan")
#data_cross = df_movies[["Number words female", "Difference in words
                                   lead and co-lead", "Number of
```

```
                                    female actors", "Number words male
                                    "]]

#new best
#euclidean and 10 uniform
#model_knn = skl_nb.KNeighborsClassifier(n_neighbors=10, weights="
                                    uniform", metric="euclidean")
#data_cross = df_movies[["Number words female", "Difference in words
                                    lead and co-lead", "Number of
                                    female actors", "Number words male
                                    ", "Female actor share"]]

#new best Lead word share, manhattan and 6 distance

#model_knn = skl_nb.KNeighborsClassifier(n_neighbors=6, weights="
                                    distance", metric="manhattan")
#data_cross = df_movies[["Number words female", "Difference in words
                                    lead and co-lead", "Number of
                                    female actors", "Number words male
                                    ", "Female actor share", "Lead word
                                     share"]]

#new best
#model_knn = skl_nb.KNeighborsClassifier(n_neighbors=4, weights="
                                    distance", metric="euclidean")
#data_cross = df_movies[["Number words female", "Difference in words
                                    lead and co-lead", "Number of
                                    female actors", "Female actor share
                                    ", "Lead word share", "Difference
                                    Words Gender"]]

#new best euclidean and 4 distance
#model_knn = skl_nb.KNeighborsClassifier(n_neighbors=4, weights="
                                    distance", metric="euclidean")
#data_cross = df_movies[["Difference in words lead and co-lead", "
                                    Female actor share", "Lead word
                                    share", "Difference Words Gender"]]

#manhattan and 4 distance new best
#model_knn = skl_nb.KNeighborsClassifier(n_neighbors=4, weights="
                                    distance", metric="manhattan")
#data_cross = df_movies[["Difference in words lead and co-lead", "
                                    Female actor share", "Lead word
                                    share", "Difference Words Gender",
                                    "Difference Age Lead"]]

#chebyshev and 8 uniform
model_knn = skl_nb.KNeighborsClassifier(n_neighbors=8, weights="
                                    uniform", metric="chebyshev")
data_cross = df_movies[["Difference in words lead and co-lead", "
                                    Female actor share", "Difference
                                    Words Gender", "Difference Age Lead
                                    ", "Number words female"]]

model_evaluation = ModelEvaluation()

model_evaluation.cross_val(model_knn,features=data_cross, target=
                                    labels)
```

I Boosting

I.1 Hyperparameters

For **AdaBoost** three different hyperparameters were used, these are *estimator, learning_rate* and *n_estimators.* The estimator is a hyperparameter where you choose which "weak" classifier you will ensemble in your AdaBoost. A common base estimator to use is a decision tree classifier which has its own hyperparameters e.g. max_depth. The weight of the base models is calculated using the exponential loss function[10]:

$$W = e^{-y \cdot f(X)} \tag{4}$$

We can add the hyperparameter learning_rate to equation (4) which generates the following equation:

$$W = \ell e^{-y \cdot f(X)}$$

The default learning rate is set to 1, a lower learning rate decreases the contribution for each classifier in the ensemble model. The final hyperparameter used for AdaBoost is number of estimators (n_estimators), the default for AdaBoost is 50 estimators. Number of estimators is the maximum number of estimators that AdaBoost will go through, in case of perfect fit the procedure will stop early [19].

For **XGBoost** nine different hyperparameters were used, however, only seven of these were tuned using RandomizedSearchCV. The hyperparameters which were not tuned using RandomizedSearchCV were *objective* and *scale_pos_weight.* The objective is the learning objective of the boosting algorithm, in this case *"binary: logistic"* was used which is suitable since we are doing a binary classification, i.e. classifying either "Male" or "Female". The loss function which we want to minimize is a logistic loss function which looks as follows [10]:

$$\frac{1}{n}\sum_{i=1}^{n} ln(1 + e^{-y_i \theta^T x_i})$$

The *scale_pos_weight* is a hyperparameter which controls the balance of positive and negative weights, i.e. the tendency for the model to predict "Male" and "Female" across our dataset. In an imbalanced dataset as we have this hyperparameter is a suitable choice. We make an assumption that the distribution will look similar for future data, thus, the scale_pos_weight was calculated as follows [21]:

$$\frac{\sum Negative\ values}{\sum Positive\ values} \Longrightarrow \frac{\#\ Female\ lead}{\#\ Male\ lead} = \frac{254}{785} \approx 0.3$$

Since 30% of the dataset consists of female leads and 70% consists of male leads the scale_pos_weight was initially set to 3, however, we noticed that we increased performance by decreasing the hyperparameter to 2.5, hence, that value was the final one chosen.

The remaining seven hyperparameters which were tuned using RandomizedSearchCV are the following *colsample_bytree, gamma, learning_rate, min_child_weight, max_depth, n_estimators* and *subsample*. Our main objective for this classification problem is to increase the accuracy of our predictions, however, accuracy and overfitting is heavily correlated. Hence, tuning these hyperparameters aim to increase accuracy and reduce overfitting simultaneously. The hyperparameters max_depth, n_estimators and max_depth are similar to the hyperparameters used in *AdaBoost*, where max_depth is the maximum depth of the decision trees used as estimators in XGBoost. These hyperparameters mainly contribute to the increase of performance, however, if unwisely chosen they will lead to overfitting. Furhter, *min_child_weight* is a hyperparameter which can reduce overfitting above 1 is chosen. The min_child_weight determines the minimum sum of instance weight needed in a child [21]. Hence, XGBoost will be more conservative if the value is above 1. However, in our case we are using a value which is below 1 which mainly contributes to higher performance but can also contribute to overfit [21].

The last hyperparameters *colsample_bytree, gamma* and *subsample* primarilty contributes to reduce overfit. The hyperparameter colsample_bytree is a subsample ratio of which columns or in our case which features will be randomly selected when constructing each tree. Gamma is a regularization parameter which determines the minimum loss reduction required to make further partition on a leaf node [21]. Subsample is a parameter which determines what ratio of the training data that will be randomly selected in each training instance which helps reduce overfit [21].

Best AdaBoost model and its features

