Machine Learning project

Edoardo Berti, Ruben Martinez Cuella, Giacomo Gattorno, Pamela Scaltrito

12/18/2020

Predicting Spotify data

Motivation

This project presents a wide variety of Machine Learning (ML) techniques to predict two outcome variables: the popularity of a song and its genre. Besides, we attempt to compare the performance of the models by using the appropriate measures. The motivation behind this analysis is two-fold: on the one hand pedagogical, in order to be able to use both regression and classification algorithms; on the other hand pragmatic, given that similar analysis can be used in real-life applications, as we will discuss in a moment.

Database

The database has been retrieved manually through the Spotify API, for which we have obtained 7,453 observations with 13 audio features for each song. Next, we have downloaded the lyrics for those observations using the Genius API, obtaining an additional variable containing the lyrics for each song. We then performed a basic sentiment analysis using the **syuzhet** package. It is based on a dictionary of English words and their associations with eight basic emotions and two sentiments (negative and positive). Thus, we could add 10 additional variables for each song based on this simple sentiment analysis. This brings in an interesting question that is answered through the report: can performing a basic sentiment analysis improve the predictive power of the models substantially? It could potentially boost predictive power with low effort. The variables are the following:

- `genre`: genre of the song. Factor with 7 levels: rock, pop, rap, EDM, Jazz, songwritter and metal
- `popularity`: value between 0-100. It is based, in the most part, on the total number of plays the track has had and how recent those plays are.
- `danceability`: continous variable between 0-1. How suitable it is for dancing.
- `energy`: continous measure from 0-1. It represents a perceptual measure of intensity and activity.
- `acousticness`: variable between 0-1.Determines whether the track is acoustic.
- `instrumentalness`: variable between 0-1, depending how many vocal sounds it has.
- `liveness`: variable between 0-1. It represents the likelihood of being a live performance.
- `loudness`: countinous variable indicating debibels (dB). Typical range is -60 to 0.
- `speechiness`: variable from 0-1 indicating the presence of spoken words in the track.

- `valence`: measure from 0-1 describing musical positiveness.
- `tempo`: overall estimated tempo of a track in beats per minute (BPM).
- `duration_ms`: duration of the track in milisecons (ms).
- `key`: the overall key of the track. Range from 0 to 11.
- `mode`: variable that indicates the modality of the track.
- `time_signature`: measures how many beats there are in every bar.
- `anger`: percentage of words in lyrics that trigger the emotion anger.
- `anticipation`: percentage of words in lyrics that trigger the emotion anticipation.
- `disgust`: percentage of words in lyrics that trigger the emotion disgust.
- `fear`: percentage of words in lyrics that trigger the emotion fear.
- `joy`: percentage of words in lyrics that trigger the emotion joy.
- `sadness`: percentage of words in lyrics that trigger the emotion sadness.
- `surprise`: percentage of words in lyrics that trigger the emotion surprise.
- `trust`: percentage of words in lyrics that trigger the emotion trust.
- `negative`: percentage of words in lyrics that trigger negative sentiment.
- `positive`: percentage of words in lyrics that trigger positive sentiment.

Load dataset

```
spotify_right_lyrics_popularity <-
read.csv("spotify_right_lyrics_popularity.csv")
#Genre as factor
spotify_right_lyrics_popularity <-spotify_right_lyrics_popularity %>% mutate(
                                genre=genre %>% as.factor())
#Data_final is a dataset with genre coded with dummy variables
data_final <- dummy_cols(spotify_right_lyrics_popularity, remove_first_dummy
= TRUE) %>% mutate(genre=NULL)

#Spotify is dataset in which genre is a factor
Spotify <- spotify_right_lyrics_popularity
```

1. Predicting *popularity*

In this section, we compare the predicting ability in terms of MSE of the popularity variable. Predicting popularity can be interesting for a few reasons. On the one hand, knowing which songs will become the most popular can be very interesting for many stakeholders, such as radio stations, advertising companies, etc. On the other hand, identifying what variables are most helpful to predict popularity can provide some insights to artists trying to infer better song-making processes.

We are using different regression models that allow us to compute predictions for the test set and compare it to the real outcome. As predictors, we are considering all audio features, all sentiment variables and the variable genre. This factor variable has to be encoded in some of the models.

```
#Here we perform the train/test split we will use in predicting popularity
# Training and Test Set
n = dim(spotify_right_lyrics_popularity)[1]
set.seed(1)
train.index = sample(n, round(n/2), replace = FALSE)
```

1.1 Variables Selection

Considering that we have a quite high number of variables it makes sense to employ some selection variables techniques for our analysis. The latter enable us to exclude the variables that are not strictly necessary for the predictive performance of our model, focusing only on the choice of the best ones.

1.1.1 Best Subset Selection

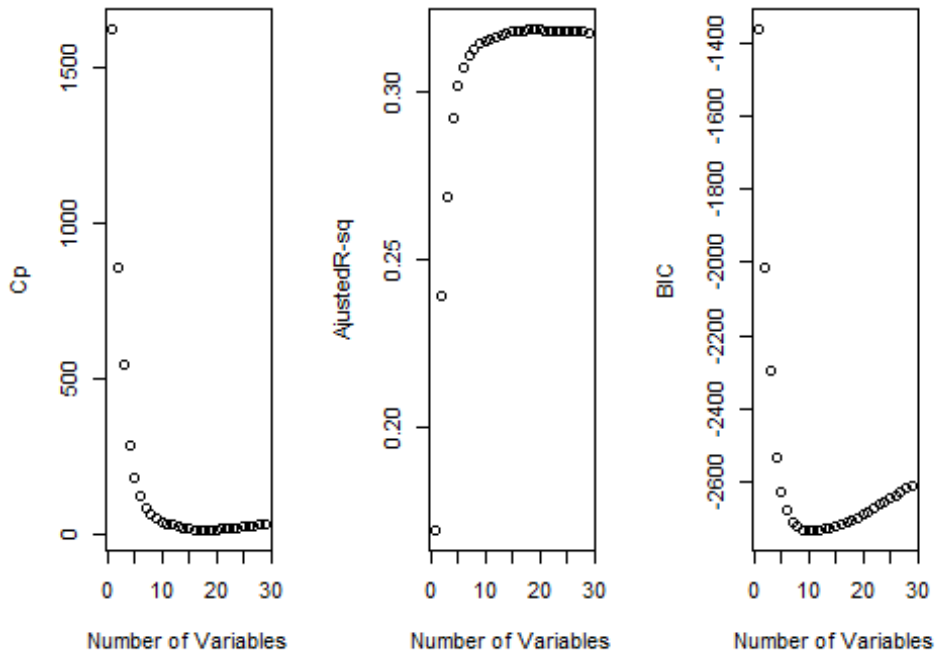
Best Subset Selection is a technique that enable us to reduce the number of variables involved in our regression task by combining all the possible models that can be computed throughout our predictive variables.

First of all we will need to define the number of predictors and the number of observation in our database.

```
dimension = dim(data_final)[1]
predictors= dim(data_final)[2]-1
```

Now with the command regsubset implemented in the leaps package we will obtain the results from the Best selection model

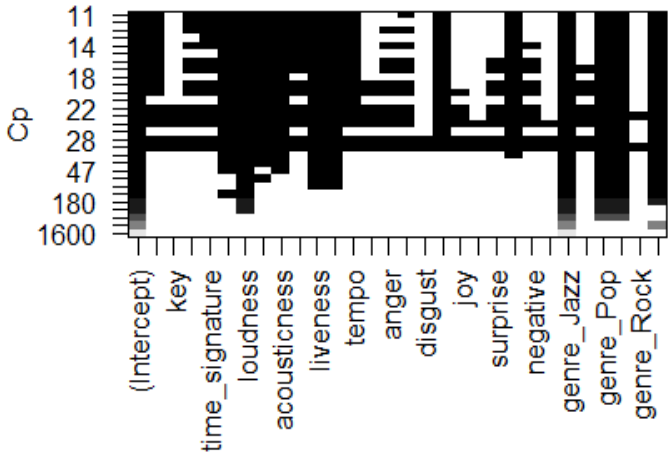
```
regfit.off = regsubsets (popularity ~ .,
                         data =data_final, nvmax=predictors,
method="exhaustive")
summary(regfit.off)
reg.summary = summary(regfit.off)

par(mfrow = c(1, 3))
plot1<-plot(reg.summary$cp, xlab = "Number of Variables", ylab = "Cp")
plot1<-plot(reg.summary$adjr2, xlab = "Number of Variables", ylab =
"AjustedR-sq")
plot2<-plot(reg.summary$bic, xlab = "Number of Variables", ylab = "BIC")
```




```r
par(mfrow = c(1, 1))
imin = which.min(reg.summary$cp)
imin#Number of variables for minimum Cp

## [1] 18

min1=min(reg.summary$cp)
min1#minimum Cp

## [1] 11.08775

imin1=which.min(reg.summary$bic)
imin1#Number of variables for minimum BIC

## [1] 10

min2=min(reg.summary$bic)
min2#minimum BIC

## [1] -2730.364

imax=which.max(reg.summary$adjr2)
imax#Number of variables for maximum R-squared

## [1] 18

max1=max(reg.summary$adjr2)
max1#maximum R-squared
```

```
## [1] 0.3181866
```

The variables selection models works through the application of a correction term to the training test error to transform it into a reliable measure of the test error. These measures are Mallow's CP, the BIC, and the Adjusted-R squared. While analysing the Mallow's CP and the adjusted-R squared, our model reached respectively a minimum and a maximum with 18 variables. By contrast, in the adoption of the BIC criterion, the minimum was reached in the model including 10 variables.



