Song Classifier

Nathaniel Ahy
John Larsson
Florian Dietrich

Abstract

In this paper we utilize five families of statistical machine learning (SML) algorithms to predict whether Andreas Lindholm would like songs within given set. We thereafter selected the Gradient Boosting method as it yielded minimal errors in a 10-fold cross-validation.

1 Introduction

Given the abundance of data produced on a regular basis machine learning algorithms have emerged as one of the most powerful methods to make predictions. Due to its popularity numerous algorithms have surfaced each having their own distinctive strengths and weaknesses. With such a large number of methods to select a major task of machine learning is choosing the right model for a given dataset. We were tasked with predicting which songs Andreas Lindholm would like using a set of characteristics, within a set of 800 labeled songs. We employed methods from five families of machine learning algorithms, each of which we provide an outline for, and selected the output of the method which yielded the best performance in a 10-fold cross-validation.

2 Methods for Predicting Songs

We considered 5 method families for predicting songs:

- Logistic Regression
- Discriminant Analysis (both linear and quadratic)
- K Nearest Neighbors
- Tree-Based Methods
- Boosting Methods

2.1 Logistic Regression

In Logistic regression we find the vector of parameters θ so that our function $g(x;\theta)$, where x is a vector of inputs, provides the most accurate probability as to whether Andreas will like a given song with feature vector x. In our case

$$g(x;\theta) = \frac{1}{1+e^{-\theta^\top x}}$$

Thus, we will predict that Andreas likes a song with feature x if and only if $g(x;\theta) \geq 0.5$. Our problem is therefore that we want to find the vector θ that maximizes the probability that the algorithm will predict a given outcome $y_i \in \{-1,1\}$ with -1 indicating Andreas doesn't like song i and 1 that he does like song i. Mathematically we get that our estimation for $\hat{\theta}$ is found through:

$$\hat{\theta} = arg\max_{\theta} p(y|X;\theta) = arg\max_{\theta} \sum_{i=1}^{n} \ln p(y_i|x_i;\theta),$$

where X is the matrix with column i representing the feature vector x_i, which along with y_i is given as input data. Clearly $1 - g(x_i;\theta)$ is the probability Andreas won't like song i, using this identity and the above equation we acquire Equation 3.37 in the course literature (Smlbook)[1]:

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

which is solved using numerical methods.

2.2 Discriminant Analysis

The idea behind discriminant analysis is it describes how data {x,y} is generated. We want to find out whether to predict that Andreas likes a song given its features: $p(y = 1|x) \geq 0.5$. Letting $Y \in \{-1, 1\}$ be a potential value for y then

$$p_{LDA}(x|y = Y) = \begin{cases} \mathcal{N}(x|\mu_{-1}, \Sigma), & Y = -1 \\ \mathcal{N}(x|\mu_1, \Sigma), & Y = 1 \end{cases}$$

in the case of linear discriminant analysis (LDA) and

$$p_{QDA}(x|y = Y) = \begin{cases} \mathcal{N}(x|\mu_{-1}, \Sigma_{-1}), & Y = -1 \\ \mathcal{N}(x|\mu_1, \Sigma_1), & Y = 1 \end{cases}$$

in the case of Quadratic Discriminant Analysis (QDA). Meaning we have mean vector μ_1 and covariance matrix Σ_1 in the QDA setting when $Y = 1$ as well as μ_{-1} Σ_{-1} when $Y = -1$. The vectors and matrices μ_Y, and Σ_Y are computed using the standard definition of mean and covariance matrix for the song's different features when the song belongs to class Y. Denoting $\hat{\pi_m}$ as the share of songs belonging to class Y, applying Bayes' Theorem, and using the above equalities we get that in the QDA case:

$$p(y = Y|x_*) = \frac{\hat{\pi_Y}\mathcal{N}(x_*|\hat{\mu_Y}, \hat{\Sigma_Y})}{\sum_{j\in\{-1,1\}} \hat{\pi_j}\mathcal{N}(x_*|\hat{\mu_j}, \hat{\Sigma_j})}.$$

One merely uses the covariance matrix for the entire dataset in the LDA case. Finally, we classify song * according to:

$$\hat{y}_* = arg\max_{Y} p(y = Y|x_*)$$

2.3 K Neighbors Classifier

The idea behind the K Neighbors Classifier is extremely simple. Letting p denote the number of features we take into account we place our data points in p dimensional space, observe the k nearest[2] data points of a given observation and assign that observation to the class that has the majority share of said k points.