```
 # Create Hyperparameter tuned AdaBoost
Ada_tuned = AdaBoostClassifier(
    tree(max_depth=2), learning_rate=0.10180606012821478, n_estimators
                                  =250
)
# Evaluate AdaBoost tuned
CV.cross_val(Ada_tuned, select_features_12, target, 5)
```

```
 select_features_12 = ['Number of words lead', 'Difference in words
                                    lead and co-lead', 'Number of male
                                     actors', 'Female word share', '
                                    Male word share', 'Lead word share
                                    ', 'Female actor share', 'Male
                                    actor share', 'Decade', '
                                    Difference Age Lead', 'Yearly mean
                                     Number words male', 'Yearly mean
                                    Mean Age Male']
```

Best XGBoost model and its features

```
 #Fine tuned XGB model
xgb_model = xgb.XGBClassifier(
    objective="binary:logistic",
    scale_pos_weight=2.5,
    colsample_bytree=0.9556494672664894,
    gamma=0.9536326078296895,
    learning_rate=0.06284905813788089,
    min_child_weight=0.17909493932818155,
    max_depth=6,
    n_estimators=400,
    subsample=0.20610889671677007,
)

#Evaluate
CV.cross_val(xgb_model, best_features, target, 5)
```

```
    best_features = ['Number of words lead', 'Difference in words lead
                                     and co-lead', 'Number of
                                    female actors', 'Age Co-Lead',
                                    'Female word share', 'Male word
                                     share', 'Lead word share', '
                                    Female actor share', '
                                    Difference Actors', 'Difference
                                     Age Lead', 'Difference Mean
                                    Age', 'Yearly mean diff Number
                                    of male actors']
```

I.2 AdaBoost code

```
    import os
import numpy as np
import pandas as pd
from __future__ import annotations
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import (
    cross_val_score,
    GridSearchCV,
    KFold,
    RandomizedSearchCV,
)
from sklearn.preprocessing import (
    MaxAbsScaler,
    MinMaxScaler,
    RobustScaler,
    StandardScaler,
)
from scipy.stats import uniform, randint
from sklearn.ensemble import AdaBoostClassifier
```

```
# Load data
df_initial = pd.read_csv("data/train.csv")

# Get libraries
cwd = os.getcwd()
os.chdir(cwd + "\statMLgroup16\\core")
%load_ext autoreload
%autoreload
from features import FeatureEngineering
from model_eval import ModelEvaluation
```

```
    # Initiate FeatureEngineering and ModelEvaluation class
FE = FeatureEngineering()
CV = ModelEvaluation()
```

```
df = df_initial.copy()

# Preprocess step, scale data
FE.scaling(df, method="standard")

# Encode class column to numeric values
FE.encode_class_column(df)

# Get Features and Target
features = df[df.columns.drop(["Lead"])]
target = df["Lead"]
```

```
# Create AdaBoost Classifier
Ada = AdaBoostClassifier()

