Mini Project in Statistical Machine Learning Spring 2021

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

After testing three different algorithms on the data set we decided to proceed using the Logistic Regression algorithm. It was enhanced using Ridge Regularization. (Will add final result here later)

1 Introduction

The goal of the project was to determine whether it is possible to tell if the lead actor of a movie is female or male, based on up to 13 different variables of data, and if so, what algorithm is best suited for doing so. The variables in the data were: 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.

1.1 Cross Validation

The Cross Validation method is a way of testing a classification method against another. The idea behind cross validation is that the data set is split into different combinations of training and testing data, and the model is tested on each one of them. This leads to a better understanding of the model's performance as we basically can generate different training and testing data sets from the same set of data.

2 Methods

The following methods were implemented and tested. The implementation of each method in Python can be found in the appendix.

2.1 Linear Discriminant Analysis

The Linear Discriminant Analysis is a known method for dimensionality reduction, but can also be very useful as a classification method. The goal of the method is to reduce the dimensions by focusing on optimizing separating the classes. The method does this by calculating the separability between its classes, known as the between-class variance, which is the distance between the mean of its different classes [2]. This between-class variance can be calculated and stored in the so called between-class matrix with the following equation [2],

$$S_b = \sum_{i=1}^{g} N_i(\bar{x}_i - \bar{x})(\bar{x}_i - \bar{x})^T \tag{1}$$

where N_i represents the sample size of class i, the $\bar{x}_i$ is the sample mean of class i and the x_i is the overall mean. The next step in the method is to calculate the within-class variance, which is the distance between the mean and the sample of each class. This can be done with the following equation [2],

$$S_w = \sum_{i=1}^{g}(N_i - 1)S_i = \sum_{i=1}^{g}\sum_{j=1}^{N_i}(x_{i,j} - \bar{x})(\bar{x}_{i,j} - \bar{x})^T \tag{2}$$

where the parameters represent the same things as in the previous equation with the addition of $x_{i,j}$ as the sample of each class. The final step in the LDA method is to create a lower-dimensional space that maximizes the separability between the classes and at the same time minimizes variance within the classes. This can be done in several different ways, some examples are by using a least square method, by using the eigenvalues or by using singular value decomposition [2]. The Scikit we used in the project does this by using the singular value decomposition if nothing else is specified. This is preferred since it does not require the computation of a covariance matrix which makes it more efficient for higher dimensions.

The between-class variance (the distance between the mean of different classes) along with the within-class variance (the distance between the mean and the sample of each class) can be seen as measures of how difficult the separation will be. Where a shorter distance in the between-class variance and a longer distance in the within-class variance indicate a more difficult separation.

2.1.1 Evaluation

An important parameter when running the linear discriminant analysis is which solver to use. As we mentioned before the linear discriminant analysis can be run using the least square method, eigenvalues or singular value decomposition which we will henceforth refer to as lsqr, eigen and svd respectively. We used Scikits model-selection package to compare the different solvers. The package include methods which allow you to run repeated k-fold and grid-searches which run the solvers multiple times and then compare their averages. The best solver was svd with an accuracy of 0.865.

Another thing we explored with linear discriminant analysis was if adding a penalty in the form of a shrinkage would increase performance. This is only possible for the lsqr and eigen solvers since svd does not support this feature. The packages mentioned above includes options for testing shrinkage aswell and implementing and testing the shrinkage where the shrinkage varied between 0 and 1 with an increase of 0.01 in each step resulted in no enhancement in performance for the lsqr-solver but yielded a slight improvement for the eigen-solver when a shrinkage of 0.01 was used. However, the mean accuracy for the eigen-solver was still less than the mean accuracy for the scd performance and when taking these two tests in consideration we concluded that the svd-solver is the best one for our particular problem. We decided to proceed with the svd-solver when comparing the linear discriminant analysis with the other methods.

2.2 K-nearest neighbors

The k-nearest neighbors algorithm is a supervised classification algorithm that first was developed in the 1950's. It classifies an object based on the k closest objects from the training data set. The k is a positive, normally small integer that lets the model know how many neighbors the algorithm takes into consideration when classifying each object [**?**]. The choice of k is not clear and there is often data dependent. Different k values can result in different classification results, as seen in Figure 1 and Figure 2 below.

Since the KNN method relies on distance to assign class the data set is almost always normalized to handle different scaling and units in the data.

2.2.1 Evaluation

When testing for different k values from 1 to 23 on the entire normalized data set the KNN method failed to achieve an accuracy higher than in the low to mid seventies. This may seem like a good results, until the fact that the who data set is labeled 75.5 per cent male. The method's accuracy for

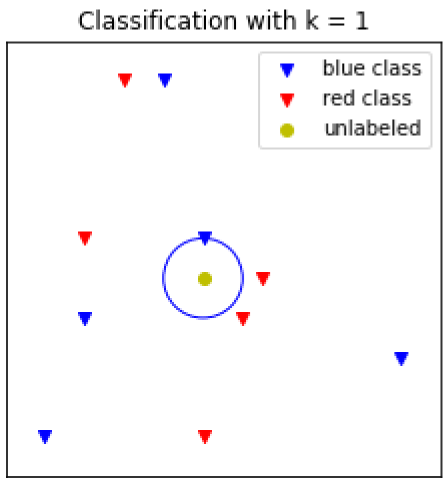

Figure 1: Classification of the yellow dot using k = 1 results in the blue class.