2.4 Tree-based methods

The main idea behind the so-called tree-based methods is to divide the input space into complementary regions and to use very simple predictors for these regions. Namely, a majority vote in the case of a classification problem or a simple mean in the case of a regression problem. The problem with the approach is that finding the optimal partitioning of the input space is a computationally infeasible optimisation problem, even if just simple shapes like squares are assumed. The most common approach used in practice is the so-called recursive binary splitting. This is a "greedy" algorithm that optimises each split at a time. In each step one of the input variables x_j and a cut-point s is selected. The input space is then partitioned into two half-spaces:

$$R_1(j, s) = \{\mathbf{x} : x_j < s\}$$
$$R_2(j, s) = \{\mathbf{x} : x_j \geq s\}$$

[1]Supervised Machine Learning, Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, Thomas B. Schön,Draft version: February 14, 2020

[2]One mostly uses the Euclidean norm to evaluate distance in this case.

This is repeated until a certain criterion is met or the tree is "fully grown". In the latter case each training data point has its own region reducing the training error E_{Train} to zero.

In each cycle only a finite number of cutting points *s* exist that yield different regions R_n. Therefore, during each iteration *j* and *s* can be chosen by optimising a loss-function with "brute force". The loss function for regression trees typically looks like this:

$$L(j,s) = \sum_{i:\mathbf{x}_i \in R_1(j,s)} (y_i - \hat{y}_1(j,s))^2 + \sum_{i:\mathbf{x}_i \in R_2(j,s)} (y_i - \hat{y}_2(j,s))^2$$
$$with \quad \hat{y}_n = \text{average}\{y_i : \mathbf{x}_i \in R_n(j,s)\}$$

The process of training classification trees has two main differences. For one, the predictions are not done via regression (see above) but rather through a majority vote or rather a set of class of probabilities is returned by assuming that the probability of class *m* for a given input ***x*** is determined by the proportion of training data points of class *m* in the region R_l:

$$\hat{\pi}_{l,m} = \frac{1}{n_l} \sum_{i:\mathbf{x}_i \in R_l} \mathbb{I}\{y_i = m\}$$
$$p(y = m|\mathbf{x}) \approx \sum_{l=1}^{L} \hat{\pi}_{l,m} \mathbb{I}\{\mathbf{x} \in R_l\}$$

The other difference is that the loss function is replaced by a categorical alternative, like:

$$\text{Miss-classification error}: \quad 1 - \max_m \hat{\pi}_{l,m}$$
$$\text{Entropy/deviance}: \quad -\sum_{m=1}^{M} \hat{\pi}_{l,m} \log \hat{\pi}_{l,m}$$
$$\text{Gini index}: \quad \sum_{m=1}^{M} \hat{\pi}_{l,m}(1 - \hat{\pi}_{l,m})$$

2.4.1 Bagging

Typically, classification and regression trees (CARTs) suffer from poor performance even compared to rather rudimentary methods. The reason for that is in order to achieve a low bias very deep trees are needed. Those, however, suffer from high variance. The principle behind bagging is that the variance of random variables can be reduced by averaging. The predictor or the "bagging-tree" is obtained by averaging over a set of trees:

$$\tilde{y}^b_{bag}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^{B} \tilde{y}^b(\mathbf{x})$$

The training data to train *B* trees is obtained via bootstrapping. Bootstrapping creates multiple data sets from the original data set by drawing *n* times with replacement from the original for each new data set that is to be created.

2.4.2 Random Forests

The problem with bagging is that the variance reduction is diminished by the fact that the *B* bootstrapped data sets are correlated. The random forest method solves that problem by randomly perturbing each tree. This leads to a higher variance in each individual tree but an overall reduction in error. The perturbation is done by only considering a random subset $q \leq p$ data points when making each split in the training process.

2.5 Boosting

Boosting similarly to bagging, is a method of using "weak" models in a clever way for a better result. Here we only consider boosting used with trees for classification between two different categories. Intuitively in boosting several models are trained sequentially where in each step more emphasis is put on the errors of the previous models and the end result is a weighted average of the models with larger weights on the models with lower errors. The models and their weights are found by minimizing a cost function in each step. This is quite clear in AdaBoost where the exponential loss function is used and an exact solution exists.