# Evaluate Adaboost Basic
CV.cross_val(Ada, features, target, 5)
```

```
initial_features = [
    "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",
    "Age Lead",
    "Age Co-Lead",
]

# Create AdaBoost Classifier
Ada = AdaBoostClassifier()
```

```
# Evaluate Adaboost Initial Select
CV.cross_val(Ada, df[initial_features], target, 5)
```

Step 1: Feature Engineering
AdaBoost uses an exponential loss function, therefore, it is sensitive to outliers and uncertain data. We will try to scale the features with different scaling method to see if it affects the performance.

```
    # Create AdaBoost Classifier
Ada = AdaBoostClassifier()

# Initialize unscaled df
df = df_initial.copy()

# Scaling with standard
FE.scaling(df, "Standard")
features = df[df.columns.drop("Lead")]
target = df["Lead"]

# Evaluate Adaboost
CV.cross_val(Ada, features, target, 5)

# Initialize unscaled df
df = df_initial.copy()

# Scaling with MinMax
FE.scaling(df, "MinMax")
features = df[df.columns.drop("Lead")]
target = df["Lead"]

# Evaluate Adaboost
CV.cross_val(Ada, features, target, 5)

# Initialize unscaled df
df = df_initial.copy()

# Scaling with MaxAbs
FE.scaling(df, "MaxAbs")
features = df[df.columns.drop("Lead")]
target = df["Lead"]

# Evaluate Adaboost
CV.cross_val(Ada, features, target, 5)

# Initialize unscaled df
df = df_initial.copy()

# Scaling with Robust
FE.scaling(df, "Robust")
features = df[df.columns.drop("Lead")]
target = df["Lead"]

# Evaluate Adaboost
CV.cross_val(Ada, features, target, 5)
```

Scaling did not improve nor deteriorate performance. Let us now try to create new features.

```
# Create AdaBoost Classifier
Ada = AdaBoostClassifier()

# Initialize unscaled df
df = df_initial.copy()

# Create features from feature engineering class
FE.run_feature_engineering(df, ratios=False)
```

```python
# Get features and target from df
features = df[df.columns.drop("Lead")]
target = df["Lead"]

# Evaluate Adaboost with all feature engineered features
CV.cross_val(Ada, features, target, 5)
```

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Use Spearman correlation to measure the correlation between features
                                    and output 'lead'
corr = df[df.columns[1:]].corr(method="spearman")["Lead"][:]
corr = corr.drop(labels=["Lead"])

# Prepare data
corr2 = corr.sort_values(ascending=False)
features_ = list(corr2.keys())
values = []
for feature in features_:
    values.append(corr2[feature])

# Plot correlation
sns.set_theme(style="darkgrid")
f, ax = plt.subplots(figsize=(10, 10))
diverging_colors = sns.color_palette("coolwarm", 10)
sns.barplot(x=values, y=features_, palette=diverging_colors)
ax.set(
    xlim=(-0.5, 0.5),
    ylabel="",
    xlabel="Correlation",
    title="Features' correlation to male output",
)
```

```python
# Plot heatmap of correlation
f, ax = plt.subplots(figsize=(15, 12))
sns.heatmap(df.corr(), annot=True, linewidths=0.5, fmt=".1f", ax=ax)
ax.set(title="Heatmap - Correlation of features")
```

```python
    def plotFeatureImportances(features, importances, modelName):
    y_pos = np.arange(features.size)
    plt.clf()
    indexes = np.argsort(importances)
    plt.figure(figsize=(10, 20))
    plt.title("Feature importances - " + modelName)
    plt.barh(y_pos, np.sort(importances))
    plt.yticks(y_pos, features[indexes])
    plt.xlabel("F score")
    plt.ylabel("Feature")
    plt.show()


# Create AdaBoost Classifier
Ada = AdaBoostClassifier()
Ada.fit(features, target)


plotFeatureImportances(features.columns, Ada.feature_importances_, "
                                   AdaBoost")
```

Creating a bunch of new features seems to improve performance a bit, however, there are quite many of the features which correlates with each other, we should try to reduce the amount of features which add no additional information for our model and instead keep the best which should hopefully improve the performance of our model.