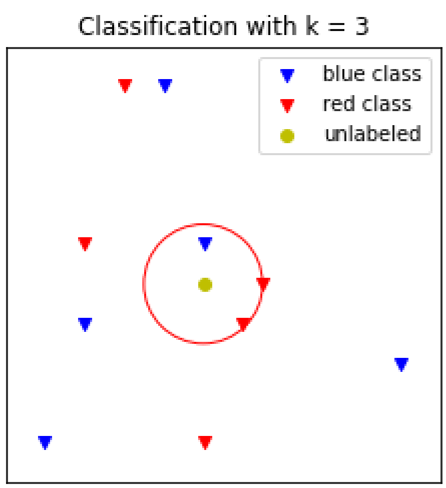

Figure 2: Classification of the yellow dot using k = 3 results in the red class.

predicting male actors was fairly good; ranging from around 0.85 to 0.95, but the number of female actors' accuracy predicted never exceeded 0.50.

As k increased the number of predicted female leads grew lower, this is a result of how skewed the data was towards male actors. For example: for a testing data set of size 159 and $k = 1$ the model predicted 60 female leads (38 % of the total actors), but for $k = 11$ on the same data set only 26 female leads were predicted (16 %). In Table 1 the accuracy and number of predicted females are shown.

When using $k = 3$ and only one of the features in the data set the highest accuracy was achieved when only looking at the number of female actors. The model achieved an accuracy of 0.714, however the model only predicted that 18 of 259 lead actors were female, the lowest out of any single feature. The features Words Spoken by Males & Females, Year of Release and Gross resulted accuracies of 0.617,

Table 1: K-NN results for different K values.

K Value	Accuracy	Predicted Females
1	0.745	60
3	0.733	53
5	0.722	34
7	0.722	36
9	0.737	30
11	0.737	26
13	0.729	20
15	0.729	14
17	0.725	13
19	0.714	8
21	0.722	6
23	0.718	5

0.644, 0.629, each run predicting 53, 26, 46 female leads. When applying ten-fold sampling without replacement to the KNN model returned a lower accuracy and the number of predicted female leads was heavily reduced. We decided not to move forward with the KNN algorithm.

2.3 Logistic Regression

The following section uses content from the course literature [1], in order to explain the logistic regression model as a classifier.

In general, a classification model can be specified in terms of the conditional class probabilities

$$p(y = m|\mathbf{x}) \quad \text{for} \quad m \in 1, ..., M \tag{3}$$

which describes the probability for class $y = 1, ..., M$, given some known input variable(s) $\mathbf{x}$. For binary classification problems, that is $M = 2$ and $y \in \{-1, 1\}$, we can learn a model $f(\mathbf{x})$ that describes the conditional class probability for the positive class $y = 1$

$$f(\mathbf{x}) = p(y = 1|\mathbf{x}) \tag{4}$$

where $0 \leq f(\mathbf{x}) \leq 1$, since it is a model for probability. And given by the law of total probability, we can model the conditional class probability for the negative class $y = -1$ as

$$p(y = -1|\mathbf{x}) = 1 - f(\mathbf{x}) \tag{5}$$

One classification method using conditional probabilities is the *logistic regression model*. The idea of the logistic regression is based on the *linear regression model*, together with conditional probabilities for classification. The linear regression model is given by

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n + \epsilon = \boldsymbol{\theta}^T \mathbf{x} + \epsilon \tag{6}$$

where the output variable z, a numerical value, is assumed to be a linear combination of the input variables $\mathbf{x}$ and unknown model parameters $\boldsymbol{\theta}$, including the intercept term θ_0. The ϵ term accounts for random errors in the data, not explained by the model. The values assigned to the parameters $\boldsymbol{\theta}$ determine the relationship between the input and output variables described by the model.
In order to interpreter the output z in terms of conditional class probabilities, the *logistic function* $h(z) = \frac{e^z}{1+e^z}$ is used, as it is restricted to the interval [0,1]. For binary classification problems, $y \in \{-1, 1\}$, this results in the *logistic regression model*

$$p(y = 1|\mathbf{x}; \boldsymbol{\theta}) = f(\mathbf{x}; \boldsymbol{\theta}) = \frac{e^{\boldsymbol{\theta}^T \mathbf{x}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}}} \tag{7}$$

describing the conditional class probability for the positive class $y = 1$. Again, by the law of total probability, this also gives the logistic regression model for the negative class $y = -1$

$$p(y = -1|\mathbf{x}; \boldsymbol{\theta}) = 1 - f(\mathbf{x}; \boldsymbol{\theta}) = 1 - \frac{e^{\boldsymbol{\theta}^T \mathbf{x}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}}} = \frac{e^{-\boldsymbol{\theta}^T \mathbf{x}}}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}} \tag{8}$$

Note that there is no random error ϵ included the logistic regression model (7), as in the linear regression model (6), due to the randomness in classification that is statistically modeled by the conditional class probability $p(y|\mathbf{x})$, instead of an additional error.

So, by using the logistic function, the linear regression model - a model for solving regression problems - is modified into the logistic regression model - a model for classification problems. In order to use the logistic regression model for predicting class probabilities, or actual class predictions, we need to learn the parameters $\boldsymbol{\theta}$ from the training data $T = \{\mathbf{x_i}, y_i\}_{i=1}^{n}$. The parameters $\boldsymbol{\theta}$ can be learned by using the *likelihood function* $\ell(\boldsymbol{\theta})$, considering the (natural) logarithm of the likelihood function for numerical reasons,

$$\ln \ell(\boldsymbol{\theta}) = \sum_{i=1}^{n} \ln p(y_i \mid \mathbf{x_i}; \boldsymbol{\theta}) \tag{9}$$

to find the *maximum likelihood estimate* $\hat{\boldsymbol{\theta}}$ for $\boldsymbol{\theta}$