2.5.1 AdaBoost

Consider a classification problem with the output $y_i \in \{-1, 1\}$. Denote the B models trained in a boosting scheme by $\hat{y}^{(1)}(\mathbf{x}), \ldots, \hat{y}^{(B)}(\mathbf{x})$ and the combined predictor

$$\hat{y}(\mathbf{x}) = \text{sign}\left[c^{(B)}(\mathbf{x})\right],$$

where

$$c^{(b)} = c^{(b-1)} + \alpha^{(b)}\hat{y}^{(b)}.$$

starting with $c^{(0)} = 0$. The final prediction is a combination of all the trained models with weights $\alpha^{(b)}$. The choices $\alpha^{(b)}$ and $\hat{y}^{(b)}$ are done by minimizing the loss function and can be derived as in the Smlbook (draft 14feb) equation (7.9)

$$\hat{y}^{(b)} = \arg\min_{\hat{y}} \sum_{i=1}^{n} w_i^{(b)} \mathbb{1}_{\{\hat{y}(x_i) \neq y_i\}}$$

where the weights are defined as $w_i^{(b)} = \exp\left(-y_i c^{(b-1)}(x_i)\right)$. Here we can see that if y_i and $c^{(b-1)}(x_i)$ have the same sign (i.e. data point i was correctly classified by $c^{(b-1)}$) then the exponent will be negative and if the absolute value of $c^{(b-1)}(x_i)$ is large this results in a small weight. The converse is true for misclassified points where the exponents will be positive and larger for larger absolute values of $c^{(b-1)}(x_i)$. Further derivation yields equation (7.10) and (7.11)

$$\alpha^{(b)} = \frac{1}{2}\ln\left(\frac{1 - E^{(b)}_{\text{train}}}{E^{(b)}_{\text{train}}}\right),$$

$$E^{(b)}_{\text{train}} = \sum_{i=1}^{n} \frac{w_i^{(b)} \mathbb{1}_{\{\hat{y}(x_i) \neq y_i\}}}{\sum_{i=1}^{n} w_i^{(b)}}.$$

Here we can see that since $E^{(b)}_{\text{train}} \in [0, 1]$, we will get that $\alpha^{(b)} \to \infty$ as $E^{(b)}_{\text{train}} \searrow 0$, meaning that the predictor $\hat{y}^{(b)}$ has a large contribution to the overall prediction. We also get $\alpha^{(b)} = 0$ when $E^{(b)}_{\text{train}} = \frac{1}{2}$ and lastly $\alpha^{(b)} \to -\infty$ as $E^{(b)}_{\text{train}} \nearrow 1$. Even though the weight should never be negative we can intuitively reason that if the weighted error is low we want a positive contribution from that estimator and if the weighted error is very high the estimator is doing exactly the opposite of what we want and we want the opposite of its prediction.

2.5.2 Gradient Boosting

Gradient boosting is a generalization of AdaBoost where we also consider other loss functions. Since it may not be possible to derive a solution as in the case of exponential loss, the algorithm must be altered. In algorithm 11 from Smlbook (draft 14feb) the negative partial derivatives of the loss function is used

$$g_i^{(b)} = -\left(\frac{\partial L(y_i, c)}{\partial c}\right)_{c = c^{(b-1)}(x_i)}.$$

Then a regression $\hat{f}^{(b)}$ is fitted on the values $g_i^{(b)}$ and the model is updated as $c^{(b)}(\mathbf{x}) = c^{(b-1)}(\mathbf{x}) + \gamma\hat{f}^{(b)}(\mathbf{x})$. The weights γ can be chosen as in a gradient descent algorithm. This model is more difficult to interpret but the values $g_i^{(b)}$ have a similar function as the previous weights $w_i^{(b)}$ in the sense that extreme values have the largest effect on the regression. In GradientBoostingClassifier from ScikitLearn.ensemble you can choose either exponential loss function or binomial deviance.

3 Method performances and Selection

The first three methods mentioned in the Methods section had the most straightforward application:

- LDA: test the default Singular Value Decomposition method, then least squares solution, and eigenvalue (spectral) decomposition. All of which yielded similar results
- QDA: we merely used the default options here as the LDA had a far better training error
- $k - NN$: we simply varied the number of neighbors

We used a tenfold cross validation on the training set to compare the performance of the methods in order to choose which one to use in production. We can rule out KNN right away because of its atrocious error of 39% in the cross validation, compared to the simple classifier which gets a 40% error. Of the three first methods introduced LDA performed the best with a 10-fold error of roughly 0.19 whereas QDA yielded an error of approximately 0.24.