Feature Selection
We choose the features which have low feature importance to our model.

```
 # Find all features which have less than 0.02 importance and add them
                                                        to list
zero_features = [
    i
    for i in range(len(Ada.feature_importances_))
    if Ada.feature_importances_[i] <= 0.02
]

# Drop all features with less than 0.02 importance
sel_features = features.drop(features.columns[zero_features], axis=1)
sel_features.head()
```

```
from sklearn.feature_selection import SelectFromModel

# split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(features, target)
# Fit model
Ada = AdaBoostClassifier(tree(max_depth=2))
Ada.fit(X_train, y_train)

# Create thresholds from feature importance
thresholds = np.sort(Ada.feature_importances_)
print(thresholds)

for thresh in thresholds:
    # Select features using threshold
    selection = SelectFromModel(Ada, threshold=thresh, prefit=True)
    select_features = selection.transform(X_train)
    feature_index = selection.get_support()
    select_features = df[df.columns.drop("Lead")[feature_index]]

    # Evaluate selected features
    Ada = AdaBoostClassifier(tree(max_depth=2))
    Ada.fit(X_train, y_train)
    print(thresh)
    CV.cross_val(Ada, select_features, target, 5)
```

```
# Best feature threshold
best_thresh = 0.016752652340812434

# Select features
selection = SelectFromModel(Ada, threshold=best_thresh, prefit=True)
select_features = selection.transform(X_train)
feature_index = selection.get_support()
select_features = df[df.columns.drop("Lead")[feature_index]]

# Evaluate Adaboost with features from threshold
CV.cross_val(Ada, select_features, target, 5)

# Evaluate Adaboost with selected features with low importance
CV.cross_val(Ada, sel_features, target, 5)
```

```
print("Features chosen with threhold:")
print(select_features.columns)
print("Features chosen with low importance")
print(sel_features.columns)
```

```
from mlxtend.feature_selection import SequentialFeatureSelector as SFS

# Split data
X_train, X_test, y_train, y_test = train_test_split(features, target)

# Initiate model
Ada = AdaBoostClassifier()

# Choose 12 best features through sequential feature selection
sfs12 = SFS(Ada, k_features=12).fit(X_train, y_train)
print(sfs12.k_feature_names_)
```

```
select_features_12 = df[
    [
        "Number of words lead",
        "Difference in words lead and co-lead",
        "Number of male actors",
        "Female word share",
        "Male word share",
        "Lead word share",
        "Female actor share",
        "Male actor share",
        "Decade",
        "Difference Age Lead",
        "Yearly mean Number words male",
        "Yearly mean Mean Age Male",
    ]
]
# Evaluate performance
CV.cross_val(Ada, select_features_12, target, 5)
```

Feature selection did improve the performance a bit.

Hyperparameter tuning

We do some hyperparameter tuning which could improve the model's performance.

```
def report_best_scores(results, n_top=3):
    for i in range(1, n_top + 1):
        candidates = np.flatnonzero(results["rank_test_score"] == i)
        for candidate in candidates:
            print("Model with rank: {0}".format(i))
            print(
                "Mean validation score: {0:.3f} (std: {1:.3f})".format
                                                        (
                    results["mean_test_score"][candidate],
                    results["std_test_score"][candidate],
                )
            )
            print("Parameters: {0}".format(results["params"][candidate
                                                    ]))
            print("")
```

```
#Split data
X_train, X_test, y_train, y_test = train_test_split(select_features_12
                                   , target)

#Choose parameter values to search through
params = {
    "learning_rate": uniform(0.01, 0.3),  # default 0.1
    "n_estimators": [10, 50, 100, 150, 200, 250, 300, 350, 400],  #
                                        default 100
    "base_estimator": [
        tree(max_depth=1),
        tree(max_depth=2),
        tree(max_depth=3),
        tree(max_depth=4),
    ],
}

#Create AdaBoost model
Ada = AdaBoostClassifier()

#Search different parameter values
search = RandomizedSearchCV(
    Ada,
    param_distributions=params,
    random_state=42,
    n_iter=1000,
    cv=5,
    verbose=1,
    n_jobs=5,
    scoring="accuracy",
    return_train_score=True,
)

#Fit the search
search.fit(X_train, y_train)

#Print the best result of parameter values
report_best_scores(search.cv_results_, 1)
```

```
# Create Hyperparameterer tuned AdaBoost
Ada_tuned = AdaBoostClassifier(
    tree(max_depth=2), learning_rate=0.10180606012821478, n_estimators
                                       =250
)
# Evaluate AdaBoost tuned
CV.cross_val(Ada_tuned, select_features_12, target, 5)
```