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, p(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{n} \ln p(y_i \mid \mathbf{x_i}; \boldsymbol{\theta}) \tag{10}$$

Solving the problem, means estimating, or learning, the parameters $\hat{\boldsymbol{\theta}}$ that maximizes the likelihood that an input $\mathbf{x_i}$ belongs to class y_i. Generally, this maximization problem is turned into the dual minimization problem $\underset{\boldsymbol{\theta}}{\operatorname{argmin}}\ -\sum_{i=1}^{n} \ln p(y_i \mid \mathbf{x_i}; \boldsymbol{\theta})$, together with the statistical average, giving us $J(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{n} \ln p(y_i \mid \mathbf{x_i}; \boldsymbol{\theta})$, commonly known as the *cost function*. From the logistic regression model given in (7) and (8), together with the cost function $J(\boldsymbol{\theta})$, this gives us

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \begin{cases} -\ln\ f(\mathbf{x_i}; \boldsymbol{\theta}) & \textit{if}\ \ y_i = 1 \\ -\ln\ (1 - f(\mathbf{x_i}; \boldsymbol{\theta})) & \textit{if}\ \ y_i = -1 \end{cases} \tag{11}$$

which can be written in a more compact way, using the binary class formulation of $y_i = \{-1, 1\}$, since the logistic regression model in (7) and (8) gives us

$$\begin{cases} f(\mathbf{x}; \boldsymbol{\theta}) = \frac{e^{\boldsymbol{\theta}^T\mathbf{x}}}{1+e^{\boldsymbol{\theta}^T\mathbf{x}}} = \frac{e^{y_i\boldsymbol{\theta}^T\mathbf{x}}}{1+e^{y_i\boldsymbol{\theta}^T\mathbf{x}}} & \textit{if}\ \ y_i = 1 \\ 1 - f(\mathbf{x}; \boldsymbol{\theta}) = \frac{e^{-\boldsymbol{\theta}^T\mathbf{x}}}{1+e^{-\boldsymbol{\theta}^T\mathbf{x}}} = \frac{e^{y_i\boldsymbol{\theta}^T\mathbf{x}}}{1+e^{y_i\boldsymbol{\theta}^T\mathbf{x}}} & \textit{if}\ \ y_i = -1 \end{cases} \tag{12}$$

the same expression on both cases, and thus we can formulate the cost function (11) as

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} -\ln \frac{e^{y_i\boldsymbol{\theta}^T\mathbf{x}}}{1+e^{y_i\boldsymbol{\theta}^T\mathbf{x}}} = \frac{1}{n}\sum_{i=1}^{n} -\ln \frac{1}{1+e^{-y_i\boldsymbol{\theta}^T\mathbf{x}}} = \frac{1}{n}\sum_{i=1}^{n} \ln(1+e^{-y_i\boldsymbol{\theta}^T\mathbf{x}}) \tag{13}$$

Hence, learning a logistic regression model, means solving

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\ \frac{1}{n}\sum_{i=1}^{n} \ln(1+e^{-y_i\boldsymbol{\theta}^T\mathbf{x}}) \tag{14}$$

and this is equivalent to finding the maximum likelihood estimate $\hat{\boldsymbol{\theta}}$ as written in (10). This is a nonlinear optimization problem that needs to be solved numerically, as there is no direct solution for finding $\hat{\boldsymbol{\theta}}$.

Once we have learned the parameters $\hat{\boldsymbol{\theta}}$, we can use the logistic regression model,(7) and (8), to predict class probabilities, given some test input data $\mathbf{x}_*$. The next step is to turn these predicted class probabilities into actual class predictions $\hat{\mathbf{y}}(\mathbf{x}_*) = \mathbf{1}$ or $\hat{\mathbf{y}}(\mathbf{x}_*) = -\mathbf{1}$. The most common approach is to let $\hat{\mathbf{y}}(\mathbf{x}_*)$ be the most probable class by

$$\hat{\mathbf{y}}(\mathbf{x}_*) = \begin{cases} 1 & \textit{if}\ \ f(\mathbf{x}; \hat{\boldsymbol{\theta}}) > r, \\ -1 & otherwise \end{cases} \tag{15}$$

with decision threshold $r = 0.5$, but in general the threshold is chosen by the user $0 \leq r \leq 1$. The input points where the prediction changes from one class to the other, that is either of the two classes

$(y = 1 \text{ or} -1)$ is equally probable, is the decision boundary. This corresponds to solving the equation

$$f(\mathbf{x};\boldsymbol{\theta}) = 1 - f(\mathbf{x};\boldsymbol{\theta}) \tag{16}$$

and for logistic regression this means

$$\frac{e^{\boldsymbol{\theta}^T\mathbf{x}}}{1+e^{\boldsymbol{\theta}^T\mathbf{x}}} = 1 - \frac{e^{\boldsymbol{\theta}^T\mathbf{x}}}{1+e^{\boldsymbol{\theta}^T\mathbf{x}}} \;\Leftrightarrow\; e^{\boldsymbol{\theta}^T\mathbf{x}} = 1 \;\Leftrightarrow\; \boldsymbol{\theta}^T\mathbf{x} = 0 \tag{17}$$

The equation $\boldsymbol{\theta}^T\mathbf{x} = 0$ indicates a (linear) hyperplane, which means that the decision boundary for logistic regression is always linear.