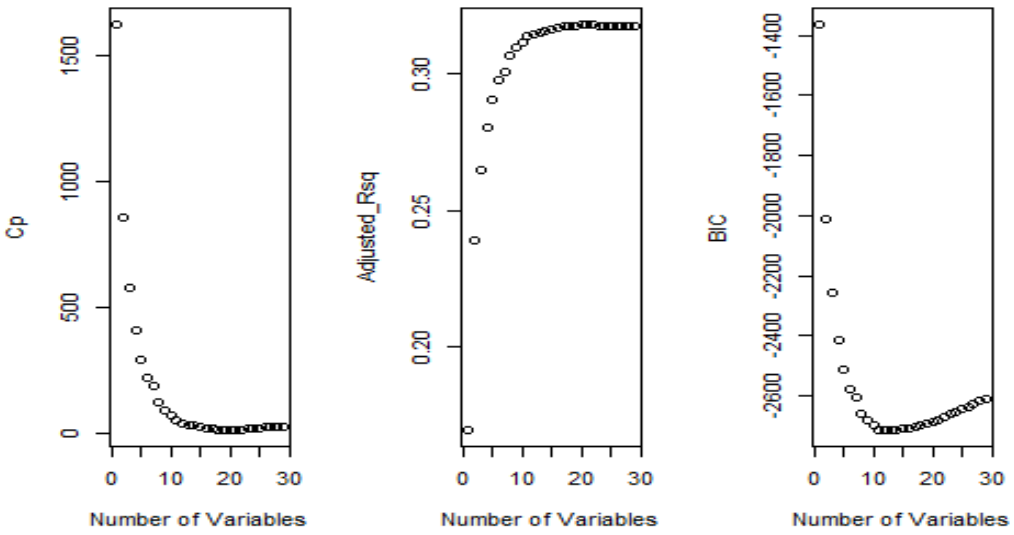
This graph shows us the variables selected by BSS and gives us an insightful understanding of the role that they play. As we were expecting, some of the genre dummies (in particular Pop, Rap and Songwriter) seem to be very important in the prediction of the popularity of a song. At the same time, we can notice that some technical features of the music like "key", "duration" and "mode" seem not to be very useful for our predictions. Furthermore, the variables that derive from sentiment analysis seem to play a minor role except "trust" and "fear" that are included in the last model with 18 variables

1.1.2 Forward Stepwise Selection

We will now run Forward Stepwise selection. This model exploits the same rationale of the BSS, but it is defined as a "greedy" algorithm. It selects, indeed, only nested models starting from the "null model" (in the FSS) to then gradually increase the number of variables included avoiding in this way the computational burden associated with BSS. If this reduces the accuracy of our predictions, it also enables us to prevent possible problems of overfitting. We will avoid showing the result of the Backward Stepwise Selection, considering that it delivers results very similar to FSS.

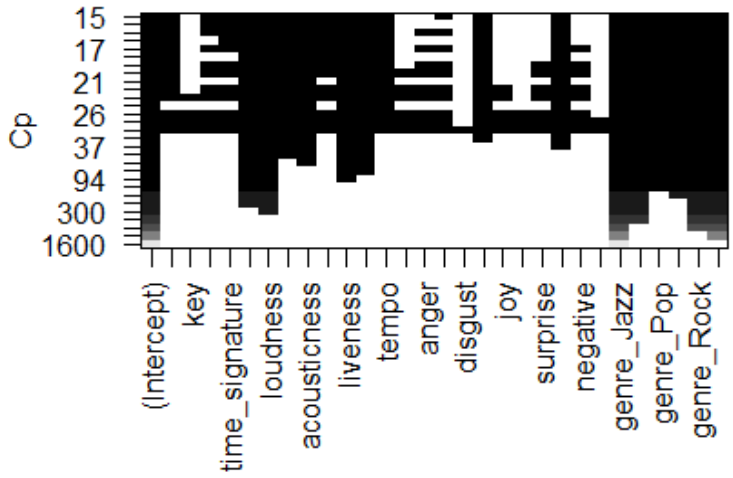
```
regfit.forward = regsubsets(popularity ~ .,
                            data = data_final, nvmax=predictors, method =
```

```
"forward")
summary(regfit.forward)
reg.summary2 = summary(regfit.forward)

par(mfrow = c(1, 3))
plot(reg.summary2$cp, xlab = "Number of Variables", ylab = "Cp")
plot(reg.summary2$adjr2, xlab = "Number of Variables", ylab = "Adjusted_Rsq")
plot(reg.summary2$bic, xlab = "Number of Variables", ylab = "BIC")
```




```
par(mfrow = c(1, 1))
# find the index corresponding to the minimum Mallow's Cp and BIC and to the
max. R-squared
iminf = which.min(reg.summary2$cp)
iminf#Number of variables for minimum Cp

## [1] 20

minf2=min(reg.summary2$cp)
minf2#minimum Cp

## [1] 14.55944

iminf1=which.min(reg.summary2$bic)
iminf1#Number of variables for minimum BIC

## [1] 12

minf3=min(reg.summary2$bic)
minf3#minimum BIC

## [1] -2712.851
```

```
imaxf=which.max(reg.summary2$adjr2)
imaxf#Number of variables for maximum R-squared

## [1] 20

maxf1=max(reg.summary2$adjr2)
maxf1 #maximum R-squared

## [1] 0.3180517

plot(regfit.forward, scale = "Cp")
```



When comparing the previous Best Selection Model and the Forward Stepwise selection, we can see that the number of variables picked by the two models changes considerably. For what concerns the analysis of the main variables employed by the model, the interpretation remains similar.

Error Measure	#of Variables BSS	#of Variables FSS
CP	18	20
Adj. R-squared	18	20
BIC	10	12

1.1.3 Choice of the model

Based on the measures that we have calculated we will now choose the best model for our predictive analysis. In particular we will consider the model that delivers the lower Mallow's Cp and BIC while the higher adjusted R-squared.

```
model_choose1<-c(min1,minf2 )#CP: min1=BSS, minf2=FSS
model_choose2<-c(min2,minf3)#BIC
model_choose3 <-c( max1,maxf1)#Adjusted-Rsquared
#min(model_choose1)
#min(model_choose2)
#max(model_choose3)
which.min(model_choose1)

## [1] 1

which.min(model_choose2)

## [1] 1

which.max(model_choose3)

## [1] 1
```

Error Measure	Best Subset Selection	Forward Stepwise Selection
CP	11.09	14.56
Adj. R-squared	0.3181	0.3180
BIC	-2730.38	-2712.851

We can notice how the Best Subset Selection works better when we consider all the three measures that we are taking into account. The latter delivers the lowest value for Mallow's Cp and BIC while the highest value for the adjusted R-squared. However, we have to be cautious when we decide to adopt the Best Selection Model, given that BSS tends to overfit also when we are employing the test set errors. For this reason, we would probably prefer to adopt FSS that despite giving us less accurate results, it will avoid potential overfitting problems.

1.1.4 Resampling methods

In the previous section, we have adopted some analytical correction to obtain a reliable (and faster) measure of the test set error. We will now instead turn to the estimation of the test set error by employing cross-validation on the best model that we have retained from the previous analysis (Forward Stepwise Selection). The main goal of this subsection is two-folded. From one side, we want to understand whether the two measures of the test set error are similar. On the other side, we want to gain some reliable estimation of the Rooted Means squared error (RMSE) to make the models that we will use comparable. To use this procedure, we would need to download the 'caret' package that will enable us to divide our sample between training and test set.

```
#FSS
pGrid=expand.grid(nvmax=seq(from=1, to=predictors, by=1))
set.seed(5)
fitbd <- trainControl(method = "repeatedcv")
set.seed(5)
reg.fitfwd1 <- train(popularity ~ ., data =data_final,
```

```
                  method = "leapForward", trControl = fitbd, tuneGrid = pGrid)
reg.fitfwd1
names(reg.fitfwd1)

#head(reg.fitfwd1$results)
mincv1=which.max(reg.fitfwd1$results$Rsquared)
mincv1 #number of variables associated to highest R-squared

## [1] 20

mincv2=max(reg.fitfwd1$results$Rsquared)
mincv2 #highest R-squared for Cross Validation

## [1] 0.3161762

imaxf #number of variables associated to highest R-sq in previous exercise

## [1] 20

maxf1 #highest R-squared from the previous exercise

## [1] 0.3180517

mincv3=which.min(reg.fitfwd1$results$RMSE)
mincv3#Number associated to a Lower RMSE

## [1] 20

mincv4=min(reg.fitfwd1$results$RMSE)
mincv4#Lower RMSE

## [1] 16.9767

mse.fss = mincv4^2
```

When we examine the results of the Cross-Validation with Forward Stepwise Selection we can notice that the model suggest considering 20 variables that correspond to an adjusted R-squared equal to 0.3162. If we compare these results with what we have obtained by employing the "adjusted" test error, we can notice that the latter suggested us the same number of variables and delivered a higher level of adjusted R-squared (0.3180). The two techniques for taking into account test error are thus not significantly different. Finally, if we consider the model with the lowest RMSE, the model with 20 variables provides the lowest value of this error measure (=16.98).

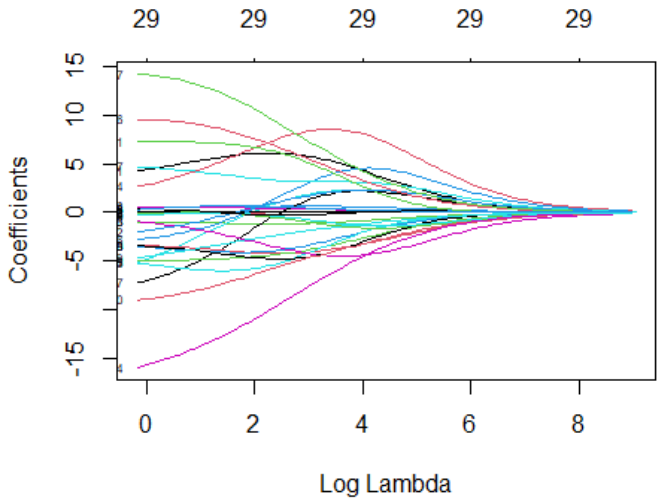
1.2. Shrinkage methods

We present some results based on ridge regression and the lasso in order to predict **Popularity** on the *Spotify* dataset using the `glmnet` package.

