Group S181(4 members): Mini Project Report

1 Introduction

Algorithms that predict user preferences is a hot topic both within academic research and industrial productions. In this report, different algorithms for predicting user preference on music will be tested. To do this, a data set of 750 songs and several of their high-level characteristics such as the *acousticness*, *danceability*, *energy*, *instrumentalness* and *valence* have been used. This dataset also includes *label*, which denotes whether the user has marked the song as "liked" or "disliked". The song features, which are extracted from Spotify's API, will be used to train and evaluate four different classifiers. The classifiers are supposed to predict whether or not a person *likes* or *dislikes* a particular song based on the persons previously labeled songs and their characteristics. The performance of the different classifiers will then be evaluated with the purpose of choosing one to *put in production*, i.e. which one to use to classify 200 song with unknown labeling.

2 Description of considered methods

2.1 Logistic regression

Regression is a method that describes the relationship between input variables x and output variables y, and is intended for binary classification problems. The method is used when there are one or more independent variables in the dataset that determines the outcome, which is the case in this project. Furthermore, the dependent variable should only contain data coded as 0 or 1. The method uses an equation as the representation, much like linear regression, and the input values are combined using coefficient values, given by β, to predict the output value. The β-values are estimated from the training data by using the maximum-likelihood estimation. Logistic regression tries to predict the probability that a given input belongs to a certain class. One assumption is that the input space can be separated into different regions for each class by a decision boundary; for example, this boundary could in the three dimensional case be a plane and helps to differentiate probabilities into different classes. To deal with outliers, logistic regression uses the logistic function, a Sigmoid function, that converts log-odds to probability. The graph from the function will look like an S-shaped curve if plotted. It takes any real value and outputs a value of zero or one, and is defined as:

$$f(x) = \frac{1}{1+e^{-x}} \tag{1}$$

2.2 Discriminant analysis: LDA, QDA

Linear (LDA) and quadratic (QDA) discriminant analysis are two parametric models, i.e have a predefined form, which are tuned to fit the training data. The two models the distribution of the predictors separately in each of the response category, and then flip these around into probability estimates for the response classes (given the predictor), by using Bayes' theorem for conditional probabilities. Both LDA and QDA assumes that the predictor variables are drawn from a Gaussian distribution. However, LDA assumes equality of the covariances among the predictor variables while QDA assumes that each class has its own covariance matrix. This makes QDA a more flexible classifier than LDA, and therefore has substantially higher variance.

2.3 *k*-nearest neighbor

k-nearest neighbor is a non-parametric method. Instead of learning parameters from training data, the model considers the *k* closest data points in the training data in terms of Euclidean distance to make its predictions. In classification problems, the prediction is based on the majority class of the *k* nearest neighbors. In case of a tie, some method to break the tie is required. When it comes to selecting a good value for *k* it is important to realize that the training data will always be correctly classified with $k = 1$, but because of this higher flexibility the model might be overfit to the training data and perform worse on test data.

2.4 Boosting

The idea behind boosting is to train several weak models (simple and with high bias) of classification or regression and tying them all together, i.e. an *ensemble* of weak models are combined into a single strong model. While each model in itself might not capture the relation between input and output very well, they do usually capture some aspect of it. Like bagging, boosting is a meta algorithm. That is, an algorithm built on top of other algorithms and it can be built upon almost any classification or regression algorithm. Furthermore, bagging and boosting are both ensemble methods that uses several weak models to come up with a prediction. However, unlike bagging, which makes use of its ensemble members in *parallel*, a boosting ensemble is built *sequentially* where each model takes previous models in the sequence into account and tries to correct any errors made by previous ensemble members. It does so by modifying the training data set in each iteration to put more emphasis on data points for which the ensemble so far has been performing poorly.

One well known boosting algorithm, and also the one used in this report, is called AdaBoost (short for adaptive boosting). This method is usually constructed with shallow classification trees (depth 1, so called "stumps") as its base classifiers. The number of base classifiers is denoted *B* and each of their predictions are not treated equally. Instead, they are weighted with some positive coefficient $\{\alpha^b\}_{b=1}^{B}$ (for the b:th iteration). This coefficient is calculated keeping the previous ensemble members and their errors in mind, thus creating the boosted classifier.