2.4 Lasso and Ridge Regularization

To improve a regression model and avoid overfitting we can use regularization. The key concept behind regularization is that we want to make sure that our parameters $\hat{\boldsymbol{\theta}}$ as small as possible. This is done to ensure that the model complexity won't be too high and can be done in many ways. In Ridge, or L^2 Regularization we alter the cost function by adding a penalty, defined as the sum of the square of the θ terms. They are then scaled by a penalty term, c (often called λ). The size of c can vary, but as it approaches ∞ θ will approach zero. In LASSO, or L^1 Regularization, the penalty is defined as the absolute value of the θ term. The difference between these methods is that in Ridge Regularization the model will typically push its θ parameters towards small non-zero values, whereas in LASSO Regularization the model will set θ values to zero, and therefore can act as a feature selection model. [1]

2.4.1 Evaluation

When plotting the validation errors from cross validation (e.i misclassification errors) in a box plot, Ridge Regularization out-performed both LASSO, as well as logistic regression without regularization, for our classification problem. Next step was to choose an appropriate value for the penalty term c. The was done by testing a myriad of values between 1 and 2000 and comparing the validation errors between these different models. Using cross validation ($k = 10$), the validation errors were calculated and plotted in a box diagram. We were looking for a value with an error as small as possible, but also with a small error variance too. Of the tested values of c, many produced similar mean errors, but $c = 1000$ had the "smallest box", i.e. the lowest error variance. c was therefore set to 1000.

3 Our Choice of Method to Proceed With

After testing the three aforementioned methods, we decided to proceed with the Logistic Regression method, and try to tune it. This was done since the model performed best in the cross validation testing.

4 Conclusions

When using the logistic regression method with Ridge regularization, $c = 1000$, we managed to achieve a validation error of between 0.08 and 0.18.

5 The Feature Importance Task

Unfortunately, we have not yet succeeded in evaluating the importance of the different features of the data. So far, we have examine the parameters $\hat{\theta}$ of the logistic regression model with regularization, both of the Ridge and LASSO regression, since the values of these parameters indicate which input variables $\mathbf{x}$ that are most important (i.e those that are most predictive). The input variables with the highest valued parameters, that is the ones hat indicate importance, where especially *Number of male actors*, *Number of female actors*, followed by *Mean Age Male*. However, this is no intuitive to us, and we will go on and examine the feature importance by the following actions; choosing a combination of all input variables, including non-linear transformation of these, and measuring the AIC (Akaike Information Criterion), which will indicate the best fit.

References

[1] Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, and Thomas B. Schön. *Supervised Machine Learning*, 2020.
https://smlbook.org

[2] C.R. Rao, Venu Govindaraju. *Handbook of Statistics, Machinge Learning: Theory and Applications*, 2013

6 Appendix

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import sklearn.preprocessing as skl_pre
import sklearn.linear_model as skl_lm
import sklearn.discriminant_analysis as skl_da
import sklearn.neighbors as skl_nb
import sklearn.model_selection as skl_ms

import os

#from IPython.display import set_matplotlib_formats
#set_matplotlib_formats('png')
from IPython.core.pylabtools import figsize
figsize(10, 6) # Width and hight
#plt.style.use('seaborn-white')


# In[64]:


# Set working directory and read in the csv file
os.chdir("D:\Skola\System i teknik och samh lle\Statistisk Maskininl rning\Project")
cwd = os.getcwd()
print("The current working directory is: ", cwd)

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


# In[65]:


#Make sure we can access the file and what it contains
trainData.info()
trainData.describe()
print(trainData)
pd.plotting.scatter_matrix(trainData)



# In[66]:


print(f"auto.shape: {trainData.shape}") #No. of rows, No. of columns

#Split the data randomly into a training test and a test set of approximately same size
#Set seed to get reproducible results
np.random.seed(1)

trainI = np.random.choice(trainData.shape[0], size = 500, replace = False)
trainIndex = trainData.index.isin(trainI)
train = trainData.iloc[trainIndex]
test = trainData.iloc[~trainIndex]

#Size of Data ---> 1039


# In[67]:

```

```
#Set up train and test data

X_train = train[['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']]
Y_train = train['Lead']
X_test = test[['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']]
Y_test = test['Lead']


# In[68]:


# set up the KNN-solver for comparison

model = skl_nb.KNeighborsClassifier(n_neighbors = 1)
model.fit(X_train, Y_train)

prediction = model.predict(X_test)
print('Confusion Matrix :\n')
print(pd.crosstab(prediction, Y_test), '\n')
print(f'Accuracy: {np.mean(prediction == Y_test):.3f}')


# In[69]:


#Predict using LDA

model = skl_da.LinearDiscriminantAnalysis()
model.fit(X_train, Y_train)

predict_prob = model.predict_proba(X_test)
print('The class order in the model:')
print(model.classes_)

print('Examples of predicted probabilities for the above classes:')
with np.printoptions(suppress = True, precision = 4): #Supress scientific notation, e.g. 1.0e-2.
    print(predict_prob[0:5], '\n \n Actual class:\n', trainData.loc[0:5,'Lead']) #Inspect the first five predictions



# In[70]:


prediction = np.empty(len(X_test), dtype = object)
prediction = np.where(predict_prob[:,0]>=0.5, 'Female', 'Male')
print('First five predictions:')
print(prediction[0:5], '\n') #Inspect the first five predictions after Labeling
print('First five actual data:')
print(trainData.loc[0:5, 'Lead'])

#Confusion matrixt
```

```
print(' \n Confusion matrix:')
print(pd.crosstab(prediction, Y_test,),'\n')

#Accuracy
print(f'Accuracy: {np.mean(prediction == Y_test):.3f}')


# In[71]:


#Code for comparison of the different solvers for LDA

X_train2 = trainData[['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']]
Y_train2 = trainData['Lead']