The next family of methods that was analysed, were the tree-based methods. In the beginning, a CART, a tree obtained from bagging and one obtained from a random forest were trained with default settings and compared using 10-fold cross-validation. The expected trends that Bagging is an improvement over CARTs and that Random Forests are an improvement over simple Bagging, were clearly visible. Each method performed about 4% better than the respective simpler version in terms of accuracy. The best 10-fold cross-validation accuracy in this step, was therefore obtained from the Random Forest with default settings (83.6%).

In the next steps three input parameters for the Random Forest classifier were optimised. Namely, the number of data-points required for a split in a leaf, the chosen loss-function (Gini index or Enthalpie) and the random pertubation or rather how many data points were considered for each split ($\sqrt{n}$ or $\log_2(n)$).

For the leaf-size the values 2 to 21 were considered. For each value the 10-fold validation error was calculated 20 times and averaged. The results are plotted in figure 2 in the appendix. The graph indicates that the choice of samples required for the split only has a rather minor impact on the accuracy of the model. However, a trend can be seen which indicates that the maximum accuracy is achieved with values between 5 and 10 and that higher values slowly lead to a decrease in accuracy.

The same approach was used to choose the other two parameters. The Gini index turned out to be the better loss-function with a slightly higher accuracy (0.2%). Similarly, it turned out to better to consider $\sqrt{n}$ data-points for each split by a small margin of again 0.2%.

For the boosting methods we tried both AdaBoost with decisiontreeclassifer as the base estimator and gradient boosting with the deviance loss function. In AdaBoost the 10-fold cross validation error hovered around 20% on the training data and different choices of parameters seemingly made no meaningful difference. We did not compare different base estimators in AdaBoost. The gradient boosting function in scikit-learn used either a exponential loss function or deviance. Since the exponential loss function would have reduced it to the same as AdaBoost we only used the deviance loss function. The only parameter that seemed to make a meaningful difference in 10-fold cross validation error was the maximum depth of the base tree estimators. In the end the cross validation error was around 17% with the maximum depth set as three.

4 Conclusion

Both the gradient boosting and the Random Forest yielded rather similar 10-fold cross-validation results of 0.828 and 0.836 respectively. Therefore, both of the methods were tried out in production. It was however expected that the the Random Forrest would perform marginally better or at least evenly well compared to gradient boosting. But in production the gradient boosting outperformed the random forest by 2 percentage points. One may only speculate about the origin of that difference. The Random forest seems to have over-fitted the training data compared to the gradient boosting. This should, however, be indicated in the cross-validation performance. A shortcoming of this paper is that we did not experiment with various inputs (i.e. using differing subsets of song characteristics). Additionally, there are many more options of the family of methods themselves which we could have explored.

5 Reflection Task b

By basing our beliefs on the Code of Honour for Engineers, primarily point 3 of their itemization,[3] it is crucial that an engineer provides the best possible basis for decision-making within the context of their product. This can only be done by discussing the merits as well as the shortcomings of one's own products. Therefore if there is a shred of possibility that the product can have some form of bias then it is necessary for the engineer to convey this issue to the extent it entails, that is, not to under- nor over-exaggerate this issue. The aforementioned argument pertains heavily to points 8, and 9, of said Code of Honour; an engineer should strive to remain impartial, or at the very least report what could contribute to one's own partiality (8), in this case the provider of said product, us, risk intimidating our client by the systematic biases within our ML product leading them to take their business to a team who has either overcome these biases or withholds them from said client. Additionally, one should strive for factual presentation and minimize misleading statements meaning one should report the whole truth and nothing else (9); thereby describing the ML product as it is and not in the manner that makes it as attractive as possible.

Had every engineer followed the above description then it would likely lead to the most productive outcome since the issues addressed would be open thereby encouraging attempts to mitigate or eliminate these shortcoming of the product. However, one would need to ensure that every engineer would do such a thing, if it is the case that two engineering parties have an equally competent ML product then it is possible that the party who does the best job at hiding their flaws will dominate the market. This gives the more deceiving party incentive to not fix their product's biases since in order to fix a problem one must acknowledge its existence. A high number of such scenarios could be the end of engineers who follow the above-mentioned code of honour. But perhaps if neither party followed said code of honour then eventually the market, clients on a macro scale, would select the product that performs the best which is the product with the minimum bias. Thus, leaving the burden of identifying the shortcomings of said product to the clients could bring the existence of the least biased algorithm.