I.3 XGBoost code

```
import os
import numpy as np
import pandas as pd
from __future__ import annotations
import xgboost as xgb
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import (
    cross_val_score,
    GridSearchCV,
    KFold,
    RandomizedSearchCV,
)
from scipy.stats import uniform, randint
```

```
# Load data
df = pd.read_csv("data/train.csv")

# Get libraries
cwd = os.getcwd()
os.chdir(cwd + "\statMLgroup16\\core")
%load_ext autoreload
%autoreload
from features import FeatureEngineering
from model_eval import ModelEvaluation
```

Basic XGBoost

```
#Initiate classes
FE = FeatureEngineering()
CV = ModelEvaluation()

# Encode class column
FE.encode_class_column(df)

# Create XGBoost model
xgb_model = xgb.XGBClassifier(objective="binary:logistic")

#Extract features and target
features = df[df.columns.drop("Lead")]
target = df["Lead"]

# Evaluate basic XGBoost
CV.cross_val(xgb_model, features, target, 5)
```

We can see at the training accuracy that the model is overfitting. Before we try to improve the model we need to choose hyperparameters that reduces this overfitting.

Hyperparameter tuning to reduce overfitting

```
#Split into training and validation data
X_train, X_test, y_train, y_test = train_test_split(features, target)

#Parameter values to search from
params = {
    "colsample_bytree": uniform(0, 1),
    "gamma": uniform(0, 1),
    "learning_rate": uniform(0.03, 0.3),  # default 0.1
    "max_depth": randint(2, 9),  # default 3
    "n_estimators": randint(10, 250),  # default 100
    "subsample": uniform(0, 1),
}

#Initiate XGB model with scale_pos_weight=3 since data is unevenly
                                   distributed
xgb_model = xgb.XGBClassifier(objective="binary:logistic",
                                   scale_pos_weight=3)

#Search through parameter values
search = RandomizedSearchCV(
    xgb_model,
    param_distributions=params,
    random_state=42,
    n_iter=200,
    cv=3,
    verbose=1,
    n_jobs=1,
    scoring="roc_auc",
    return_train_score=True,
)

#Fit the searched parameters
search.fit(X_train, y_train)

#Print the best parameter values found
report_best_scores(search.cv_results_, 1)
```

```
Parameters: {'colsample_bytree': 0.9915346248162882,
'gamma': 0.4812236474710556,
'learning_rate': 0.10553468874760924,
'max_depth': 3,
'n_estimators': 140,
'subsample': 0.7203513239267079,
min_child_weight=1.42,}
#allows training without overfitting.
```

```
#Create tuned XGB model
xgb_model = xgb.XGBClassifier(
        objective="binary:logistic",
        scale_pos_weight=3,
        colsample_bytree=0.99,
        gamma=0.48,
        learning_rate=0.106,
        min_child_weight=1.42,
        max_depth=4,
        n_estimators=140,
        subsample=0.72,
)

# Evaluate tuned model
CV.cross_val(xgb_model, features, target, 5)
```

Feature Engineering

```
FE.run_feature_engineering(df, ratios=False, scaling="Standard")
df.head()
```

```
from matplotlib import pyplot as plt

def my_plot_importance(booster, figsize, **kwargs):
    from xgboost import plot_importance

    fig, ax = plt.subplots(1, 1, figsize=figsize)
    return plot_importance(booster=booster, ax=ax, **kwargs)

# Plot feature importance
my_plot_importance(xgb_model, (10, 20))
```

```
from sklearn.feature_selection import SelectFromModel

# split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(features, target)
# Fit model
xgb_model = xgb.XGBClassifier(
    objective="binary:logistic",
    scale_pos_weight=3,
    colsample_bytree=0.9,
    gamma=0.107,
    learning_rate=0.0974733164896875,
    min_child_weight=0.04586126640463295,
    max_depth=8,
    n_estimators=450,
    subsample=0.2,
)
model.fit(X_train, y_train)

# Create thresholds from feature importance
thresholds = np.sort(model.feature_importances_)

for thresh in thresholds:
    # Select features using threshold
    selection = SelectFromModel(model, threshold=thresh, prefit=True)
    select_features = selection.transform(features)
    feature_index = selection.get_support()
    select_features = df[df.columns.drop("Lead")[feature_index]]

    # Evaluate selected features
    xgb_model = xgb.XGBClassifier(
    objective="binary:logistic",
    scale_pos_weight=3,
    colsample_bytree=0.9,
    gamma=0.107,
    learning_rate=0.0974733164896875,
    min_child_weight=0.04586126640463295,
    max_depth=8,
    n_estimators=450,
    subsample=0.2,
    )
    print(thresh)
    CV.cross_val(xgb_model, select_features, target, 5)
```