3 Application on test data

All the methods in the previous section was used for binary classification and were implemented and evaluated in Python using the external libraries "numpy", "pandas", "matplotlib" and "sci-kit learn". The following section will present how the methods were applied on the test data.

3.1 Quantitative or qualitative input

All methods showed the same or better results when treating the predictor variable *time signature* as qualitative input, except when using logistic regression where the result was a little better when keeping it as a quantitative variable. The variable *key* could also be treated as qualitative input instead of quantitative input, but was not. This decision was made because *key* would have to be split into 12 different dummy variables, which affected the time complexity of testing methods, and also because tests never showed that the *key* variable had any big impact on music preference. It could also be argued that the *key* inputs are ordered since it refers to a scale, which supports the choice of quantitative treatment. The *mode* variable was said in to be a string in the Spotify API documentation but was already represented as 0 or 1, with no natural order, when examining the training data set. Thus nothing was changed but *mode* was treated as qualitative. The rest of the predictor variables were treated as quantitative input.

3.2 Choice of parameters

How the different classifiers were applied to data was somewhat method specific. What was commonly done on all classifiers except boosting was evaluating which of and how many the different predictor variables to use in that specific method. To find out the most important, as well as the least important, predictor variables, a brute-force algorithm was used. The method tested and evaluated, using cross-validation, every combination of every number of inputs and returned the best result together with the predictor variables that resulted in that value.

All the cross-validations on the different classifiers except boosting was done with a 10-fold splitting of the test data set, within the brute-force algorithm. Thereby computing 10 different method fits and averaging their accuracy. Because computing all different combinations took too long for some methods, only subsets of all possible combinations was tested in those cases. After choosing predictor variables using brute-force, leave-one-out cross-validation was used to get better accuracy of method performance.

3.3 Method specific tuning

The *k*-nearest neighbor method will treat small and large values differently, since it is measuring euclidean distance. To avoid that predictor variables with generally large values having greater impact on the prediction, since this is not necessarily true, the values were normalized. For example *duration*, which typically has high values was rescaled which improved the classifier's predictions. Our brute-force approach proved very slow for *k*-nearest neighbor and some compromises had to be made. First, two choices of predictor variables that showed some promise during initial testing was tested with cross validation for a large range of *k*. *k*'s between 3 and 51 seemed to perform better than larger values, and since scikit-learn's implementation of *k*-nearest neighbor does not handle ties we chose to only use odd values of *k* in that range when looping our brute-force algorithm. We kept track of the algorithm's results for each *k* and noted that some combinations of the predictor variables had a high average score and showed up multiple times in the results. We then proceeded to test the four best combinations across the range of *k*'s with leave-one-out cross-validation to find the definite best performing combination and the corresponding *k*.

In tuning the parameters for *AdaBoost*, looping over different values for number of stumps, or base classifiers (`n_estimators`) and learning rate showed that the best performance was reached with the base classifiers to train set to 180 and learning rate to 0,1. To compare results between different loops we afforded ourselves 50-fold splitting for the cross-validation. While a lower learning rate (1 being the default) results in slower training of the model, it can result, as in this case, in better performance. The lowered learning rate is also compensated for by increasing the amount of base classifiers (50 being the default).

With both logistic regression and LDA/QDA, one way to tune the classification is to alter the value of the threshold. However, doing so with our data set made little sense, since a false positive is no less welcome than a false negative. Instead different transformations of inputs was tried, but this yielded no significant gain.

4 Evaluation

As seen in Table 1, the best performing method is *k*-nearest neighbor. All methods outperformed the naïve classifier which always predicts *like*. Worth noting is that high performance of *k*-nearest neighbor seen in the table was achieved with *k*=3 which could be considered too low for the model to be reliable because of a suspected risk of overfit. When using a higher *k*, namely *k*=7 a slightly lower accuracy of ∼0.837, was acquired. However, it still outperforms all other methods evaluated.

Table 1: Best achieved accuracy for all evaluated methods

Classifier	Accuracy
k-nearest neighbor	∼0.841
LDA	∼0.823
Boosting	∼0.827
Logistic Regression	∼0.811
QDA	∼0.797
Naïve Classifier	∼0.603

When using logistic regression and picking all predictors as input variables in the model, the ROC curve resembled a straight line, indicating that the classifier assigned random guesses, see figure 1. After applying the brute-force method the result showed that the most important predictor variables were acousticness, danceability, energy, instrumentalness, loudness, speechiness, time signature, and valence. By using only the most important predictors as input variables in the model, the ROC curve changed and resembled a more typical classifier, see figure 2. It was not a perfect classifier, but certainly better than the initial scenario where all predictors were used as input variables.