# set up the model
model = skl_da.LinearDiscriminantAnalysis()
# setup the evaluation method
cv = skl_ms.RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# set up grid
grid = dict()
grid['solver'] = ['svd', 'lsqr', 'eigen']
# set up search
search = skl_ms.GridSearchCV(model, grid, scoring='accuracy', cv=cv, n_jobs=-1)
# execute the search
results = search.fit(X_train2, Y_train2)
# summarize and evaluate which solver works best, we continue with the best one
print('Mean Accuracy: %.3f' % results.best_score_)
print('Config: %s' % results.best_params_)
print('best param: %s' %results.best_params_)


# In[72]:


#Check how shrinkage affects the methods

X_train3 = trainData[['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']]
Y_train3 = trainData['Lead']


# set up the model
model = skl_da.LinearDiscriminantAnalysis(solver='eigen')
# set up the evaluation method
cv = skl_ms.RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# set up the grid
grid = dict()
grid['shrinkage'] = np.arange(0, 1, 0.01)
# set up the search
search = skl_ms.GridSearchCV(model, grid, scoring='accuracy', cv=cv, n_jobs=-1)
```

```python
# execute the search
results = search.fit(X_train3, Y_train3)
# summarize the affects of the shrinkage
print('Mean Accuracy: %.3f' % results.best_score_)
print('Config: %s' % results.best_params_)


# In[73]:


#cross-validation comparisons of the different methods

X_train4 = trainData[['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']]
Y_train4 = trainData['Lead']

n_fold = 10

models = []
models.append(skl_da.LinearDiscriminantAnalysis()) #Set up LDA
models.append(skl_nb.KNeighborsClassifier(n_neighbors = 3)) #Set up K-nn with k = 2
models.append(skl_lm.LogisticRegression(solver='liblinear')) #Set up Logistic Regression

missclassification = np.zeros((n_fold, len(models)))
cv = skl_ms.KFold(n_splits=n_fold, random_state = 1, shuffle = True)

for i, (train_index, val_index) in enumerate (cv.split(X_train4)):
    X_trainfold, X_val = X_train4.iloc[train_index], X_train4.iloc[val_index]
    Y_trainfold, Y_val = Y_train4.iloc[train_index], Y_train4.iloc[val_index]

    for m in range(np.shape(models)[0]): #try different models
        model = models[m]
        model.fit(X_trainfold, Y_trainfold)
        prediction = model.predict(X_val)
        missclassification[i,m] = np.mean(prediction != Y_val)
        #print(missclassification[i,m])

        modelLDA = models[0]
        X_lda = modelLDA.fit_transform(X_trainfold, Y_trainfold)
        #print(modelLDA.explained_variance_ratio_)


print(missclassification)
plt.boxplot(missclassification)
plt.title('cross validation error for different methods')
plt.xticks(np.arange(4)+1, ('LDA', 'K-nn', 'Logistic Regression'))
plt.ylabel('validation error')
plt.show()

#model.explained_variance_ratio_
```

```python
#!/usr/bin/env python
# coding: utf-8

# Load info from the csv file train.csv, located at 'C:\Users\vikin\Documents\STS\Statistisk maskininl rning\projects\1\train.csv'. The file consists of 13 datapoints and then the classifier.
```

```
#
# Implement a KNN algorithm and use the training data as a basis for it.
#
#
# Number of males:
# 785
#
# Number of females:
# 254
#
# -> predicting only males will result in an accuracy of 0.7555341674687199

# In[11]:


import matplotlib.pyplot as plt
import numpy as np
import csv

# KNN related functions

def eucledian_distance(p1, p2):
    '''returns the eucledian distance between the two tuples p1 and p2'''
    d = 0
    for i in range(len(p1)):
        d += (p1[i] - p2[i])**2
    return d**0.5

def knn_classify_me(k, distance_and_class):
    '''returns the most common class from k points with shortest distance
        in the unordered list of tuples (distance, class) in distance_and_class'''
    distance_and_class.sort(key=lambda x: x[0])

    class_dict = {}

    for i in range(k):
        temp_class = distance_and_class[i][1]
        if temp_class in class_dict:
            class_dict[temp_class] += 1
        else:
            class_dict[temp_class] = 1

    return max(class_dict, key = class_dict.get)

def knn_return_distance_and_class_vector(p, training_values, training_classes):
    '''returns a list of tuples (distance, class) distance_and_class
        for every instance in the two training lists and their distance to the point p'''
    distance_and_class = []

    for i in range(len(training_values)):
        current_point = training_values[i]
        distance = eucledian_distance(p, current_point)
        current_class = training_classes[i]
        distance_and_class.append((distance, current_class))

    return distance_and_class

def knn_p(k, point, training_values, training_classes):
```

```
    '''returns the class of point p accoring to the k nearest neighbors
        in trainig_values and their classes in training_classes'''
    distance_and_class = knn_return_distance_and_class_vector(point, training_values, training_classes)
    return knn_classify_me(k, distance_and_class)

def knn(k, testing_values, training_values, training_classes):
    '''returns a list of classes our testing_values get based on
        our training_values and training_classes'''
    classes = []

    for point in testing_values:
        current_class = knn_p(k, point, training_values, training_classes)
        classes.append(current_class)
    return classes


# In[72]:


# general data related functions


def feasible(k, testing_values, training_values, training_classes):
    '''ensures that the length of all lists and tuples match'''
    if len(training_values) != len(training_classes):
        print("training data doesn't match")
        return False

    if len(training_values[0]) != len(testing_values[0]):
        print("values don't match")
        return False
    if k > len(training_values):
        print("k is too big")
        return False
    return True

def results(predicted, actual):
    '''returns accuracy and prints all sorts of stuff'''
    predicted_males_correct = 0
    predicted_females_correct = 0

    predicted_males_false = 0
    predicted_females_false = 0

    for i in range(len(predicted)):
        if predicted[i] == "Male":
            if actual[i] == "Male":
                predicted_males_correct += 1
            else:
                predicted_males_false += 1
        else:
            if actual[i] == "Female":
                predicted_females_correct += 1
            else:
                predicted_females_false += 1
    print('total predicted males {}'.format(predicted_males_correct + predicted_males_false))
    print('total predicted females {}'.format(predicted_females_correct + predicted_females_false))
    print('predicted_males_correct / total males {}'.format(predicted_males_correct/(predicted_males_correct+predicted_females_false)))
```