1.2.1. Ridge Regression

Given the nature of the `glmnet` package, in order to let it work properly, we need to supply a matrix of predictors, x, and a vector of outcomes y, as follows [1]

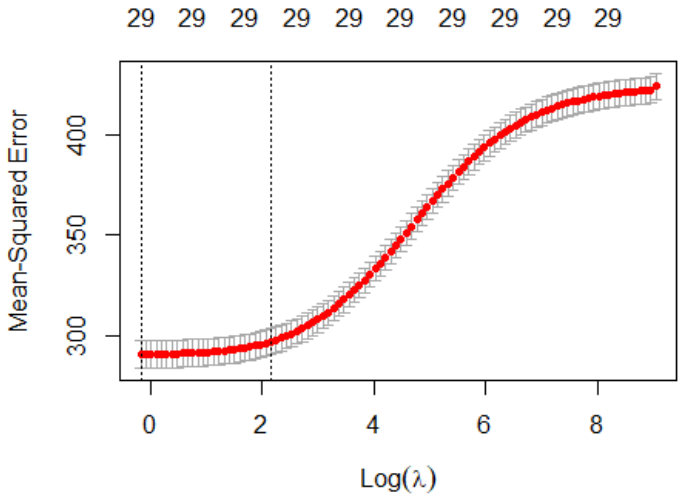
```
x = model.matrix(popularity ~ . -1, data = data_final)
y = data_final$popularity

# Training and Test Set
set.seed(1)
train.index = sample(n, round(n/2), replace = FALSE)
x.train = x[train.index, ]
y.train = y[train.index]
x.test = x[-train.index, ]
y.test = y[-train.index]
data_final.train = data_final[train.index, ]
data_final.test = data_final[-train.index, ]

# Ridge Regression Model
fit.ridge = glmnet(x.train, y.train, alpha = 0)
plot(fit.ridge, xvar = "lambda", label = TRUE)
```



The plot above shows how the parameters estimates evolve as a function of (log)λ. Unfortunately it shows only the index of the variables, not the name. The degrees of freedom are always equal to 29 because the models are *dense*. We can identify the optimal λ and the subsequent coefficients using Cross-Validation.

[1] We dropped the constant from x.

```
# 10-fold CV on Lambda
cv.ridge = cv.glmnet(x.train, y.train, alpha = 0)
plot(cv.ridge)
```




```
#coef(cv.ridge)
```

No parameter is equal to zero showing once again the *density* of the model. The (log)λ associated to the lowest value of the MSE has a value near to zero, corresponding to the more constrained model, therefore shrinking the estimated coefficients effectively reduces the variance at the cost of a negligible increase in bias. This can lead to substantial improvements in the accuracy with which we can predict the response for observations not used in model training, making the Ridge Regression preferred also to the Least Squares.

Given that the optimal λ tends to overfit a bit, we can use the best model according to the *one-S.E.* rule to predict on the test set and compute the MSE.

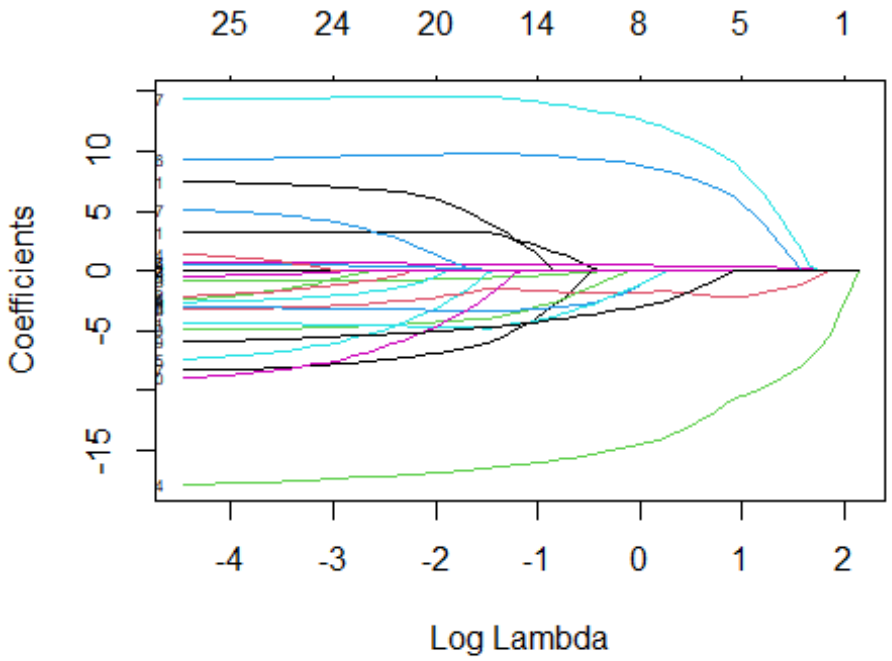
```
pred.ridge = predict(cv.ridge, x.test, s = cv.ridge$lambda.1se)
mse.rr = mean((pred.ridge-y.test)^2)
mse.rr
```

```
## [1] 292.8461
```

1.2.2. LASSO

We now turn our interest in applying the LASSO regression method to our data.

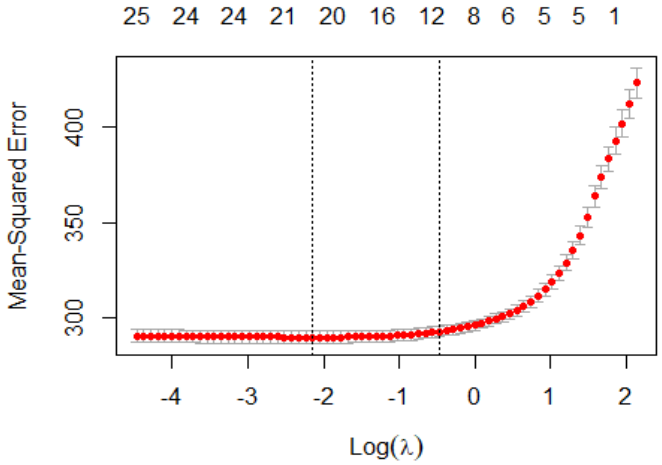
```
# LASSO
fit.lasso = glmnet(x.train, y.train)
plot(fit.lasso, xvar = "lambda", label = TRUE)
```




```
fit.lasso
```

The plot easily shows how the parameters start getting away from zero one at a time as λ decreases. Again we apply Cross Validation to estimate the optimal λ.

```
#Cross-Validation
set.seed(1)
cv.lasso = cv.glmnet(x.train, y.train)
plot(cv.lasso)
```



```
#coef(cv.lasso)

# Choice of the best model
i.best = which(fit.lasso$lambda == cv.lasso$lambda.1se)
i.best

## [1] 29

# Nonzero parameters
fit.lasso$df[i.best]

## [1] 11
```

The advantage of the LASSO with respect to Ridge Regression is that it offers insights also on *variable selection*. This can be seen also on the plot where the second dotted vertical line shows that from the initial 29 coefficients only 11 remains relevant.

Ultimately the test MSE can be predicted

```
# Prediction (test MSE)
pred.lasso = predict(fit.lasso, x.test, s = cv.lasso$lambda.1se)
mse.lasso = mean((pred.lasso-y.test)^2)
mse.lasso

## [1] 292.8998
```

The result is slightly above the one obtained with Ridge Regression underlining that the latter performs slightly better than LASSO. This is probably due to the fact that we are in a *dense* dataset rather than a *sparse* one otherwise the more remarkable performances of the LASSO would have emerged.

1.3. Tree-based methods

In this section, we are going to apply **tree-based methods** in a regression model, aiming to predict our outcome variable '*popularity*'. The underlying idea of tree-based methods is to split the predictor space into some distinct and non-overlapping regions. To do so, we proceed by splitting the sample using a random 50%/50% split. Once we have this partition, the value of popularity can be predicted as the localized average of the observed outcomes in each region.

```
set.seed(1)
n = dim(Spotify)[1]
train = sample(1:n, n/2)
```

1.3.1. Pruned tree

The starting values of this algorithm are the sample average and the residual sum of squares. Then, we select one predictor and one threshold to split the predictor space, using the criterion of RSS minimization. The previous step is repeated until a stopping criterion ('mindev=0.001') is defined.

```
tree.popularity = tree(popularity ~ ., data = Spotify,
                       control = tree.control(n/2, mindev = 0.001), subset =
train.index)
summary(tree.popularity)
```

Next, we predict the MSE of the unpruned tree on the test set.

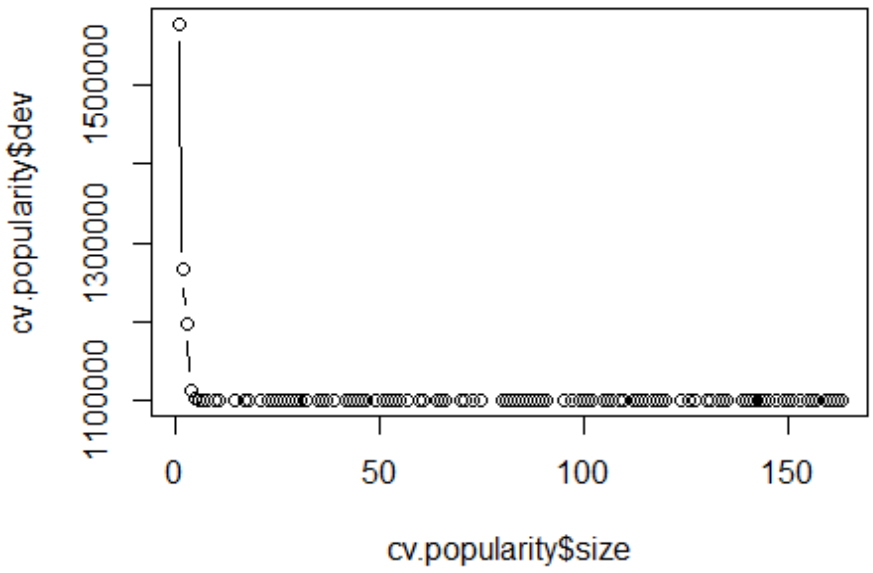
```
tree.popularity.pred = predict(tree.popularity, Spotify[-train.index,])
mse.tree = mean((Spotify$popularity[-train.index]-tree.popularity.pred)^2)
mse.tree
```

```
## [1] 359.7143
```

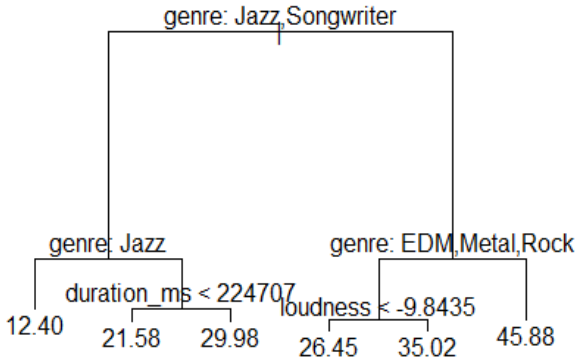
The Mean Squared Error for the test set is equal to 359.71. We can store it to later compare it to the other results.