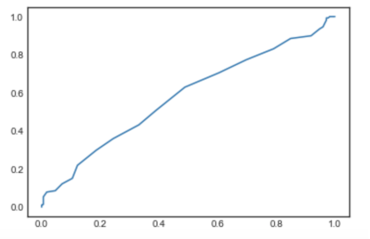
Figure 1: ROC using all variables

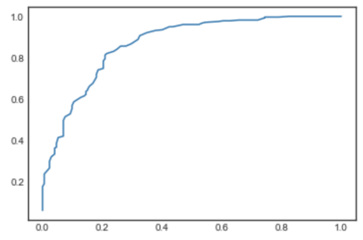
Figure 2: ROC using the most important variables

The same pattern could be identified when limiting input parameters with Linear and Quadratic discriminant analysis, see Table 2.

Table 2: LDA/QDA prediction accuracy

Classifier	Accuracy
LDA with all predictors	∼0.804
LDA with best predictors	∼0.823
QDA with all predictors	∼0.609
QDA with best predictors	∼0.797

5 Production and conclusions

All the different classifiers had better performance than the naïve classifier, i.e always predicting *LIKE*, as shown in Table 1. This justifies tackling the music preference prediction problem with one of the four classifiers.

The fact that both LDA and logistic regression outperforms QDA suggests that the decision boundary can be described fairly well as a linear boundary.

Most of our methods were improved by removing some of the predictor variables using a brute-force algorithm. This, however, was not done for the boosting method. This is because boosting methods per definition is self correcting. Predictor variables with high importance for the performance of the classifier are weighted higher and unimportant variables are weighted lower.

In the end, *k-nearest neighbor* with $k = 3$ and the predictor variables acousticness, danceability, energy, loudness, speechiness and tempo was chosen for production since it gave the best accuracy results. We do not know the performance of this model in terms of correctly classifying the 200 songs with no label as we never submitted this prediction. Why we chose to use *k-nearest neighbor* in production without having submitted the prediction, is because we have evaluated and tuned its performance very extensively with cross validation and we trust that the results we get reflects quite well what we would get if we put it into production.

6 Reflection task (b)

Do we, as machine learning engineers, have a responsibility to inform and educate our client about the risk of obtaining machine learning biases in the solution?

On one hand, machine learning engineers certainly have the responsibility to inform the client about the risk of bias in the algorithm, especially since they need to comply to the IEEE Code of Ethics (IEEE, 2018). It is important that the client is aware of the risks associated with machine learning since they would probably be affected to a high extent if their customers notice a major flaw in the system, which could lead to a potential backlash for the insurance company and the loss of a lot of money. This in turn could affect the reputation and career of the engineer if the insurance company would deem him or her liable for the flawed system. By making the company aware of the risks, they can make a well-informed decision of whether they still want to use the product or not, regardless of the eventual negative consequences; at least then they would have all the necessary facts to base their decision on.

More importantly perhaps, is the potential harm a biased system can do to the company's customers since some customers could face unfair and expensive insurance premiums because of the bias. By informing the insurance company about the risk of some kind of bias, and they decide to use it anyways, the sales agents would probably be advised to not trust the system too much and hopefully re-evaluate any system based decisions about customer insurances that appear to be suspicious. By a more critical monitoring of the system, the company can avoid the risk of unfair insurance deals and increase the chance of happy and loyal customers. After all, by informing the client about the risk of bias and how it could potentially harm the customers, the engineer would act in accordance with the IEEE Code of Ethics which states that members must "hold paramount the safety, health and welfare of the public, to strive to comply with ethical design and sustainable development practices, and to disclose promptly factors that might endanger the public or the environment" (IEEE, 2018). So, based on these arguments the engineer should inform the client about the risk of machine learning bias and get some form of legal contract where it is stated that the client is aware of the risk and maybe even a statement that the client would take full responsibility for any major shortcomings of the system.