Another point can be made about the first point in the Code of Honour for Engineers, about striving for technology use to benefit humanity. In the case of ML being used for an insurance company, the biases in the models may be subtle enough that they only affect a small minority of the customers. If this is overlooked by not informing the customer this could easily be seen as a violation of that point of the Code of Honour.

An argument one may make against informing the customer is that the first point of the code of honour could be interpreted as saying that it is your responsibility as the engineer to make sure the technology is working as intended and does not harm people. Educating the customer about the potential machine learning bias, invites him/her to project her own personal bias onto the training data, subconsciously or even consciously. Even if the customer has the best intentions it has to be assumed that he/she has no expertise in the subject of machine learning which causes a high likelihood of miss-understandings in the education about machine learning biases. This makes it likely that the customer will struggle differentiating between actual trends and biases. Which again, redirects the responsibility back to the engineer since he is the one with the expertise to deal with this issue.

A Code for LDA, QDA, k-NN, and Logistic Regression

```
#LDA, QDA, Logistic Regression, and k-Nearest Neighbors
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
```

[3] Sveriges Ingenjörer (2018). Code of Honour. URL: https://arkiv. sverigesingenjorer. se/Global/Dokumentbibliotek/Hederskodex%20ENG%20till%20webb.pdf.

```
import sklearn.neighbors as skl_nb
import sklearn.model_selection as skl_ms
url='http://www.it.uu.se/edu/course/homepage/sml/project/training_data.csv'
url2='http://www.it.uu.se/edu/course/homepage/sml/project/songs_to_classify.csv'

songs=pd.read_csv(url, dtype={'ID': str}).dropna().reset_index(drop=True)
songs_test=pd.read_csv(url2, dtype={'ID': str}).dropna().reset_index(drop=True)
#use k-fold cross validation to test songs
n_split=10
#we tested for n_split=3,4, and 10, LDA was always the best
cross_val=skl_ms.KFold(n_splits=n_split,shuffle=True,random_state=2)

number_of_k=100#for k nearest neighbors
error_k=np.zeros(100)
error_LDA=0
error_QDA=0
error_LR=0
X=songs.copy().drop(columns=['label'])
y=songs['label']

model_LDA=skl_da.LinearDiscriminantAnalysis(tol=10**(-9))#adjusting the tolerance
#made very little difference, and svd, eigen, as well as lsqr all performed
#the same
model_QDA=skl_da.QuadraticDiscriminantAnalysis(tol=10**(-9))
model_LR=skl_lm.LogisticRegression(multi_class='ovr')

  #need to try all the different k-values
for k in range(number_of_k):#I doubt any n-neighbor model above 100 would be good
  model_k=skl_nb.KNeighborsClassifier(n_neighbors=k+1)#start at 0

  for train, test in cross_val.split(X):
    X_train,X_test=X.iloc[train], X.iloc[test]
    y_train, y_test=y.iloc[train],y.iloc[test]

    model_k.fit(X_train,y_train)
    pred_k=model_k.predict(X_test)
    error_k[k]=error_k[k]+np.mean(pred_k!=y_test)

    if(k==0):#we only have to do these once
      model_LDA.fit(X_train,y_train)
      pred_LDA=model_LDA.predict(X_test)
      error_LDA=error_LDA+np.mean(pred_LDA!=y_test)

      model_QDA.fit(X_train,y_train)
      pred_QDA=model_QDA.predict(X_test)
      error_QDA=error_QDA+np.mean(pred_QDA!=y_test)

      model_LR.fit(X_train,y_train)
      pred_LR=model_LR.predict(X_test)
      error_LR=error_LR+np.mean(pred_LR!=y_test)

  error_k[k]/=n_split#the error for k+1 nearest neighbors

error_LDA/=n_split#average error
error_QDA/=n_split
error_LR/=n_split
#now find the k with lowest error i.e. the minimum argument and compare, also
#implement logistic regression
min_nb=np.amin(error_k)
min_k=np.where(error_k==min_nb)
plt.plot(error_k)
print(f"error_LDA:{error_LDA}")
print(f"error_QDA:{error_QDA}")
print(f"error_LR:{error_LR}")
print(f"error_k:{min_nb} for k=:{min_k}")#for k=93+1=94, but according to the
```

```
#plot is seems like anything from 40-45 is the best, although LDA seems to be
#the best choice of all our models

#We now fit the best model to ALL the data and send that string
#get the final string to submit
model_final=skl_da.LinearDiscriminantAnalysis(tol=10**(-15))#extra high tolerance
model_final.fit(X,y)
predict=model_final.predict(X)
error=np.mean(predict!=y)
print(error)
predict_submission=model_final.predict(songs_test)
print(predict_submission)
```