```
    print('predicted_females_correct / total females {}'.format(predicted_females_correct/(predicted_females_correct+predicted_males_false)))
    return (predicted_males_correct + predicted_females_correct) / len(predicted)

def normalize(eg_array):
    '''subtracts the mean and divides by the standard deviation'''
    return (eg_array - eg_array.mean(axis=0)) / eg_array.std(axis=0)

def split_into_k(arr, k = 10):
    '''returns a list of k subsections of arr'''
    part = len(arr) // k
    list_of_matrices = [0] * k

    for i in range(k):
        if i == k - 1:
            list_of_matrices[i] = arr[part * i : ]
        else:
            list_of_matrices[i] = arr[part * i : part * (i + 1)]
    return list_of_matrices


def combine_results(results_arrays):
    '''returns a list of the most common results for each index of the results_arrays'''
    results = []
    for i in range(len(results_arrays[0])):
        temp = {}
        for j in range(len(results_arrays)):
            if results_arrays[j][i] in temp:
                temp[results_arrays[j][i]] = temp[results_arrays[j][i]] + 1
            else:
                temp[results_arrays[j][i]] = 1
        results.append(max(temp, key = temp.get))
    return results


def make_predicted_csv(data, csv_link = "predictions.csv"):
    '''takes the list data and pirnts its content on csv_link
        expects a list of either ints or female/male'''
    with open(csv_link, 'w', newline = '') as f:
        wr = csv.writer(f)
        for prediction in data:
            if type(prediction) == int:
                wr.writerow(str(prediction))
            else:
                pred = prediction.lower()
                if pred == 'female':
                    wr.writerow(str(1))
                else:
                    wr.writerow(str(0))
        f.close()
        return


# In[22]:


# testing of our algorithm
# only really works in 2D
'''

```

```
training_values = np.array([(-1, -10), (-3, -20), (-4, -40), (-1, -10), (-5, -10), (10, 100), (11, 110), (10, 90), (9, 80), (20, 150)])
training_classes = np.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
testing_values = np.array([(-5, -5), (-2, -1), (4, 4), (9, 9)])
k = 3


testing_classes = np.array(knn(k, testing_values, training_values, training_classes))
print(testing_classes)


plt.scatter(training_values[:, 0], training_values[:, 1], c = training_classes, label = "training")
plt.scatter(testing_values[:, 0], testing_values[:, 1], c = testing_classes, marker = "v", label = "testing")
plt.legend()
plt.show()


print(training_values)
training_values = normalize(training_values)
print(training_values)'''


# In[13]:


# we declare our data
training_values = []
training_classes = []
testing_values = []
testing_actual = []

total_lines = 1039

number_of_training = 780


link = r"C:\Users\vikin\Documents\STS\Statistisk maskininl rning\projects\1\train.csv"
short = r"C:\Users\vikin\Documents\STS\Statistisk maskininl rning\projects\1\short.csv"

# we fill our data
file = open(link)
csv_reader = csv.reader(file)
next(csv_reader) # skip the header

i = 0
for row in csv_reader:
    if i < number_of_training:
        training_values.append(tuple(row[0 : -1]))
        training_classes.append(row[-1])
    else:
        testing_values.append(tuple(row[0 : -1]))
        testing_actual.append(row[-1])
    i += 1

# normalization of the data
training_values = np.array(training_values).astype(np.float)
training_values = normalize(training_values)

testing_values = np.array(testing_values).astype(np.float)
testing_values = normalize(testing_values)
```

```
# # Finding the Best K Value
# We'll test K values in [1, 41] and pick a K value based on accuracy.

# In[29]:


for k in range(1, 42):
    print("K:", k)

    testing_classes = np.array(knn(k, testing_values, training_values,
     training_classes))
    print(results(testing_classes, testing_actual))


# # About the Data
# To test only certian attributes:
#     modify the attributes list to contain the indices you want to
    test and move it like below
#
#     altered_testing = testing_values[:, attributes]
#     altered_training = training_values[:, attributes]
#
#
# ## Key:
# * Number words female: 0,
# * Total words: 1,
# * Number of words lead: 2,
# * Difference in words lead and co-lead: 3,
# * Number of male actors: 4,
# * Year: 5,
# * Number of female actors: 6,
# * Number words male: 7,
# * Gross: 8,
# * Mean Age Male: 9,
# * Mean Age Female: 10,
# * Age Lead: 11,
# * Age Co-Lead: 12

# In[61]:


# let's run this shit
#for i in range(13):
#print(i)
attributes = [0,7] #[0, 5, 7, 8]

k = 3
altered_testing = testing_values [:, attributes]
altered_training = training_values [:, attributes]

testing_classes = np.array(knn(k, altered_testing, altered_training,
    training_classes))
print(results(testing_classes, testing_actual))


# # Sampling Without Replacement
#
# We split our matrix into k subsets and predict based on that. We the
     choose the most common prediction.