On the other hand, it is possible to argue that the machine learning engineer does not have a responsibility to inform the client about the risk of machine learning bias, since they should already be aware of the problems that can arise. As the insurance company hires a machine learning consultant, it is probably safe to assume that they know how the basic principles of machine learning works. And by hiring a machine learning consultant it is their own responsibility to educate themselves about the risks, especially since there are many well-known real cases where problems surrounding machine learning bias have occurred, for example Microsoft's Twitter bot. The insurance business also have a long history of using statistics in determining insurance premiums for different customers, which further indicates that these companies should be aware of the risks of self enforcing biases in statistical machine learning. This is something that O'Neil (2016, p. 119) points out: Bias has been a real problem in the insurance industry even before the introduction of data science, as insurers for a long time clung to the idea that entire groups of people were riskier than others, even if that was not the case.

Our personal opinion is that machine learning engineers have a general responsibility to check that their solutions do not have any machine learning bias to the largest extent possible. This is something they need to do in order to comply to the IEEE Code of Ethics (IEEE, 2018); however, this is a very complex thing to do and in reality we may just need to accept that some bias will always be present. Since machine learning algorithms are developed by individuals that all have different backgrounds, training, perspectives and worldviews, it is basically impossible to avoid bias completely as a result of the human factor. Maybe the most important thing to make sure is that every machine learning engineer has a good intention with their work and that they follow the code of ethics to the best of their ability; in other words, it should be completely forbidden to *intentionally* put personal bias into machine learning algorithms. And considering that bias is more often than not something that exists subconsciously, it is hard to imagine how it would be possible for engineers to detect bias in their own solutions; unfortunately, it is probably a lot easier for the public to detect potential bias in the algorithms after the product has already been put into commercial use.

We should keep in mind, however, that bias is apparent not only in machine learning, but in other types of technology as well. Kleinman (2005, p. 10) notes that technological artifacts embody or are associated with values, and illustrates this view by describing the development of the overpasses that cross Wantagh Park to Long Island, New York and Jones Beach. The overpasses were built in such a way that made it impossible for buses to pass under them, and this design choice represented the designer's social class bias since the low overpasses made the parkway inaccessible for low-income citizens that relied on public transportation. This tells us that bias is present not only in machine learning algorithms, which is why we should not dismiss the technology because of potential bias in the system. As long as we are aware of the risks associated with machine learning and that we work as hard as possible in order to avoid and prevent them, the machine learning bias may be a side effect that we need to accept and live with. To sum up, there will always be some bias but it is just important to be aware of that fact, and act accordingly.

7 References

IEEE (2018). *IEEE Code of Ethics*. `https://www.ieee.org/about/corporate/governance/p7-8.html` (Retrieved 2019-02-15)

Kleinman, Daniel (2005). *Science and Technology in Society: From Biotechnology to the Internet*. New Jersey: Wiley-Blackwell, p. 160.

O'Neil, Cathy (2016). *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. New York: Crown Publishing Group, p. 272.