The Recursive Binary Splitting can go ahead until each region reaches only one observation but this would cause overfitting problems. At the same time, stopping the procedure too early would prevent us from obtaining the most considerable reduction in the error. Cost Complexity Pruning is beneficial at this point. This approach consists of growing a very large tree and pruning it back to get a subtree.This procedure is equivalent to apply to RR Regularized Criterion a penalty multiplying a measure of flexibility of the model. The flexibility of the model is represented by the number of leaves while the penalty by the component α, which modulates the trade-off between complexity and fit to training data.

```
cv.popularity = cv.tree(tree.popularity)
```



Looking at the graph, it is clear that 6 is the optimal number for cross validation, that is, for the identification of the optimal α. It is important to recall that for each α, there exists a unique subtree. For size equal to 6, deviance stops descreasing. Let us pick this subtree and plot it



Finally, we have a clear plot of the pruned tree. From the bottom to the top, the number of leaves is 6, and we can see that jazz improves a lot in the reduction of the error.

Let us check whether the subtree works as well as the full tree on the test dataset. To do so, we also compute the MSE on the test set.

```
prune.popularity.pred = predict(prune.popularity, Spotify[-train.index,])
mse.prune = mean((Spotify$popularity[-train.index]-prune.popularity.pred)^2)
mse.prune
```

```
## [1] 296.4633
```

In order to check for the fit of the subtree on the test set, we can compute the MSE. The latter is equal to 296.46 . 359.71 was instead computed for the original tree. By computing the square roots for both values, we obtain about 17 and 19, respectively. The pruned tree has improved the prediction ability of popularity.

1.3.2. Bagging and Random Forests

Bootstrap Aggregating aims to reduce the variance of a statistical method by averaging the results of separate sub-trees.The outcome consists in the estimation of a parameter which is typically much less noisy than the estimate provided by a single tree.

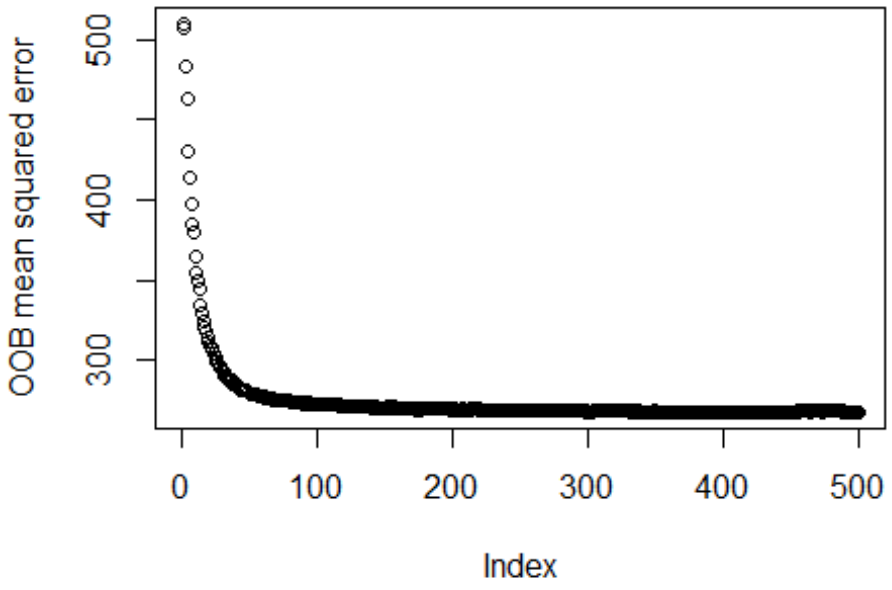
One of the advanced methods mentioned so far is the Random Forest approach. The presence of correlation in the bootstrapped samples reflects in a less effective variance

reduction. Thus, the underlying idea of Random Forests consists in introducing randomness such that trees become uncorrelated, and the variance reduction can be more effective. Applying this strategy, at each step of the tree building procedure we randomly draw a subset m of predictors.

```
train.Spotify = Spotify[train.index,]
test.Spotify = Spotify[-train.index,]
```

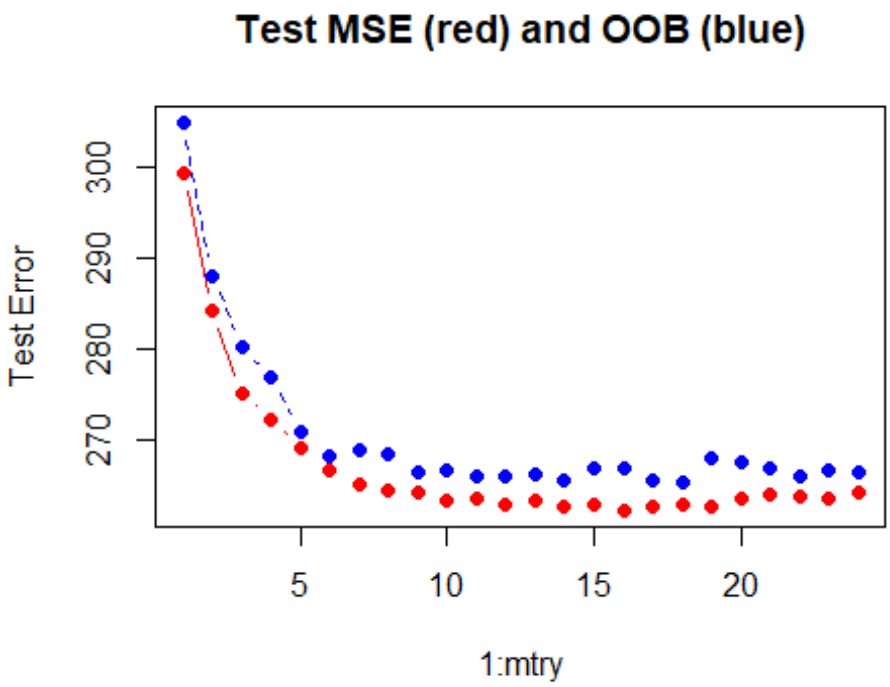
Let us now fit the Random Forest approach. The default value for *mtry* in regression is $p/24$.

```
rf.Spotify = randomForest(popularity ~ .,
                          data = train.Spotify)
```



The number of variables that can be randomly selected at each split is a tunning parameter. To evaluate it optimal value, we apply a loop that reports the OOB MSE and the test set MSE for each value of *mtry*.

```
## 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21
22  23  24
```



Random Forest

It is unclear which is the optimal value for *mtry*, but probably around 12. Next, we compute the test MSE for this RF specification.

```
rf.Spotify = randomForest(popularity ~ .,
                          data = train.Spotify,
                          mtry = 12, ntree = 500)
rf.pred = predict(rf.Spotify, test.Spotify, type = "response")
mse.rf = mean((test.Spotify$popularity-rf.pred)^2)
mse.rf

## [1] 262.6376
```

We obtain a MSE on the test set of 262.64.

Bagging

Finally, we compute the MSE for the test set for *metry* equal to 0 and store it in a vriable to compare later.

```
bagg.Spotify = randomForest(popularity ~ .,
                          data = train.Spotify,
                          mtry = 24, ntree = 500)
bagg.pred = predict(bagg.Spotify, test.Spotify, type = "response")
#rmse(test.Spotify$popularity, bagg.pred)
mse.bagg = mean((test.Spotify$popularity-bagg.pred)^2)
```

By comparing the value of the MSE obtained in bagging, which is equal to 263.97, we can see that it provides a better performance than unpruned and pruned trees, although interpretability gets worse.

1.4. Support Vector Regression

Support Vector Regression (SVR) works on similar principles as Support Vector Machine (SVM) classification. In particular, it is the adapted form of SVM when the dependent variable is numerical rather than categorical. SVR is a non-parametric technique that allows high flexibility in terms of distribution of underlying variables, relationship between independent and dependent variables and the control on the penalty term.

1.4.1. SVR

In order to carry out our analysis, the first step is to divide our database in a training and test set by employing the caTools package.

```
#Training set and test set
set.seed(123)
split = sample.split(Spotify$popularity, SplitRatio = 0.75)
training_set = Spotify[train.index,]
test_set = Spotify[-train.index,]
```

First, we have tried to include a support vector predictor with linear Kernel. The funcion tune.svm allows us to analyse different combinations of gamma and cost parameters that reshape the flexibility of the model. We have tunned sequantially in order to reduce the computational power required.

```
set.seed(12)
tuning <- tune.svm(popularity ~ ., data = training_set, kernel='linear',
gamma = 10^(-5:-3), cost = 10^(-3:0))
bestmod = tuning$best.model
summary(bestmod)
```

```
##
## Call:
## best.svm(x = popularity ~ ., data = training_set, gamma = 10^(-5:-3),
##     cost = 10^(-3:0), kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  eps-regression
##  SVM-Kernel:  linear
##        cost:  0.1
##       gamma:  1e-05
##     epsilon:  0.1
##
##
## Number of Support Vectors:  3293
```

```
#Compute MSE on test set for linear kernel
y_pred_lin = predict(bestmod, newdata = test_set)
mse.svr.lin = mean((test.Spotify$popularity-y_pred_lin)^2)
mse.svr.lin
```

```
## [1] 297.3487
```

The setting with gamma=0.0001 and cost=1 seems to be the best for our model. We can now proceed with implementing different types of kernels with the use of the same parameters arising from the tune.svm function.

```
#Radial kernel
predictor_rd=svm(formula = popularity ~ .,
                 data = training_set,
                 type = 'eps-regression',
                 kernel = 'radial',gamma=0.0001, cost=1)
#Sigmoid kernel
predictor_sgm=svm(formula = popularity ~ .,
                 data = training_set,
                 type = 'eps-regression',
                kernel = 'sigmoid',gamma=0.0001, cost=1)
#Compute MSE on test set for radial and sigmoid kernel
y_pred_rd = predict(predictor_rd, newdata = test_set)
mse.svr.rd = mean((test_set$popularity-y_pred_rd)^2)
mse.svr.rd
```