# In[35]:


```

```
k_split = 10
k_neighbors = 3

#print(len(training_values))
#print(training_values)

#print(len(training_classes))
#print(training_classes)

split_training_values = split_into_k(training_values, k_split)
#print(split_training_values)
split_training_classes = split_into_k(training_classes, k_split)
#print(split_training_classes)

sampling_results = [0] * k_split

#print(split_training_values[i])
#print(split_training_classes[i])

for i in range(k_split):
    sampling_results[i] = knn(k_neighbors, altered_testing,
    split_training_values[i], split_training_classes[i])


#print(cross_validaton_results)
sampling_predictions = combine_results(sampling_results)


print(results(sampling_predictions, testing_actual))


# # Making a CSV File
#
#

# In[71]:


import os
print(os.getcwd())

d = [1,0,1,0,1,0,1,0,2,3,1,1]
#d = ["male", "female", "MAle", "Female"]


make_predicted_csv(d)


# In[64]:


# 2D plot



blue_class = np.array([(1,1),(4,10),(5,6),(10, 3),(2,4)])
red_class = np.array([(5,1),(6,4),(2,6),(3, 10),(6.5,5)])
yellow_class = np.array([(5, 5)])



fig, ax = plt.subplots()
#plt.xticks([])
#plt.yticks([])

```

```
ax.set(xlim=(0, 11), ylim = (0, 11))
ax.set_aspect('equal')
ax.axes.xaxis.set_visible(False)
ax.axes.yaxis.set_visible(False)

ax.scatter(blue_class[:, 0], blue_class[:, 1], c = "b", marker = "v",
    label = "blue class")
ax.scatter(red_class[:, 0], red_class[:, 1], c = "r", marker = "v",
    label = "red class")
ax.scatter(yellow_class[:, 0], yellow_class[:, 1], c = "y", label = "
    unlabeled")
#ax.scatter([x^2 + y^2 = 1])
cir1 = plt.Circle((5, 5), 1, color='b',fill=False)
#ax.add_artist(cir1)
cir2 = plt.Circle((5, 5), 1.5, color='r',fill=False)
ax.add_artist(cir2)
plt.title('Classification with k = 3')



ax.legend()

fig.savefig('plot2.png')

#ax.show()

'''

training_values = np.array([(-1, -10), (-3, -20), (-4, -40), (-1, -10)
    , (-5, -10), (10, 100), (11, 110), (10, 90), (9, 80), (20, 150)])
training_classes = np.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
testing_values = np.array([(-5, -5), (-2, -1), (4, 4), (9, 9)])
k = 3


testing_classes = np.array(knn(k, testing_values, training_values,
    training_classes))
print(testing_classes)


plt.scatter(training_values[:, 0], training_values[:, 1], c =
    training_classes, label = "training")
plt.scatter(testing_values[:, 0], testing_values[:, 1], c =
    testing_classes, marker = "v", label = "testing")
plt.legend()
plt.show()