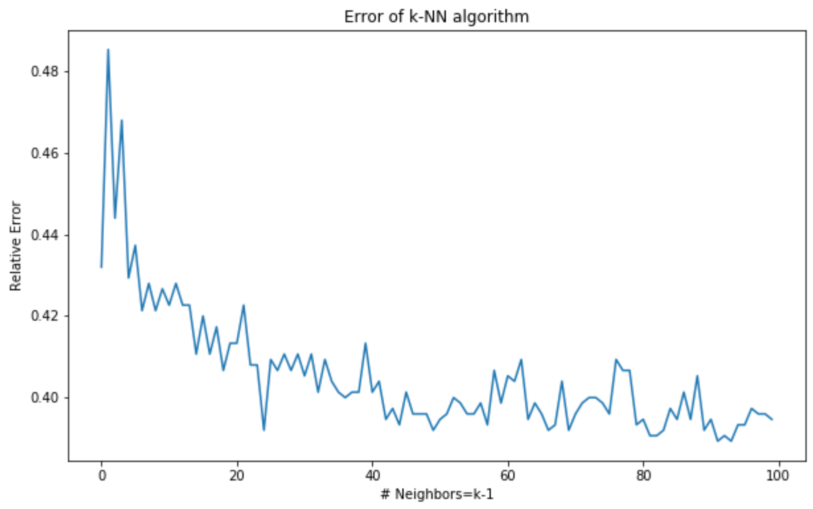

Figure 1: Errors for different number of neighors in k-NN algorithm

Output:

```
error_LDA:0.19333333333333333
error_QDA:0.24266666666666664
error_LR:0.39466666666666667
error_k:0.38933333333333336 for k=:(array([93]),)
0.184
[0 0 0 1 0 1 1 1 1 0 1 1 0 1 1 0 1 0 1 1 0 0 1 1 0 0 1 0 0 0 1 1 0 1 1 1 1
 1 0 1 0 1 0 1 0 1 0 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 0 0 1 1 1 1 1 0 1 0 1 1
 1 1 0 1 1 0 1 1 0 1 0 1 1 1 1 0 1 0 0 0 1 0 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1
 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1
 1 0 1 0 0 0 1 1 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 1 1 1 0 0 1 1 1 0 0 1 1 1 1
 1 1 0 1 0 0 1 1 1 1 0 0 1 1 1]
```

B Code Random Forest

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

from sklearn import tree
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
import sklearn.preprocessing as skl_pre
import sklearn.model_selection as skl_ms

import graphviz

train = pd.read_csv('training_data.csv')
target = pd.read_csv('songs_to_classify.csv')
X_train = train.copy().drop(columns='label')
y_train = train['label']

#CART
#10-fold cross validation
n_fold = 10
cv = skl_ms.KFold(n_splits=n_fold, shuffle=True)
accuracy = 0

for train_index, val_index in cv.split(X_train):
    X_k = X_train.iloc[train_index]
    X_val = X_train.iloc[val_index]
    y_k = y_train.iloc[train_index]
    y_val = y_train.iloc[val_index]

    cart = tree.DecisionTreeClassifier(min_samples_split=5)
    cart.fit(X_k,y_k)
    y_hat = cart.predict(X_val)
    accuracy += np.mean(y_hat == y_val)

accuracy /= n_fold
print("Accuracy CART: ", accuracy)

#Bagging
#10-fold cross validation
n_fold = 10
cv = skl_ms.KFold(n_splits=n_fold, shuffle=True)
accuracy = 0

for train_index, val_index in cv.split(X_train):
    X_k = X_train.iloc[train_index]
    X_val = X_train.iloc[val_index]
    y_k = y_train.iloc[train_index]
    y_val = y_train.iloc[val_index]

    cart = BaggingClassifier()
    cart.fit(X_k,y_k)
    y_hat = cart.predict(X_val)
    accuracy += np.mean(y_hat == y_val)

accuracy /= n_fold
print("Accuracy Bagging: ", accuracy)

#Random Forrest
#10-fold cross validation
n_fold = 10
cv = skl_ms.KFold(n_splits=n_fold, shuffle=True)
accuracy_gini = 0

for train_index, val_index in cv.split(X_train):
    X_k = X_train.iloc[train_index]
    X_val = X_train.iloc[val_index]
    y_k = y_train.iloc[train_index]
    y_val = y_train.iloc[val_index]

    cart = RandomForestClassifier()
    cart.fit(X_k,y_k)
    y_hat = cart.predict(X_val)
    accuracy_gini += np.mean(y_hat == y_val)

accuracy_gini /= n_fold
print("Accuracy RF: ", accuracy_gini)