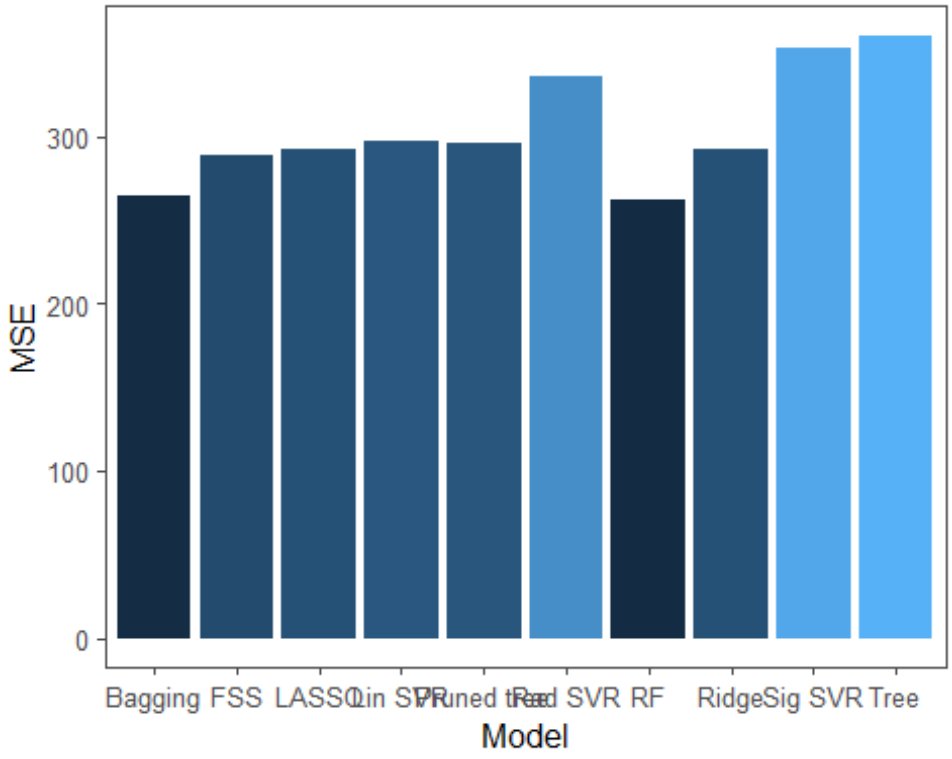
```
## [1] 336.1029
```

```
y_pred_sgm = predict(predictor_sgm, newdata = test_set)
mse.svr.sgm = mean((test_set$popularity-y_pred_sgm)^2)
mse.svr.sgm
```

```
## [1] 352.9418
```

We can observe how by incorporating more flexible kernels the MSE on the test set decreases. It would be interesting to try tunning *gamma* and *cost* with different kernels. Next, we compare the performance on the test MSE of different models.

1.5. Conclusion *popularity*



In the table above, we can observe the MSE for the different models.

2. Predicting *genre*

In the following lines, we explored the performance of a variety of multi-class ML classification methods in order to predict the genre of a song. This task could become relevant in a number of scenarios, such becoming part of the recommendation algorithm of Spotify in order to show same-genre songs or for internal classification of the new songs that are added to Spotify every day (they sum up to over 1 million per month).

As predicting variables, we included those provided by Spotify regarding audio features, those extracted from the basic sentiment analysis regarding lyrics and popularity.

```
#Here we perform the train/test split we will use in predicting genre
#It is a balanced split in the genre variable
# Training and Test Set
set.seed(1011)
train.label = createDataPartition(spotify_right_lyrics_popularity$genre,
                                  p = 2/3, list = FALSE)[,1]
```

2.1. Subset Selection

In this first section, we do Forward Subset Selection and Backward Subset Selection using LDA models. We selected LDA instead of Logistic Regression because the discriminant analysis is more prevalent when there are more than 2 response classes.

```
#Prepare variables, test and train
spotify_data_std <- spotify_right_lyrics_popularity %>%
  mutate_at(vars(-genre), scale)
train.spotify_data_std = spotify_data_std[train.label,]
test.spotify_data_std = spotify_data_std[-train.label,]
```

2.1.1. Forward Subset Selection

```
# FSS based on classification rate
forward.lda <- stepclass(train.spotify_data_std,
                         grouping =
                           train.spotify_data_std$genre,
                         method = "lda", improvement = 0.005,
                         direction ="forward", criterion = "CR",
                         fold = 10, output = TRUE)

## correctness rate: 0.38503;  in: "acousticness";  variables (1):
acousticness
## correctness rate: 0.4142;  in: "speechiness";  variables (2):
acousticness, speechiness
## correctness rate: 0.44438;  in: "danceability";  variables (3):
acousticness, speechiness, danceability
## correctness rate: 0.46429;  in: "loudness";  variables (4): acousticness,
speechiness, danceability, loudness
## correctness rate: 0.4824;  in: "valence";  variables (5): acousticness,
speechiness, danceability, loudness, valence
## correctness rate: 0.49729;  in: "energy";  variables (6): acousticness,
speechiness, danceability, loudness, valence, energy
## correctness rate: 0.50916;  in: "mode";  variables (7): acousticness,
speechiness, danceability, loudness, valence, energy, mode
## correctness rate: 0.51801;  in: "positive";  variables (8): acousticness,
speechiness, danceability, loudness, valence, energy, mode, positive
## correctness rate: 0.52566;  in: "liveness";  variables (9): acousticness,
speechiness, danceability, loudness, valence, energy, mode, positive,
liveness
## correctness rate: 0.53068;  in: "disgust";  variables (10): acousticness,
speechiness, danceability, loudness, valence, energy, mode, positive,
liveness, disgust
##
##  hr.elapsed min.elapsed sec.elapsed
##        0.00        0.00       26.96

formula <- forward.lda[9][[1]]

# Compute best LDA fit according to FSS CR using 10-fold-CV
```

```
lda.forward.bestfit <- lda(formula = formula,
                           data = train.spotify_data_std)
# Predict genre on test set
lda.forward.pred <- predict(lda.forward.bestfit,
                            newdata = test.spotify_data_std)
# Confussion matrix
cm.fss <- table(true = test.spotify_data_std$genre,
      predict = lda.forward.pred$class)

# Performance of the model
class.perf.fss <- NULL
acc.perf.fss <- NULL
err.perf.fss <- NULL
(class.perf.fss <- cm.performance(true = test.spotify_data_std$genre,
               predicted = lda.forward.pred$class))

##            balanced.accuracy       FNR        FPR precision specificity
## EDM                0.5359559 0.9134615 0.01462664 0.3214286   0.9853734
## Jazz               0.7427893 0.3059867 0.20843471 0.5491228   0.7915653
## Metal              0.8384765 0.2061856 0.11686144 0.6226415   0.8831386
## Pop                0.6836884 0.4542936 0.17832957 0.4539171   0.8216704
## Rap                0.7013046 0.5320000 0.06539075 0.5879397   0.9346093
## Rock               0.5870153 0.6683805 0.15758896 0.3728324   0.8424110
## Songwriter         0.6329328 0.5393082 0.19482619 0.5486891   0.8051738

(acc.perf.fss <- cm.performance(true = test.spotify_data_std$genre,
               predicted = lda.forward.pred$class, metrics = 2))

## [1] 0.5193392

(err.perf.fss <- cm.performance(true = test.spotify_data_std$genre,
               predicted = lda.forward.pred$class, metrics = 3))

## [1] 0.4806608
```

We can observe that FSS had an accuracy of 0.52 and and a classification error rate of 0.48.

2.1.2. Backward Subset Selection

```
# BSS based on classification rate
backward.lda <- stepclass(train.spotify_data_std,
                          grouping =
                            train.spotify_data_std$genre,
                          method = "lda", improvement = 0.005,
                          direction ="backward", criterion = "CR",
                          fold = 10, output = TRUE)

## correctness rate: 0;  starting variables (25): danceability, key, mode,
time_signature, energy, loudness, speechiness, acousticness,
instrumentalness, liveness, valence, tempo, duration_ms, anger, anticipation,
disgust, fear, joy, sadness, surprise, trust, negative, positive, popularity,
genre
## correctness rate: 0.53852;  out: "genre";  variables (24): danceability,
```

```
key, mode, time_signature, energy, loudness, speechiness, acousticness,
instrumentalness, liveness, valence, tempo, duration_ms, anger, anticipation,
disgust, fear, joy, sadness, surprise, trust, negative, positive, popularity
## correctness rate: 0.54174;  out: "instrumentalness";  variables (23):
danceability, key, mode, time_signature, energy, loudness, speechiness,
acousticness, liveness, valence, tempo, duration_ms, anger, anticipation,
disgust, fear, joy, sadness, surprise, trust, negative, positive, popularity
## correctness rate: 0.54355;  out: "anticipation";  variables (22):
danceability, key, mode, time_signature, energy, loudness, speechiness,
acousticness, liveness, valence, tempo, duration_ms, anger, disgust, fear,
joy, sadness, surprise, trust, negative, positive, popularity
## correctness rate: 0.54516;  out: "fear";  variables (21): danceability,
key, mode, time_signature, energy, loudness, speechiness, acousticness,
liveness, valence, tempo, duration_ms, anger, disgust, joy, sadness,
surprise, trust, negative, positive, popularity
## correctness rate: 0.54717;  out: "negative";  variables (20):
danceability, key, mode, time_signature, energy, loudness, speechiness,
acousticness, liveness, valence, tempo, duration_ms, anger, disgust, joy,
sadness, surprise, trust, positive, popularity
## correctness rate: 0.54798;  out: "key";  variables (19): danceability,
mode, time_signature, energy, loudness, speechiness, acousticness, liveness,
valence, tempo, duration_ms, anger, disgust, joy, sadness, surprise, trust,
positive, popularity
##
##  hr.elapsed min.elapsed sec.elapsed
##        0.00        0.00       32.73

formula <- backward.lda[9][[1]]

# Compute best LDA fit according to BSS CR using 10-fold-CV
lda.backward.bestfit <- lda(formula = formula,
                            data = train.spotify_data_std)
# Predict genre on test set
lda.backward.pred <- predict(lda.backward.bestfit,
                             newdata = test.spotify_data_std)
# Confussion matrix
cm.bss <- table(true = test.spotify_data_std$genre,
      predict = lda.backward.pred$class)
# Performance of the model
class.perf.bss <- NULL
acc.perf.bss <- NULL
err.perf.bss <- NULL
(class.perf.bss <- cm.performance(true = test.spotify_data_std$genre,
               predicted = lda.backward.pred$class))

##            balanced.accuracy       FNR        FPR precision specificity
## EDM                0.5534910 0.8750000 0.01801802 0.3513514   0.9819820
## Jazz               0.7804980 0.2638581 0.17514595 0.6125461   0.8248540
## Metal              0.8422884 0.2164948 0.09892828 0.6551724   0.9010717
## Pop                0.6787658 0.4709141 0.17155425 0.4494118   0.8284457
```

```
## Rap                0.6977904 0.5320000 0.07241911 0.5545024   0.9275809
## Rock               0.5860436 0.6735219 0.15439093 0.3681159   0.8456091
## Songwriter         0.6432323 0.5078616 0.20567376 0.5452962   0.7943262

(acc.perf.bss <- cm.performance(true = test.spotify_data_std$genre,
               predicted = lda.backward.pred$class, metrics = 2))

## [1] 0.5322321

(err.perf.bss <- cm.performance(true = test.spotify_data_std$genre,
               predicted = lda.backward.pred$class, metrics = 3))

## [1] 0.4677679
```

We can observe that BSS had an accuracy of 0.53 and and a classification error rate of 0.47. In comparison to FSS, it had slightly better accuracy and lower error, as a tradeoff for including 19 variables instead of 10. Selecting a lower minimum improvement in each iteration would have helped them to achieve beter convergence.