8 Appendix

8.1 Python Code

```
# All the code used for the miniproject gathered in one file.
# We have not actually run this script because it would take
# a while, so there might be conflicts, but each section should
# run by itself after imports and defining functions.

# We tried to make sure that all the variable references are correct
# in this file, but since we got most of our output results from
# different jupyter-notebook sessions there could be some errors in
# here. We are however confident that the outputs listed in here as
# comments are results from using correct variable references,
# and the names of the variables do reflect our intent.

# Some weird linebreaks were made hastily to make it fit in Latex.

import itertools
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn.preprocessing as skl_pre
import sklearn.neighbors as skl_nb
import sklearn.linear_model as skl_lm
import sklearn.discriminant_analysis as skl_da
from sklearn.ensemble import AdaBoostClassifier
from sklearn import tree


import warnings
warnings.filterwarnings("ignore", category=FutureWarning)

#User warnings about collinear variables, we think
# the brute forcing approach bypasses this problem.
warnings.filterwarnings("ignore", category=UserWarning)

#################
### Functions ###
#################

#Cross validation with shuffling.
#Takes model and fits data. x_columns is an array
# of strings and y_column is a string. n is the desired
# amount of folds. Returns mean score.
def crossValidate(model, data, x_columns, y_column, n):
    fold_size = np.ceil(data.shape[0]/n)
    randomized_indices = np.random.choice(
                                           data.shape[0],
                                           data.shape[0],
                                           replace=False
                                         )
    all_scores = []
    actual_folds = int(np.ceil(data.shape[0] / fold_size))
    for i in range(actual_folds):
        validation_index = np.arange(
                                   i*fold_size,
                                   min(i*fold_size+fold_size,
                                    data.shape[0]),
                                    1
                                    ).astype('int')
        current_fold = randomized_indices[validation_index]
        train = data.iloc[~data.index.isin(current_fold)]
        validation = data.iloc[current_fold]

        x_train = train[x_columns]
        y_train = train[y_column]
        x_validation = validation[x_columns]
        y_validation = validation[y_column]

        model.fit(x_train, y_train)
        all_scores.append(model.score(x_validation, y_validation))
    return np.mean(all_scores)


#Leave one out cross validation, so same as above without the n.
def leaveOneOut(model, data, x_columns, y_column):
    all_scores = []
    for i in range(data.shape[0]):
        validation = data.iloc[i]
        train = data.iloc[~data.index.isin([i])]

        x_train = train[x_columns]
        y_train = train[y_column]
        x_validation = validation[x_columns].values.reshape(1, -1)
        y_validation = validation[y_column].reshape(1, -1)
        model.fit(x_train, y_train)
        all_scores.append(model.score(x_validation, y_validation))
    return np.mean(all_scores)

#Brute forces combinations of input variables from x_columns and
# returns the combnation that resulted in the best score. n and m
# specifies minimum and maximum amount of input variables to consider
# when creating the combinations. Uses crossValidate to compare
# different combinations, cross_folds determines the amount of folds.
def findBestX(model, data, x_columns, y_column, n, m, cross_folds):
    result = (0, [])
    for number_to_choose in range(n, m+1):
        X_combinations = itertools.combinations(
                                                x_columns,
                                                number_to_choose
                                                )
        for X_tup in X_combinations:
            X = [x for x in X_tup]
            score = crossValidate(
                                model,
                                data,
                                X,
                                y_column,
                                cross_folds
                                )
            if score > result[0]:
                result = (score, X)
    return result


###################
```

```
### Reading data ###
####################
data = pd.read_csv('training_data.csv')
#data_nodummies should always be the data without dummy variables
data_nodummies = data
data_dummies = pd.get_dummies(data, columns=['time_signature'])

x_columns = data_dummies.columns.drop(['label'])