By looking through all thresholds we find the one which scores the highest average acurracy at threshold = 0.015246303

```
from sklearn.feature_selection import SelectFromModel

# split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(features, target)
# Fit model
xgb_model = xgb.XGBClassifier(
    objective="binary:logistic",
    scale_pos_weight=3,
    colsample_bytree=0.9,
    gamma=0.107,
    learning_rate=0.0974733164896875,
    min_child_weight=0.04586126640463295,
    max_depth=8,
    n_estimators=450,
    subsample=0.2,
)
model.fit(X_train, y_train)

# Create thresholds from feature importance
thresholds = np.sort(model.feature_importances_)

for thresh in thresholds:
    # Select features using threshold
    selection = SelectFromModel(model, threshold=thresh, prefit=True)
    select_features = selection.transform(features)
    feature_index = selection.get_support()
    select_features = df[df.columns.drop("Lead")[feature_index]]

    # Evaluate selected features
    xgb_model = xgb.XGBClassifier(
    objective="binary:logistic",
    scale_pos_weight=3,
    colsample_bytree=0.9,
    gamma=0.107,
    learning_rate=0.0974733164896875,
    min_child_weight=0.04586126640463295,
    max_depth=8,
    n_estimators=450,
    subsample=0.2,
    )
    print(thresh)
    CV.cross_val(xgb_model, select_features, target, 5)
```

```
#Select the highest accuracy features
```

```
selection = SelectFromModel(model, threshold=0.015246303, prefit=True)
select_features = selection.transform(features)
feature_index = selection.get_support()
select_features = df[df.columns.drop("Lead")[feature_index]]

#Print best features
print("Selected best features:")
print(select_features.columns)
```

```
# Output:
Selected best features:
Index(['Number words female', 'Difference in words lead and co-lead',
       'Number of female actors', 'Number words male', 'Mean Age Male'
                                       ,
       'Mean Age Female', 'Total actors', 'Female word share',
       'Female actor share', 'Difference Words Gender', 'Difference
                                       Age Lead',
       'Difference Mean Age', 'Yearly mean Total words',
       'Yearly mean Difference in words lead and co-lead', 'Yearly
                                       mean Gross',
       'Yearly mean Mean Age Male', 'Yearly mean Mean Age Female',
       'Yearly mean diff Number words female',
       'Yearly mean diff Difference in words lead and co-lead',
       'Yearly mean diff Number of male actors',
       'Yearly mean diff Number of female actors',
       'Yearly mean diff Number words male', 'Yearly mean diff Mean
                                       Age Male',
       'Yearly mean diff Mean Age Female', 'Yearly mean diff Age Lead'
                                       ,
       'Yearly mean diff Age Co-Lead'],
      dtype='object')
```

```
# Evaluate selected best features
CV.cross_val(xgb_model, select_features, target, 5)
```

It did improve the performance with quite some much, now let attempt to optimize it with another hyperparameter optimization.

```
#Split data
X_train, X_test, y_train, y_test = train_test_split(select_features,
                                   target)

#Define search space
params = {
    "colsample_bytree": uniform(0, 1),
    "gamma": uniform(0, 1),
    "learning_rate": uniform(0.03, 0.3),  # default 0.1
    "max_depth": randint(2, 12),  # default 3
    "min_child_weight": uniform(0, 1),
    "n_estimators": randint(10, 500),  # default 100
    "subsample": uniform(0, 0.8),
    "min_child_weight": uniform(0,1)
}

#Create model
xgb_model = xgb.XGBClassifier(
    objective="binary:logistic",
    scale_pos_weight=3,
    # max_depth=3,
    # gamma=0.24,
    # min_child_weight=0.63,
)
```

```
#Search parameter values
search = RandomizedSearchCV(
    xgb_model,
    param_distributions=params,
    random_state=42,
    n_iter=200,
    cv=3,
    verbose=1,
    n_jobs=1,
    scoring="accuracy",
    return_train_score=True,
)

#Fit search space
search.fit(X_train, y_train)

#Print best fit
report_best_scores(search.cv_results_, 1)
```