2.2 Shrinkahe Methods

Our focus now is to apply the same regularization methods employed on the previous model with **Popularity** as the dependent variable. We have to take into account, however, that in a multiple-output linear model, the least-squares estimates are simply the individual least squares estimates for each of the outputs. Hence, to apply selection and shrinkage methods in the multiple output case, one could apply a univariate technique individually to each outcome or simultaneously to all results. We choose the latter because it would permit all k outputs to be used in estimating the sole regularization parameter λ.

2.2.1. Ridge Regression

The (log)λ associated to the lowest value of the MSE has a value far from zero, corresponding to the unconstrained model, therefore shrinking the estimated coefficients is not effective in reducing the variance, making the Ridge Regression a poor alternative to the Least Squares. The balanced accuracy gives however reliable results for the predictive ability of the model.

```
# Ridge Regression
x = model.matrix(genre ~ . -1, data = spotify_right_lyrics_popularity)
y = spotify_right_lyrics_popularity$genre

# Training and Test Set
x.train = x[train.label, ]
y.train = y[train.label]
x.test = x[-train.label, ]
y.test = y[-train.label]
spotify_right_lyrics_popularity.train =
spotify_right_lyrics_popularity[train.label, ]
spotify_right_lyrics_popularity.test = spotify_right_lyrics_popularity[-
train.label, ]
```

```r
# Ridge Regression Model
fit.ridge = glmnet(x.train, y.train, family = "multinomial", alpha = 0)
#plot(fit.ridge, xvar = "lambda", label = TRUE)

# 10-fold CV on Lambda
cv.ridge = cv.glmnet(x.train, y.train, family = "multinomial", alpha = 0)
#plot(cv.ridge)
#coef(cv.ridge)

# Prediction
pred.ridge = predict(cv.ridge, x.test, type ="class", s =
cv.ridge$lambda.1se)
table(true = y.test, prediction = pred.ridge)
```

```
##             prediction
## true         EDM Jazz Metal Pop Rap Rock Songwriter
##   EDM          8    2    22  44  10    8         10
##   Jazz         0  321     2   7   1   17        103
##   Metal        0    4   226  20   4   22         15
##   Pop          2    5    21 205  56   28         44
##   Rap          1    4     7 120  96   12         10
##   Rock         0   24    47  38   3   91        186
##   Songwriter   1  134    24  15   4   54        404
```

```r
# class labels predicted by the classifier model
predicted_labels <- pred.ridge

# true labels (test set)
true_labels <- y.test

classifier_metrics <- ml_test(predicted_labels, true_labels, output.as.table
= FALSE)

# overall classification accuracy
(acc.rr <- classifier_metrics$accuracy)
```

```
## [1] 0.5443191
```

```r
# F1-measures for classes "cat", "dog" and "rat"
(err.rr <- classifier_metrics$error.rate)
```

```
## [1] 0.4556809
```

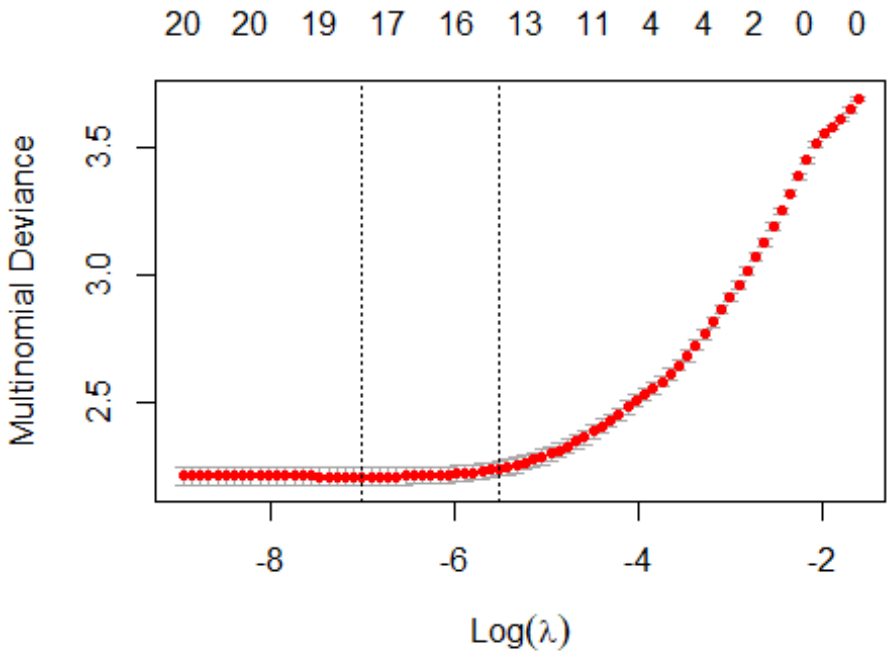
```r
# tabular view of the metrics (except for 'accuracy' and 'error.rate')
classifier_metrics <- ml_test(predicted_labels, true_labels, output.as.table
= T)[,c(1,7,9,18,20)]
classifier_metrics
```

```
##            balanced.accuracy       FNR         FPR precision specificity
## EDM                0.5369768 0.9230769 0.002969562 0.6666667   0.9970304
## Jazz               0.7839723 0.2882483 0.143807149 0.6497976   0.8561929
## Metal              0.8390373 0.2233677 0.098557692 0.6475645   0.9014423
```

```
## Pop                0.6961637 0.4321330 0.175539568 0.4565702   0.8244604
## Rap                0.6627427 0.6160000 0.058514629 0.5517241   0.9414854
## Rock               0.5666454 0.7660668 0.100642398 0.3922414   0.8993576
## Songwriter         0.6776861 0.3647799 0.279847909 0.5233161   0.7201521
```

2.2.2. LASSO

```
#LASSO
fit.lasso = glmnet(x.train, y.train, family = "multinomial")
#plot(fit.lasso, xvar = "lambda", label = TRUE)

#Cross-Validation
set.seed(1)
cv.lasso = cv.glmnet(x.train, y.train, family = "multinomial")
#coef(cv.lasso)
```




```
# Choice of the best model
i.best = which(fit.lasso$lambda == cv.lasso$lambda.1se)
i.best

## [1] 43

# Nonzero parameters
fit.lasso$df[i.best]

## [1] 24
```

The inefficiency of the Shrinkage Methods is once again shown for the best model that shows no variable selection given that the nonzero parameters remain the same of the

original model. Nevertheless the predictability of the method is still relevant given its well-behaved levels of accuracy and error rate, as shown below.

```
#Prediction (test MSE)
pred.lasso = predict(fit.lasso, x.test, type ="class", s =
cv.lasso$lambda.1se)
table(true = y.test, prediction = pred.ridge)
```

```
##             prediction
## true         EDM Jazz Metal Pop Rap Rock Songwriter
##   EDM          8    2    22  44  10    8         10
##   Jazz         0  321     2   7   1   17        103
##   Metal        0    4   226  20   4   22         15
##   Pop          2    5    21 205  56   28         44
##   Rap          1    4     7 120  96   12         10
##   Rock         0   24    47  38   3   91        186
##   Songwriter   1  134    24  15   4   54        404
```

```
# class labels predicted by the classifier model
predicted_labels <- pred.ridge

# true labels (test set)
true_labels <- y.test

classifier_metrics <- ml_test(predicted_labels, true_labels, output.as.table
= FALSE)

# overall classification accuracy
acc.lasso <- classifier_metrics$accuracy
acc.lasso
```

```
## [1] 0.5443191
```

```
# F1-measures for classes "cat", "dog" and "rat"
err.lasso <- classifier_metrics$error.rate
err.lasso
```

```
## [1] 0.4556809
```

```
# tabular view of the metrics (except for 'accuracy' and 'error.rate')
classifier_metrics <- ml_test(predicted_labels, true_labels, output.as.table
= T)[,c(1,7,9,18,20)]
classifier_metrics
```

```
##            balanced.accuracy       FNR         FPR precision specificity
## EDM                0.5369768 0.9230769 0.002969562 0.6666667   0.9970304
## Jazz               0.7839723 0.2882483 0.143807149 0.6497976   0.8561929
## Metal              0.8390373 0.2233677 0.098557692 0.6475645   0.9014423
## Pop                0.6961637 0.4321330 0.175539568 0.4565702   0.8244604
## Rap                0.6627427 0.6160000 0.058514629 0.5517241   0.9414854
```

```
## Rock               0.5666454 0.7660668 0.100642398 0.3922414   0.8993576
## Songwriter         0.6776861 0.3647799 0.279847909 0.5233161   0.7201521
```

2.3. Tree-based methods

In this section, we explored the predicting ability of tree based methods. To be precise, we compared a single pruned tree, bagging, random forest and boosting.

```
#Prepare train and test set
train.spotify_right_lyrics_popularity =
spotify_right_lyrics_popularity[train.label,]
test.spotify_right_lyrics_popularity = spotify_right_lyrics_popularity[-
train.label,]
```