x_columns_nodummies = data_nodummies.columns.drop(['label'])
y_column = 'label'

#Also scales all the data points to 0-1 for use in knn
skl_minmax = skl_pre.MinMaxScaler()
data_dummies_scaled = pd.DataFrame(
                              skl_minmax.fit_transform(data_dummies),
                              columns=data_dummies.columns
                              )

#data should be data_dummies, except for KNN
# when it instead should be data_dummies_scaled.
data = data_dummies

#Creating train and test sets for some initial testing.
trainI = np.random.choice(data.shape[0], size = 375, replace = False)
trainIndex = data.index.isin(trainI)
train = data.iloc[trainIndex]
test = data.iloc[~trainIndex]

X_train = train[x_columns]
Y_train = train[y_column]
X_test = test[x_columns]
Y_test = test [y_column]

###########
### KNN ###
###########

#Using scaled data for KNN
data = data_dummies_scaled

#Examines an appropriate range of k's to test for two
# different sets of input variables that proved interesting
# during initial testing. From the plots one can conclude
# that k's between 3 and 51 seems to be the most interesting.
k_range = range(1, 200)

scores = []
X = ['speechiness', 'loudness']
for k in k_range:
    model = skl_nb.KNeighborsClassifier(n_neighbors = k)
    result = crossValidate(model, data, X, y_column, 10)
    scores.append(result)
plt.plot(k_range, scores)

scores = []
X = ['acousticness', 'danceability', 'energy', 'loudness',
    'speechiness', 'tempo']
for k in k_range:
    model = skl_nb.KNeighborsClassifier(n_neighbors = k)
    result = crossValidate(model, data, X, y_column, 10)
    scores.append(result)
plt.plot(k_range, scores)

#Finds out which combinations of the input variables that usually
# gets good results. Only tests for odd k's since sklearn's knn
# doesn't handle ties very well. Also only tests for a maximum of
# 6 input variables since this part takes a while (hours) to run.
first = time.time()

MIN_LENGTH = 1
#Higher MAX_LENGTH would be fun, but it takes too long
MAX_LENGTH = 6
CROSS_FOLDS = 10
K_RANGE = range(3, 52, 2)

win_counter = {}
score_counter = {}
for k in K_RANGE:
    model = skl_nb.KNeighborsClassifier(n_neighbors = k)
    result = findBestX(
                    model,
                    data,
                    x_columns,
                    y_column,
                    MIN_LENGTH,
                    MAX_LENGTH,
                    CROSS_FOLDS
                    )
    win_counter[str(result[1])] = win_counter.get(
                                                str(result[1]),
                                                0
                                                ) + 1
    score_counter[str(result[1])] = score_counter.get(
                                                str(result[1]),
                                                0
                                                ) + result[0]

winner = ''
wins = 0
score = 0
for key in win_counter.keys():
    if win_counter[key] > wins:
        wins = win_counter[key]
        winner = key
        score = score_counter[key] / wins

#Winner is the combination that wins for many different k's,
# not necessarily the one with the highest score overall.
print(winner, score)

then = time.time()

#Saves results in a file for better overview.
with open('results.txt', 'w') as f:
    f.write('time: ' + str(then-first) + ' seconds\n')
    f.write('wins\twinner\t\t\t\t\t\t\t\t\tscore\n')
    for key, value in sorted(
                        win_counter.items(),
                            key = lambda winner: winner[1],
                            reverse=True
                            ):
        line = str(value) + '\t' + str(key) + '\t\t'
        line = line + str(score_counter[key] / value) + '\n'
        f.write(line)
    f.close()

#The results indicates that the following input variables might
# yield the highest scores, this parts tests for the definite best
# combination and choice of k.
X_to_test = []
X_to_test.append(
    ['acousticness', 'danceability', 'energy',
    'loudness', 'speechiness', 'tempo']
    )
X_to_test.append(
    ['acousticness', 'danceability', 'duration',
    'loudness', 'speechiness', 'tempo']
    )
X_to_test.append(
    ['acousticness', 'danceability', 'instrumentalness',
    'loudness', 'speechiness', 'tempo']
    )
X_to_test.append(
    ['acousticness', 'danceability', 'loudness',
    'speechiness', 'tempo', 'time_signature_1']
    )

result = (0, 0, [])
for k in range(3, 51):
    model = skl_nb.KNeighborsClassifier(n_neighbors = k)
    for X in X_to_test:
        score = leaveOneOut(model, data, X, y_column)
        if score > result[0]:
            result = (score, k, X)
print(result)
#sample output:
#(0.8413333333333334, 3,