Note that this process is done multiple times, were we concentrate the search space to find better parameters.

```
#Tuned model
xgb_model = xgb.XGBClassifier(
    objective="binary:logistic",
    scale_pos_weight=3,
    colsample_bytree=0.974,
    gamma=0.209,
    learning_rate=0.2622,
    min_child_weight=0.422,
    max_depth=3,
    n_estimators=100,
    subsample=0.74,
)

#Evaluate performance of tuned model
CV.cross_val(xgb_model, select_features, target, 5)
```

A lot more work was done but then we moved on to attempt using sequential feature selection which improved the performance a bit.

```
#Tuned model
xgb_model = xgb.XGBClassifier(
    objective="binary:logistic",
    scale_pos_weight=3,
    colsample_bytree=0.974,
    gamma=0.209,
    learning_rate=0.2622,
    min_child_weight=0.422,
    max_depth=3,
    n_estimators=100,
    subsample=0.74,
)

#Evaluate performance of tuned model
CV.cross_val(xgb_model, select_features, target, 5)
```

```
feature_select_12 = df[
    [
        "Number of words lead",
        "Difference in words lead and co-lead",
        "Number of female actors",
        "Age Co-Lead",
        "Total actors",
        "Female word share",
        "Male word share",
        "Lead word share",
        "Difference Words Gender",
        "Difference Actors",
        "Difference Age Lead",
        "Yearly mean diff Age Co-Lead",
    ]
]
```

We now proceed with evaluating which features matters the worst out of the ones we have. We do this through shuffle testing and cross evaluation by marking out features one at a time.

```
CV.shuffle_test(xgb_model, best_features, target, 5)
```

After determining which features are performning the worst we proceed with Exhaustive Feature Selection were we fix the best remaining features and try to find 1 additional good features with those features.

```
from mlxtend.feature_selection import ExhaustiveFeatureSelector as EFS

# Split Data
features = df[df.columns.drop("Lead")]
target = df["Lead"]
X_train, X_test, y_train, y_test = train_test_split(features, target)

# Best features with the worst removed
feature_names = [
    "Number of words lead",
    "Difference in words lead and co-lead",
    "Number of female actors",
    "Age Co-Lead",
    "Total actors",
    "Female word share",
    "Male word share",
    "Lead word share",
    "Difference Words Gender",
    "Difference Actors",
    "Difference Age Lead",
    "Gross",
]
# Get feature indeces of the best features
indeces = [df.columns.get_loc(c) for c in feature_names if c in df]
print(indeces)

# Run Exhaustive Feature Selection where best features are fixed,
                                   choose 1 from the outstanding
                                   features.
efs = EFS(
    xgb_model, 13, 13, fixed_features=(2, 3, 6, 12, 14, 15, 16, 17, 21
                                      , 22, 23, 8)
).fit(X_train, y_train)
print(efs.best_feature_names_)
```

After some manual work together with EFS we end up with the following features as our best combination found.

```
best_features = df[
    [
        "Number of words lead",
        "Difference in words lead and co-lead",
        "Number of female actors",
        # "Gross",
        "Age Co-Lead",
        "Female word share",
        "Male word share",
        "Lead word share",
        "Female actor share",
        "Difference Actors",
        "Difference Age Lead",
        "Difference Mean Age",
        "Yearly mean diff Number of male actors",
    ]
]
```

We evaluate it's performance.

```
#Create XGB model
xgb_model = xgb.XGBClassifier(
        objective="binary:logistic",
        scale_pos_weight=3,
        colsample_bytree=0.99,
        gamma=0.48,
        learning_rate=0.106,
        min_child_weight=1.42,
        max_depth=4,
        n_estimators=140,
        subsample=0.72,
)

# Evaluate model
CV.cross_val(xgb_model, best_features, target, 5)
```

We then return to hyperparameter tuning which we perform several times through concentrating parameter ranges.

```
#Split data
X_train, X_test, y_train, y_test = train_test_split(best_features,
                                   target)

#Define search space
params = {
    "colsample_bytree": uniform(0.8, 1.0),
    "gamma": uniform(0.8, 1.0),
    "learning_rate": uniform(0.01, 0.15),  # default 0.1
    # "max_depth": [3, 4, 5, 6, 7, 8, 9, 10, 11, 12],  # default 3
    "min_child_weight": uniform(0, 0.5),
    "n_estimators": [50, 100, 150, 200, 250, 300, 350, 400, 450, 500],
                                          # default 100
    "subsample": uniform(0.15, 0.3),
}

#Initiate model
xgb_model = xgb.XGBClassifier(
    objective="binary:logistic",
    scale_pos_weight=2.5,
    n_estimators=400,
    # subsample=0.69,
    # gamma=0.24,
    max_depth=6,
    # gamma=0.24,
    # min_child_weight=0.63,
)

#Search through parameter ranges
search = RandomizedSearchCV(
    xgb_model,
    param_distributions=params,
    random_state=42,
    n_iter=1000,
    cv=5,
    verbose=1,
    n_jobs=5,
    scoring="accuracy",
    return_train_score=True,
)

#Fit search space
search.fit(X_train, y_train)

# Print best result
report_best_scores(search.cv_results_, 1)
```