2.3.1. Pruned tree

```
# Grow tree
tree.genre <- tree(genre ~ ., data = train.spotify_right_lyrics_popularity,
                   control = tree.control(nobs =
nrow(spotify_right_lyrics_popularity), mindev = 0.001),
                   subset = train.label)

# Compute predictions on test set of unpruned tree
tree.genre.pred <- predict(tree.genre, test.spotify_right_lyrics_popularity,
type = "class")
with(test.spotify_right_lyrics_popularity, table(true = genre, predict =
tree.genre.pred))
```

```
##             predict
## true         EDM Jazz Metal Pop Rap Rock Songwriter
##   EDM         22    4    15  26  12   11         14
##   Jazz         4  323     2  21   5   13         83
##   Metal        2    6   193  24   6   26         34
##   Pop         15   11    18 162  76   41         38
##   Rap          1    4     6  82 135   13          9
##   Rock        11   26    30  34  25  107        156
##   Songwriter   9  115    35  26   6   65        380
```

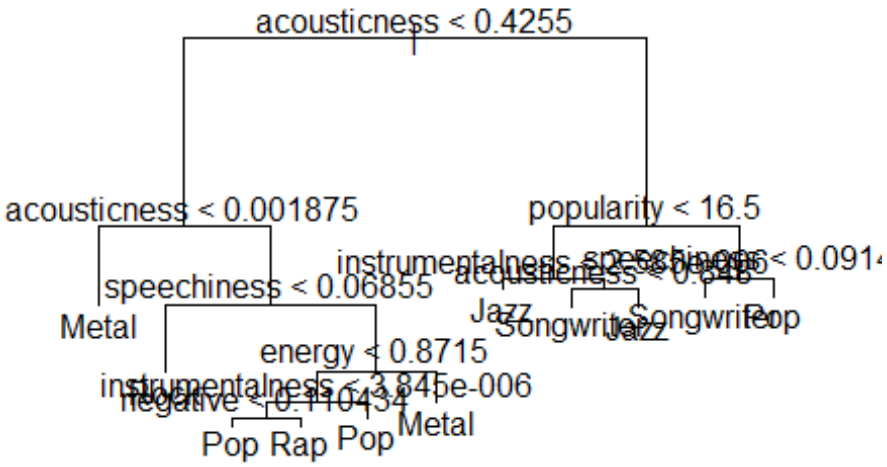
```
err.perf.tree <- NULL
acc.perf.tree <- NULL
(err.perf.tree <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
                predicted = tree.genre.pred, metrics = 3))
```

```
## [1] 0.467365
```

```
(acc.perf.tree <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
                predicted = tree.genre.pred, metrics = 2))
```

```
## [1] 0.532635
```

```
# Prune tree
cv.genre_class = cv.tree(tree.genre, FUN = prune.misclass)
prune.genre_class = prune.misclass(tree.genre, best = 11)
```




```
# Evaluate pruned tree on test data
tree.pruned.genre.pred = predict(prune.genre_class,
test.spotify_right_lyrics_popularity, type = "class")
with(test.spotify_right_lyrics_popularity, table(true = genre, predict =
tree.pruned.genre.pred))

##             predict
## true         EDM Jazz Metal Pop Rap Rock Songwriter
##   EDM          0    1    13  17   5   63          5
##   Jazz         0  246     2  17   5   40        141
##   Metal        0    2   213  16   1   50          9
##   Pop          0    3    30  95  91  100         42
##   Rap          0    3    14  41 151   31         10
##   Rock         0    4    41  26  11  211         96
##   Songwriter   0   50    35  25   4  171        351

class.perf.pruned.tree <- NULL
acc.perf.pruned.tree <- NULL
err.perf.pruned.tree <- NULL
(class.perf.pruned.tree <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
               predicted = tree.pruned.genre.pred))
```

```
##             balanced.accuracy       FNR        FPR precision specificity
## EDM                 0.5000000 1.0000000 0.00000000       NaN   1.0000000
## Jazz                0.7436682 0.4545455 0.05811808 0.7961165   0.9418819
## Metal               0.8092090 0.2680412 0.11354079 0.6120690   0.8864592
## Pop                 0.5775455 0.7368421 0.10806697 0.4008439   0.8919330
## Rap                 0.7545547 0.3960000 0.09489051 0.5634328   0.9051095
## Rock                0.6206457 0.4575835 0.30112508 0.3168168   0.6988749
## Songwriter          0.6516612 0.4481132 0.24856440 0.5366972   0.7514356
```

```
(acc.perf.pruned.tree <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
               predicted = tree.pruned.genre.pred, metrics = 2))
```

```
## [1] 0.5104754
```

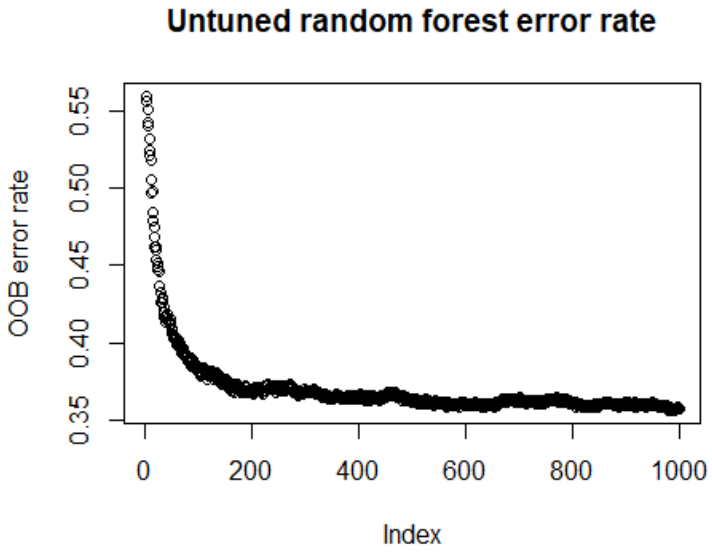
```
(err.perf.pruned.tree <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
               predicted = tree.pruned.genre.pred, metrics = 3))
```

```
## [1] 0.4895246
```

The pruned tree had a classification error on the test set of 0.49, which is slightly higher than the full grown tree. In addition, no observations predicted EDM. We expect the result to improve in the following methods due to the reduction in variance.

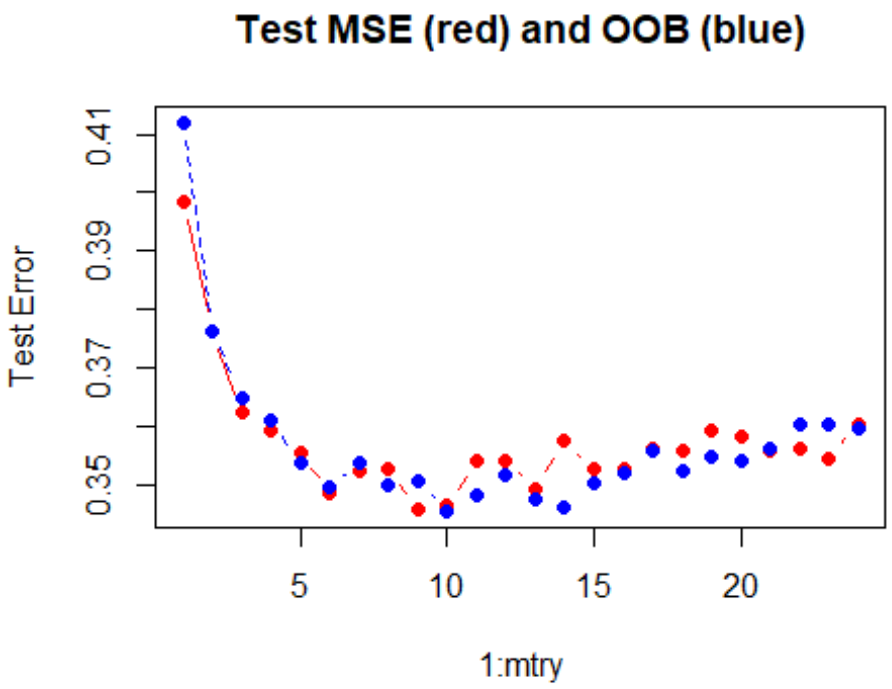
2.3.2. Bagging and Random Forest

```
#Grow RF
rf.genre <- randomForest(genre ~ .,
                         data = train.spotify_right_lyrics_popularity,
                         ntree = 1000,
                         mtry = 4) #common choice is floor(sqrt(24))
rf.genre
```



Given that *mtry* is a tuning parameter, we loop over the possible different values to see its performance. In addition, we compare the OOB error to the test set error.

```
#Estimate OOB error for every possible combination of mtry
oob.err = double(24)
test.err = double(24)
for (mtry in 1:24) {
  rf.m = randomForest(genre ~ .,
                      data = train.spotify_right_lyrics_popularity,
                      mtry = mtry, ntree = 1000)
  oob.err[mtry] = rf.m$err.rate[1000,1]
  pred = predict(rf.m, test.spotify_right_lyrics_popularity)
  test.err[mtry] = with(test.spotify_right_lyrics_popularity, mean(genre !=
pred))
  cat(mtry," ")
}
```