# ['acousticness', 'danceability', 'energy',
# 'loudness', 'speechiness', 'tempo'])

#Since k=3 leads to suspicions of overfitting, the same test
# was conducted with k > 3.
# Best result should be the one above, but since k = 3 might
# be a bit overfit this test is for comparison.
result = (0, 0, [])
for k in range(5, 16):
    model = skl_nb.KNeighborsClassifier(n_neighbors = k)
    for X in X_to_test:
        score = leaveOneOut(model, data, X, y_column)
        if score > result[0]:
            result = (score, k, X)
print(result)
#sample output:
#(0.8373333333333334, 7,
# ['acousticness', 'danceability', 'energy',
# 'loudness', 'speechiness', 'tempo'])
```

```
###########################
### Logistic Regression ###
###########################

model = skl_lm.LogisticRegression()
data = data_dummies #Switching back to data_dummies

#Testing with a train and test set and fitting the model using
# all input variables.
model.fit(X_train, Y_train)
predict_prob = model.predict_proba(X_test)
prediction = np.empty(len(X_test), dtype = object)
prediction = np.where(predict_prob[:, 0] >= 0.5, 0, 1)

print(pd.crosstab(prediction, Y_test))
#sample output:
#    0    1
#0   69   48
#1   76   182
print(np.mean(prediction == Y_test))
#sample output:
#0.6693333333333333

#Brute force the best choice of input variables
MIN_LENGTH = 1
MAX_LENGTH = 16
CROSS_FOLDS = 10
result = findBestX(
                 model,
                 data,
                 x_columns,
                 y_column,
                 MIN_LENGTH,
                 MAX_LENGTH,
                 CROSS_FOLDS
                 )
print(result)
#sample output:
#(0.8173333333333334,
# ['acousticness', 'danceability', 'instrumentalness',
# 'liveness', 'loudness', 'speechiness', 'time_signature_3',
# 'time_signature_5'])

#Comparing with the train/test results from earlier
X = ['acousticness', 'danceability', 'instrumentalness',
     'liveness', 'loudness', 'speechiness',
     'time_signature_3', 'time_signature_5']
X_train_logreg = train[X]
X_test_logreg = test[X]
model.fit(X_train_logreg, Y_train)
predict_prob = model.predict_proba(X_test_logreg)
prediction = np.empty(len(X_test_logreg), dtype = object)
prediction = np.where(predict_prob[:, 0] >= 0.5, 0, 1)
print(pd.crosstab(prediction, Y_test))
#sample output:
#    0    1
#0   115  41
#1   30   189
print(np.mean(prediction == Y_test))
```

```
#sample output:
#0.8106666666666666

#Estimating how the model would perform when all data is used
# for training
score = leaveOneOut(model, data, X, y_column)
print(score)
#0.8066666666666666

#With Logistic Regression, we also tested if it would
# perform better if time_signature was not treated
# as qualitative, with no dummy variables.
#The following X was found with findBestX for an x_columns
# without dummy variables:
X = ['acousticness', 'danceability', 'energy',
     'instrumentalness', 'loudness', 'speechiness',
     'time_signature', 'valence']
score = leaveOneOut(model, data_nodummies, X, y_column)
print(score)
#0.8106666666666666

#We also created ROC diagrams for a model using all predictor
# variables and for a model
# using the predictor variables in X (just above).
# This was done without the dummy variables
# as Logistic Regression performed better without them.
# The ROC diagrams are in the report.

#This code was used to generate the diagrams, being run twice
# with X_test being "test_nodummies[x_columns]"
# and then "test_nodummies[X]". Unfortunately we lost the exact
# code and we thus left this part commented.

####
#false_positive_rate = []
#true_postive_rate = []
#N = np.sum(Y_test == 0)
#P = np.sum(Y_test == 1)
#threshold = np.linspace(0.01, 0.99, 99)
#model = skl_lm.LogisticRegression(solver = 'liblinear')
#model.fit(X_test, Y_test)
#predict_prob = model.predict_proba(X_test)
#for i in range(len(threshold)):
#    prediction = np.empty(len(X_test), dtype=object)
#    prediction = np.where(predict_prob[:, 0] > threshold[i], 0, 1)
#    FP = np.sum((prediction == 1)&(Y_test == 0))
#    TP = np.sum((prediction == 1)&(Y_test == 1))
#    false_positive_rate.append(FP/N)
#    true_postive_rate.append(TP/P)
#plt.plot(false_positive_rate, true_postive_rate)
####

#################
### LDA & QDA ###
#################

data = data_dummies
LDAmodel = skl_da.LinearDiscriminantAnalysis()
QDAmodel = skl_da.QuadraticDiscriminantAnalysis()
```

```
#Some initial testing with the train / test data set
LDAmodel.fit(X_train, Y_train)
QDAmodel.fit(X_train, Y_train)
print('LDA score: ', LDAmodel.score(X_test,Y_test))
print('QDA score: ', QDAmodel.score(X_test,Y_test))
#sample output:
#LDA score:  0.832
#QDA score:  0.4533333333333333

#Testing LDA with all predictor variables ince LDA was