#Optimise tree depth
min_values = np.arange(2,22,1)
accuracies = []

for min_samp in min_values:
    accuracy=0
    repeats=20
    for i in range(repeats):
        n_fold = 10
        cv = skl_ms.KFold(n_splits=n_fold, shuffle=True)

        for train_index, val_index in cv.split(X_train):
            X_k = X_train.iloc[train_index]
            X_val = X_train.iloc[val_index]
            y_k = y_train.iloc[train_index]
            y_val = y_train.iloc[val_index]

            cart = RandomForestClassifier(min_samples_split=min_samp)
            cart.fit(X_k,y_k)
            y_hat = cart.predict(X_val)
            accuracy += np.mean(y_hat == y_val)

    accuracies.append(accuracy/(n_fold*repeats))

plt.plot(min_values, accuracies)
plt.ylabel('50-fold CV accuracy')
plt.xlabel('Min samples for split')
plt.savefig('Minsplit.png', dpi=500)
plt.show()
#Impact minimal seems to fall of after 20 -> 5-10

#Gini vs. Entropy
accuracy_e = 0
accuracy_g = 0
repeats = 20
for i in range(repeats):
    n_fold = 10
    cv = skl_ms.KFold(n_splits=n_fold, shuffle=True)


    for train_index, val_index in cv.split(X_train):
        X_k = X_train.iloc[train_index]
        X_val = X_train.iloc[val_index]
        y_k = y_train.iloc[train_index]
        y_val = y_train.iloc[val_index]

        cart_e = RandomForestClassifier(criterion='entropy')
        cart_g = RandomForestClassifier()
        cart_e.fit(X_k,y_k)
        cart_g.fit(X_k,y_k)
        y_hat_e = cart_e.predict(X_val)
        y_hat_g = cart_g.predict(X_val)
        accuracy_e += np.mean(y_hat_e == y_val)
        accuracy_g += np.mean(y_hat_g == y_val)

accuracy_e /= (n_fold*repeats)
accuracy_g /= (n_fold*repeats)
print("Accuracy Enthalpie: ", accuracy_e, "\n")
print("Accuracy Gini: ", accuracy_g)
```

```
#Gini slightly better

#Randomness criterion
accuracy_log = 0
accuracy_sqrt = 0
repeats = 20
for i in range(repeats):
    n_fold = 10
    cv = skl_ms.KFold(n_splits=n_fold, shuffle=True)


    for train_index, val_index in cv.split(X_train):
        X_k = X_train.iloc[train_index]
        X_val = X_train.iloc[val_index]
        y_k = y_train.iloc[train_index]
        y_val = y_train.iloc[val_index]

        cart_log = RandomForestClassifier(min_samples_split=8,max_features='log2')
        cart_sqrt = RandomForestClassifier(min_samples_split=8)
        cart_log.fit(X_k,y_k)
        cart_sqrt.fit(X_k,y_k)
        y_hat_log = cart_log.predict(X_val)
        y_hat_sqrt = cart_sqrt.predict(X_val)
        accuracy_log += np.mean(y_hat_log == y_val)
        accuracy_sqrt += np.mean(y_hat_sqrt == y_val)

accuracy_log /= (n_fold*repeats)
accuracy_sqrt /= (n_fold*repeats)
print("Accuracy log2: ", accuracy_log, "\n")
print("Accuracy sqrt: ", accuracy_sqrt)
#sqrt slightly better

the_tree = RandomForestClassifier(min_samples_split=8, max_features='log2',
     criterion='gini')
the_tree.fit(X_train, y_train)
y_hat = the_tree.predict(target)
print(y_hat)