We finally end up with a model which performs quite well and does not overfit that much compared to a default tuned XGBoost.

```
#Initiate tuned model
xgb_model = xgb.XGBClassifier(
    objective="binary:logistic",
    scale_pos_weight=2.5,
    colsample_bytree=0.9556494672664894,
    gamma=0.9536326078296895,
    learning_rate=0.06284905813788089,
    min_child_weight=0.17909493932818155,
    max_depth=6,
    n_estimators=400,
    subsample=0.20610889671677007,
)

#Evaluate performance
CV.cross_val(xgb_model, best_features, target, 5)
```

I.4 Code for submitted predictions

The code below was used to make predictions on the external test set

```
# Import standard packages

import numpy as np
import pandas as pd
import jupyter_black

jupyter_black.load()

from sklearn.discriminant_analysis import
                                    QuadraticDiscriminantAnalysis as
                                    QDA
from sklearn.preprocessing import StandardScaler

# Change working directory to core to enable codebase imports

import os

cwd = os.getcwd()
os.chdir(cwd + "\\stat-ML-project\\core")
%load_ext autoreload
%autoreload 2

# Codebase imports

import features
from model_eval import ModelEvaluation, cross_val_legacy

# Load training data
df_train = pd.read_csv("..\\data\\train.csv")
df_train

# Run feature factory to engineer all features

feature_factory = features.FeatureEngineering()
feature_factory.run_feature_engineering(
    df_train,
    scaling=False,
    add_numeric_class_column=False,
    encode_class=False,
    total_actors=True,
    relative_shares=True,
    ratios=True,
    decade=True,
    decade_gross_ratio=True,
    differences=True,
    abs_diff=True,
    yearly_mean=True,
    yearly_mean_diff=True,
)

df_train["Lead numeric"] = np.where(df_train["Lead"] == "Female", 1, 0
                                    )
df_train

# Select data to use
train_target = df_train["Lead numeric"]
train_features = df_train[
    [
        "Total words",
        "Number words female",
        "Female word share",
        "Number of words lead",
        "Lead word share",
        "Difference in words lead and co-lead",
        "Total actors",
        "Female actor share",
        "Age Lead",
        "Age Co-Lead",
        "Mean Age Male",
        "Mean Age Female",
        "Decade numeric",
    ]
]

train_features_scaled = StandardScaler().fit_transform(train_features)

# Instantiate QDA model
qda_model = QDA()

# Fit on the entire data set and predict on training data for sanity
                                   checking
qda_model.fit(train_features_scaled, train_target)
pred_train = qda_model.predict(train_features_scaled)
score = qda_model.score(train_features_scaled, train_target)
score

# Load test data
df_test = pd.read_csv("..\\data\\test.csv")
df_test

# Run feature factory to engineer all features

feature_factory = features.FeatureEngineering()
feature_factory.run_feature_engineering(
    df_test,
    scaling=False,
    add_numeric_class_column=False,
    encode_class=False,
    total_actors=True,
    relative_shares=True,
    ratios=True,
    decade=True,
    decade_gross_ratio=True,
    differences=True,
    abs_diff=True,
    yearly_mean=True,
    yearly_mean_diff=True,
)
df_test

# Select data to use
test_features = df_test[
    [
        "Total words",
        "Number words female",
        "Female word share",
        "Number of words lead",
        "Lead word share",
        "Difference in words lead and co-lead",
        "Total actors",
        "Female actor share",
        "Age Lead",
        "Age Co-Lead",
        "Mean Age Male",
        "Mean Age Female",
        "Decade numeric",
    ]
]

test_features_scaled = StandardScaler().fit_transform(test_features)

# Make predictions on test set and export csv
pred_test = qda_model.predict(test_features_scaled)
pd.DataFrame(pred_test).transpose().to_csv(
    "..\\data\\predictions.csv", header=None, index=None
)
```