# unreasonably good this one time
score = leaveOneOut(LDAmodel, data, x_columns, y_column)
print(score)
#0.804

#Finding best predictor variables for LDA from 1 variable to 6
MIN_LENGTH = 1
MAX_LENGTH = 6
CROSS_FOLDS = 10
result = findBestX(
                 LDAmodel,
                 data,
                 x_columns,
                 y_column,
                 MIN_LENGTH,
                 MAX_LENGTH,
                 CROSS_FOLDS
                 )
print(result)
#sample output:
#(0.8280000000000001,
# ['acousticness', 'danceability', 'instrumentalness',
# 'liveness', 'loudness', 'speechiness'])

#leaveOneOut score for the suggested predictor variables above
X = ['acousticness', 'danceability', 'instrumentalness',
    'liveness', 'loudness', 'speechiness']
score = leaveOneOut(LDAmodel, data, X, y_column)
print(score)
#0.8253333333333334

#We also tried finding the best predictor variables in
# a different range for LDA
MIN_LENGTH = 8
MAX_LENGTH = 16
CROSS_FOLDS = 10
result = findBestX(
                 LDAmodel,
                 data,
                 x_columns,
                 y_column,
                 MIN_LENGTH,
                 MAX_LENGTH,
                 CROSS_FOLDS
                 )
print(result)
#sample output:
#(0.8293333333333335,
# ['acousticness', 'duration', 'instrumentalness',
# 'loudness', 'speechiness', 'tempo', 'valence',
# 'time_signature_1', 'time_signature_3', 'time_signature_5'])

X = ['acousticness', 'duration', 'instrumentalness',
     'loudness', 'speechiness', 'tempo', 'valence',
     'time_signature_1', 'time_signature_3', 'time_signature_5']
score = leaveOneOut(LDAmodel, data, X, y_column)
print(score)
#0.8226666666666667

#For QDA we had min and max length set to 8 and 16 for our first run
MIN_LENGTH = 8
MAX_LENGTH = 16
CROSS_FOLDS = 10
result = findBestX(
                  QDAmodel,
                  data,
                  x_columns,
                  y_column,
                  MIN_LENGTH,
                  MAX_LENGTH,
                  CROSS_FOLDS
                  )
#sample output:
#(0.8146666666666667,
# ['acousticness', 'danceability', 'duration',
# 'instrumentalness', 'loudness', 'speechiness',
# 'tempo', 'valence', 'time_signature_5'])

X = ['acousticness', 'danceability', 'duration',
     'instrumentalness', 'loudness', 'speechiness',
     'tempo', 'valence', 'time_signature_5']
score = leaveOneOut(QDAmodel, data, X, y_column)
print(score)
#0.7973333333333333

#An overview of how much our choice of predictor variables
# for LDA and QDA respectively performs
# compared to using all predictor variables
X_l = ['acousticness', 'danceability', 'instrumentalness',
     'liveness', 'loudness', 'speechiness']
X_q = ['acousticness', 'danceability', 'duration',
     'instrumentalness', 'loudness', 'speechiness',
     'tempo', 'valence', 'time_signature_5']
y_column = 'label'


print(
    'LDA with good:',
    leaveOneOut(LDAmodel, data_dummies, X_l, y_column)
    )
print(
    'LDA with all:',
    leaveOneOut(LDAmodel, data_dummies, x_columns, y_column)
    )
print(
    'QDA with good:',
    leaveOneOut(QDAmodel, data_dummies, X_q, y_column)
    )
print(
    'QDA with all:',
    leaveOneOut(QDAmodel, data_dummies, x_columns, y_column)
    )
#LDA with good: 0.8253333333333334
#LDA with all: 0.804
#QDA with good: 0.7973333333333333
#QDA with all: 0.62

################
### Boosting ###
################

#Our initial testing found that the boosting
# classifier performed well with the following hyperparameters.
model = AdaBoostClassifier(
                        tree.DecisionTreeClassifier(max_depth=1),
                        n_estimators = 180,
                        learning_rate=0.2
                        )
score = leaveOneOut(model, data, x_columns, y_column)
print(score)
#0.8173333333333334

# We proceeded to loop over a range of different values for the
# hyperparameters, with a higher amount of folds for the
# cross validation.
result=[0,0,0]
for n in range (20,200,10):
    for l in range (1,5):
        model = AdaBoostClassifier(
                        tree.DecisionTreeClassifier(max_depth=1),
                        n_estimators = n,
                        learning_rate=l/10
                        )
        score = crossValidate(model, data, x_columns, y_column, 50)
        if result[0]<score:
            result = [score, n, l/10]
print(result)
#sample output:
#[0.8280000000000001, 180, 0.1]

# Based on our output above we then evaluated this using leaveOneOut
model = AdaBoostClassifier(
                        tree.DecisionTreeClassifier(max_depth=1),
                        n_estimators = 180,
                        learning_rate=0.1
                        )
score = leaveOneOut(model, data, x_columns, y_column)
print(score)
#0.8266666666666667
```