naive = np.ones(750)
print("Accuracy simple classifier: ",np.mean(naive==y_train))
```

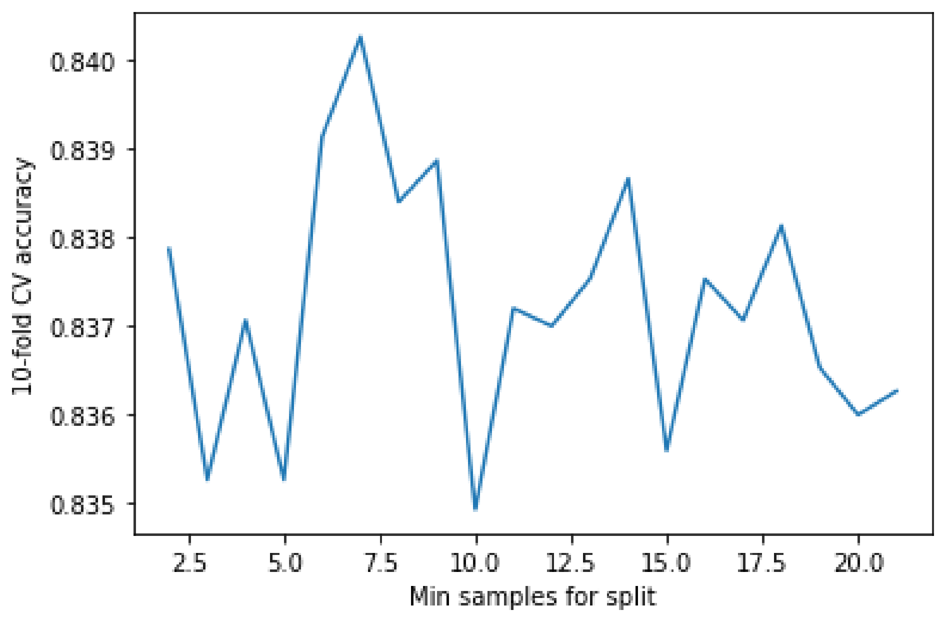

Figure 2: This figure shows the accuracy's dependency on the number of data-points required for a split when training the random forest.

Output:

```
Accuracy CART:  0.7693333333333333
Accuracy Bagging:  0.808
Accuracy RF:  0.8360000000000001

Accuracy Enthalpie:  0.8350666666666663
Accuracy Gini:  0.837133333333333

Accuracy log2:  0.8366666666666664
Accuracy sqrt:  0.8387333333333332

[0 0 0 1 0 0 0 1 0 0 1 1 0 1 1 0 1 0 1 1 1 0 1 1 0 0 0 0 0 0 1 1 0 1 1 1 1
 1 0 1 0 1 0 1 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 1 0 0 1 1 1 1 0 1 0 1 1
 1 1 0 1 1 0 1 1 0 1 0 1 1 0 1 0 0 0 0 0 1 0 1 1 1 1 1 1 1 0 0 1 0 1 1 1 1
 1 0 0 1 0 1 0 1 0 0 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1 0 1 0 1 1 0 0
 1 0 1 0 0 0 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 0 1 1 0 1 1 1 1 1
 1 1 0 1 0 0 1 1 1 1 1 0 1 1 1]

Accuracy simple classifier:  0.6026666666666667
```

C Code Boosting

```python
import pandas as pd
import numpy as np
import sklearn as skl
import sklearn.ensemble as skle
import matplotlib
import matplotlib.pyplot as plt

def kferror(model,n_split):
    #takes the arguments model and n_splits for kfold error estimation
    kf=skl.model_selection.KFold(n_splits=n_split,shuffle=True,random_state=2)
    model_err=0
    for train, test in kf.split(X):
        X_train,X_test=X.iloc[train], X.iloc[test]
        Y_train, Y_test=Y.iloc[train],Y.iloc[test]
        model.fit(X=X_train,y=Y_train)
        model_err+=np.mean(model.predict(X_test)!=Y_test)
    model_err/=n_split
    return model_err

df =
    pd.read_csv('http://www.it.uu.se/edu/course/homepage/sml/project/training_data.csv')
classify=pd.read_csv('http://www.it.uu.se/edu/course/homepage/sml/project/songs_to_classify.csv')
X=df.copy().drop(columns='label')
Y=df['label']

n_split=10
kfdev = skle.GradientBoostingClassifier(max_depth=3,min_samples_leaf=2)
kfada = skle.AdaBoostClassifier()

dev_err=kferror(kfdev,n_split)
ada_err=kferror(kfada,n_split)

print(dev_err)
print(ada_err)

dev = skle.GradientBoostingClassifier(n_estimators = 100,max_depth=3)
#ada = skle.AdaBoostClassifier()

dev.fit(X=X,y=Y)
#ada.fit(X=X,y=Y)
#gradient boosting ended up winning

prediction=dev.predict(X=classify)
print(prediction)
```

```
dev_err = 0.172
ada_err = 0.20666666666666667
```