print(training_values)
training_values = normalize(training_values)
print(training_values)'''
```

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import csv

#scikit
import sklearn.preprocessing as skl_pre
```

```
import sklearn.linear_model as skl_lm
import sklearn.model_selection as skl_ms

from IPython.core.pylabtools import figsize
figsize(10, 6)


# In[ ]:


# Read data
directory = 'directory of dataset'
traindata = pd.read_csv(directory, na_values='?', dtype={'Lead': str}).dropna().reset_index()
#traindata.info()

# Preprocessing data
np.random.seed(1)

# Splitting up dataset in training and test data: 538 training, 500 test
trainIndex = np.random.choice(traindata.shape[0], size=538, replace=False)

trainIndexBool = traindata.index.isin(trainIndex) # generates a np array with True/False if corresponding index in in trainIndex
train = traindata.iloc[trainIndexBool] # training set of 538
test = traindata.iloc[~trainIndexBool] # test set of 500 (1038-538)

# Feature selection
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', 'Number words male', 'Gross', 'Mean Age Male', 'Mean Age Female', 'Age Lead', 'Age Co-Lead']

# Training set
X_train = train[features] # input
Y_train = train['Lead'] # output

# Test set
X_test = test[features]
Y_test = test['Lead']


# In[ ]:


# Learn model
log_reg_model_ridge = skl_lm.LogisticRegression(solver='liblinear', penalty='l2', C=1000) # Ridge Regression
log_reg_model_ridge.fit(X_train,Y_train)
#print('Model coefficients, Ridge Logistic Regression: \n', log_reg_model_ridge.coef_, '\n')

# Prediction
r = 0.5
prediction = np.where(prob[:,0] >= r, 'Female','Male')

# Confusion matrix
print('Confusion matrix:\n', pd.crosstab(prediction, Y_test), '\n')

# Accuracy
print(f'Accuracy: {np.mean(prediction == Y_test):.3f}', '\n') # rounds up to 3 decimals
```

```


# In[ ]:


# Compare Logistic regression models with/without regularization
log_reg_model_lasso = skl_lm.LogisticRegression(solver='liblinear',
    penalty='l1', C=1000) # Lasso Regression
log_reg_model_ridge = skl_lm.LogisticRegression(solver='liblinear',
    penalty='l2', C=1000) # Ridge Regression
log_reg_model_none = skl_lm.LogisticRegression(solver='lbfgs', penalty
    ='none', max_iter=2500) # no regularization

models = []
models.append(log_reg_model_lasso)
models.append(log_reg_model_ridge)
models.append(log_reg_model_none)

# learn model from training data
for model in models:
     model.fit(X_train,Y_train)

print('Model coefficients, Lasso Logistic Regression: \n',
    log_reg_model_lasso.coef_, '\n')
print('Model coefficients, Ridge Logistic Regression: \n',
    log_reg_model_ridge.coef_, '\n')
print('Model coefficients, Logistic Regression: \n',
    log_reg_model_ridge.coef_, '\n')


# In[ ]:


pred_prob_lasso = log_reg_model_lasso.predict_proba(X_test)
pred_prob_ridge = log_reg_model_ridge.predict_proba(X_test)
pred_prob_none = log_reg_model_none.predict_proba(X_test)

models_prob = {}
models_prob[log_reg_model_lasso] = pred_prob_lasso
models_prob[log_reg_model_ridge] = pred_prob_ridge
models_prob[log_reg_model_none] = pred_prob_none

# Descision
r = 0.5

for model, prob in models_prob.items():
        print('Class order in the model: ', model.classes_)

        # Print first 5 predictions
        print('Examples of predicted probabilities for the above
    classes: \n', item[1][0:5], '\n')
        # print(f'Average probability for Female lead is {100*(sum(
    prob[:,0])/len(prob)):.0f} %')
        # print(f'Average probability for Male lead is {100*(sum(prob
    [:,1])/len(prob)):.0f} %', '\n')

        # Prediction
        prediction = np.where(prob[:,0] >= r, 'Female','Male')

        # Confusion matrix
        print('Confusion matrix:\n', pd.crosstab(prediction, Y_test),
    '\n')

        # Accuracy
```

```
        print(f'Accuracy: {np.mean(prediction == Y_test):.3f}', '\n') # rounds up to 3 decimals
        # print('Accuracy: ', np.mean(prediction == Y_test))

        # Precision

        # Recall

        # F1


# In[ ]:


# decide positive resp negative class
pos_class = 'Male'
neg_class = 'Female'

# count num of positive resp negative samples in test
P = np.sum(Y_test == pos_class) # num of positive samples in test
N = np.sum(Y_test == neg_class) # num of negative samples in test

# find indices with male/positive class in data
pos_class_index = np.argwhere(log_reg_model.classes_ == pos_class).squeeze()

# lists to append TP / FN rate for different r
true_pos_rate = []
false_pos_rate = []

# r [0,1], dvs r= 0.00, 0.01,...,0.1,0.11,...,0.99,1.0
threshold = np.linspace(0.00, 1, 101) # start=0, stop=1, 101 num evenly spaced samples

for model, prob in models_prob.items():
    for r in threshold:
        prediction = np.where(prob[:,pos_class_index] > r, pos_class, neg_class)
        TP = np.sum((prediction == pos_class) & (Y_test == pos_class)) #True Positive
        true_pos_rate.append(TP/P)
        FP = np.sum((prediction == pos_class)&(Y_test == neg_class)) # False Positive
        false_pos_rate.append(FP/N)

    plt.plot(false_pos_rate, true_pos_rate);
    for i in [0,1,10,50,98,100]:
        plt.text(false_pos_rate[i], true_pos_rate[i], f'r={threshold[i]:.2f}')
    plt.xlim([0, 1])
    plt.ylim([0, 1.1])
    plt.xlabel('False positive rate')
    plt.ylabel('True positive rate');

# print('false_pos_rate',false_pos_rate)
# print('true_pos_rate', true_pos_rate)


# In[ ]:


# Cross validation

n_fold = 10

```

```
models = []
models.append(skl_lm.LogisticRegression(solver='liblinear', penalty='l1', C=1000)) # Lasso Regression
models.append(skl_lm.LogisticRegression(solver='liblinear', penalty='l2', C=1000)) # Ridge Regression
models.append(skl_lm.LogisticRegression(solver='lbfgs', penalty='none', max_iter=2500)) # no regularization

missclassification = np.zeros((n_fold, len(models)))
cv = skl_ms.KFold(n_splits=n_fold, random_state = 1, shuffle = True)

for i, (train_index, val_index) in enumerate (cv.split(X_train)):
    X_trainfold, X_val = X_train.iloc[train_index], X_train.iloc[val_index]
    Y_trainfold, Y_val = Y_train.iloc[train_index], Y_train.iloc[val_index]

    for m in range(np.shape(models)[0]): #try different models
        model = models[m]
        model.fit(X_trainfold, Y_trainfold)
        prediction = model.predict(X_val)
        missclassification[i,m] = np.mean(prediction != Y_val)

print(missclassification)
plt.boxplot(missclassification)
plt.title('cross validation error for different methods')
plt.xticks(np.arange(4)+1, ('Logistic with Lasso', 'Logistic with Ridge', 'Logistic no regularization'))
plt.ylabel('validation error')
plt.show()


# In[ ]:


# Cross validation, changing pentalty parameter C (or lambda)

n_fold = 10
C = [1,10,50,100,400,999,1000,2000,5000]

missclassification = np.zeros((n_fold, len(C)))
cv = skl_ms.KFold(n_splits=n_fold, random_state = 1, shuffle = True)

for i, (train_index, val_index) in enumerate (cv.split(X_train)):
    X_trainfold, X_val = X_train.iloc[train_index], X_train.iloc[val_index]
    Y_trainfold, Y_val = Y_train.iloc[train_index], Y_train.iloc[val_index]

    for j in range(len(C)): #try different models/C
        model = skl_lm.LogisticRegression(solver='liblinear', penalty='l2', C=C[j]) # Ridge Regression
        model.fit(X_trainfold, Y_trainfold)
        prediction = model.predict(X_val)
        missclassification[i,j] = np.mean(prediction != Y_val)

#print(missclassification,'\n')
plt.boxplot(missclassification)
plt.title('cross validation error for different methods')
plt.xticks(np.arange(9)+1, ('c=1', 'c=10', 'c=50', 'c=100', 'c=400', 'c=999','c=1000', 'c=1001', 'c=5000'))
plt.ylabel('validation error')
plt.show()
```