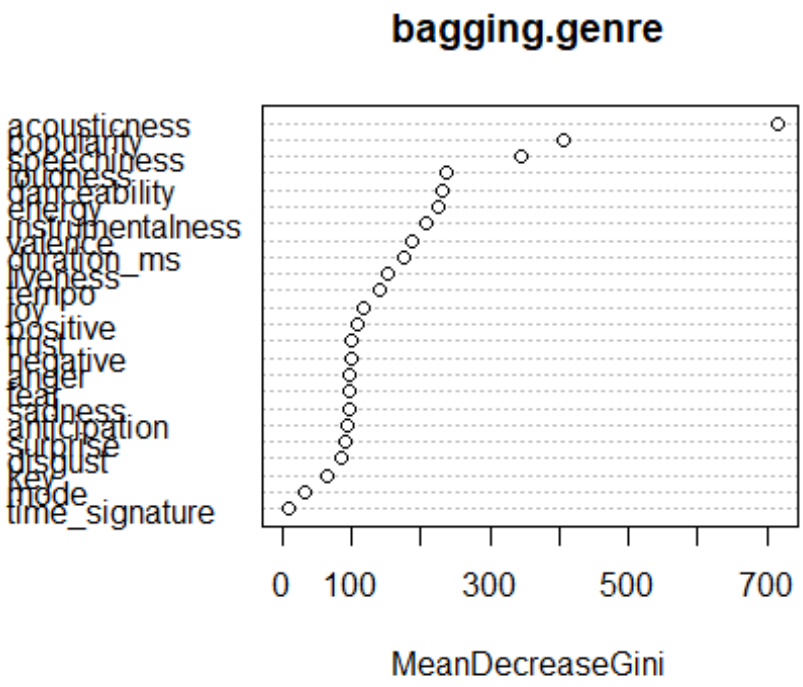
It appears like 4 was not the correct choice for *mtry*. Next, the evaluate the predicting ability of bagging and random forest with *mtry*=8.

```
# Bagging prediction
bagging.genre <- randomForest(genre ~ .,
                      data = train.spotify_right_lyrics_popularity,
```

```
                        ntree = 1000,
                        mtry = 24) #the common choice for classification is
floor(sqrt(24))
bagging.genre

##
## Call:
##  randomForest(formula = genre ~ ., data =
train.spotify_right_lyrics_popularity,      ntree = 1000, mtry = 24)
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 24
##
##         OOB estimate of  error rate: 35.89%
## Confusion matrix:
##            EDM Jazz Metal Pop Rap Rock Songwriter class.error
## EDM         53    1    26  66  11   36         16   0.7464115
## Jazz         5  685     1  20   8   25        159   0.2414175
## Metal        7    1   451  32   7   48         36   0.2250859
## Pop         16    9    29 452 124   42         52   0.3756906
## Rap          5    5     8 169 287   18         10   0.4282869
## Rock         8   28    47  79  16  294        307   0.6225931
## Songwriter   2  107    26  24   4  144        965   0.2413522

bagging.pred <- predict(bagging.genre, test.spotify_right_lyrics_popularity,
type = "class")
class.perf.bagging <- NULL
acc.perf.bagging <- NULL
err.perf.bagging <- NULL
(class.perf.bagging <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
               predicted = bagging.pred))

##            balanced.accuracy       FNR         FPR precision specificity
## EDM                0.6355818 0.7211538 0.007682458 0.7073171   0.9923175
## Jazz               0.8452774 0.2394678 0.069977427 0.7866972   0.9300226
## Metal              0.8582529 0.2371134 0.046380885 0.7708333   0.9536191
## Pop                0.7415385 0.3961219 0.120801034 0.5382716   0.8791990
## Rap                0.7846667 0.3800000 0.050666667 0.6709957   0.9493333
## Rock               0.6292264 0.6426735 0.098873592 0.4680135   0.9011264
## Songwriter         0.7621165 0.2562893 0.219477770 0.6033163   0.7805222

(acc.perf.bagging <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
               predicted = bagging.pred, metrics = 2))

## [1] 0.6361805

(err.perf.bagging <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
               predicted = bagging.pred, metrics = 3))
```

```
## [1] 0.3638195
```

We note that error rate (0.36) and accuracy (0.64) have improved very much with respect to single trees.



We observe how the variables that contribute the most for the separation of classes are mainly those associated to audio features.

```
# RF prediction
rf.genre.8 <- randomForest(genre ~ .,
                           data = train.spotify_right_lyrics_popularity,
                           ntree = 1000,
                           mtry = 8) #the common choice for classification is
floor(sqrt(24))
rf.genre.8

##
## Call:
##  randomForest(formula = genre ~ ., data =
train.spotify_right_lyrics_popularity,      ntree = 1000, mtry = 8)
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 8
##
##         OOB estimate of  error rate: 35.16%
## Confusion matrix:
##            EDM Jazz Metal Pop Rap Rock Songwriter class.error
```

```
## EDM         47    2    29  66  14   37         14   0.7751196
## Jazz         3  699     1  19   6   16        159   0.2259136
## Metal        4    3   456  34   7   42         36   0.2164948
## Pop         10    9    31 444 125   43         62   0.3867403
## Rap          4    7     7 158 306   11          9   0.3904382
## Rock         4   29    54  75  12  288        317   0.6302953
## Songwriter   1  112    31  20   2  123        983   0.2272013
```

```
random.forest.pred <- predict(rf.genre.8,
test.spotify_right_lyrics_popularity, type = "class")
class.perf.random.forest <- NULL
acc.perf.random.forest <- NULL
err.perf.random.forest <- NULL
(class.perf.random.forest <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
               predicted = random.forest.pred))
```

```
##            balanced.accuracy       FNR         FPR precision specificity
## EDM                0.6071351 0.7788462 0.006883605 0.6764706   0.9931164
## Jazz               0.8558758 0.2172949 0.070953437 0.7861915   0.9290466
## Metal              0.8826320 0.1855670 0.049168975 0.7694805   0.9508310
## Pop                0.7478346 0.3850416 0.119289340 0.5414634   0.8807107
## Rap                0.7897695 0.3720000 0.048461035 0.6796537   0.9515390
## Rock               0.6298152 0.6606684 0.079701121 0.5076923   0.9202989
## Songwriter         0.7756329 0.2358491 0.212885154 0.6151899   0.7871148
```

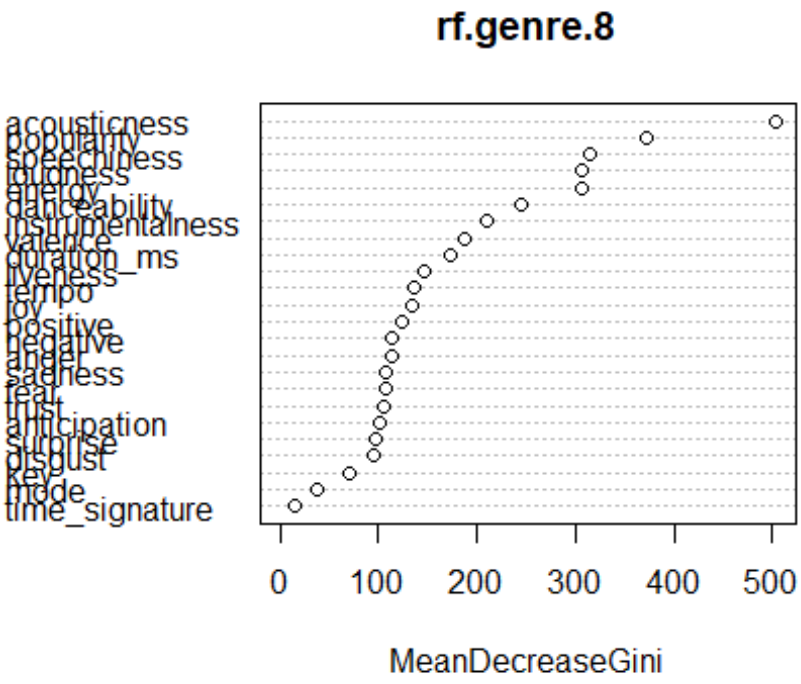
```
(acc.perf.random.forest <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
               predicted = random.forest.pred, metrics = 2))
```

```
## [1] 0.6486704
```

```
(err.perf.random.forest <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
               predicted = random.forest.pred, metrics = 3))
```

```
## [1] 0.3513296
```

We note that error rate (0.35) and accuracy (0.65) have also improved very much with respect to single trees and they are slighlty better than bagging.



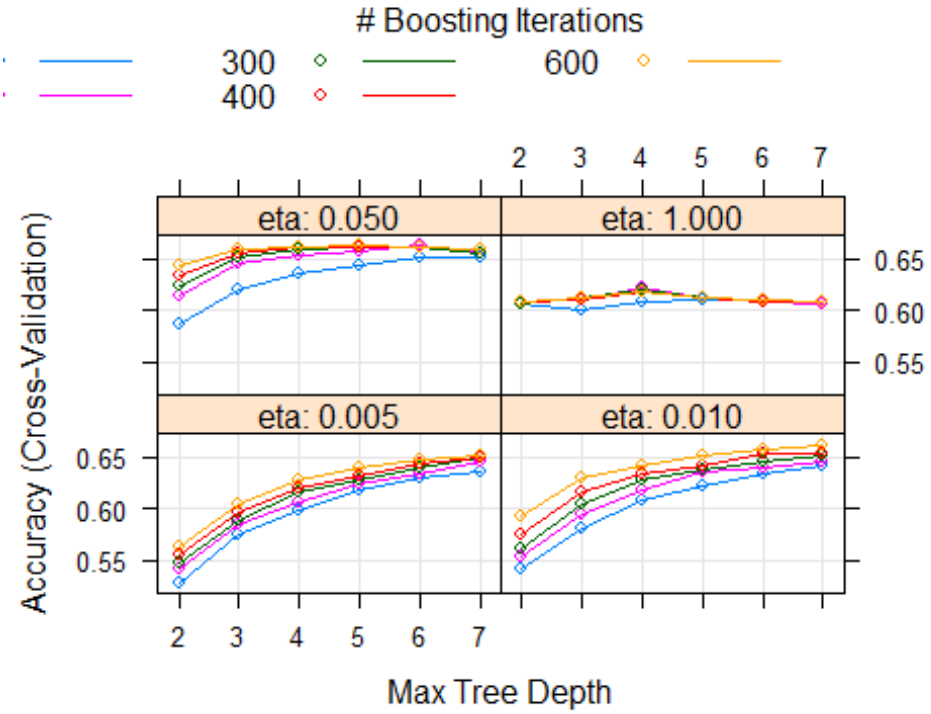
We observe how the variables that contribute the most for the separation of classes are mainly those associated to audio features.

2.3.3. Boosting

In boosting, three main parameters need to be tuned: tree depth, learning rate and the number of ensemble trees. Unfortunately, it is not possible to find the optimal values sequentially. For this reason, we create a grid of possible values for each method, and one model is computed for each unique combination. Later, we can select those parameters with the highest accuracy to make the predictions and calculate the error rate.

```
#Select possible parameters
tune_grid <- expand.grid(nrounds=c(100,200,300,400,600), #number of trees
                         max_depth = c(2:7), #depth of each tree
                         eta = c(0.005,0.01,0.05, 1), #learning rate
                         gamma = c(0.01),
                         colsample_bytree = c(0.75),
                         subsample = c(0.50),
                         min_child_weight = c(0))
#10-fold CV to compute accuracy
trctrl <- trainControl(method = "cv", number = 10)
```

We can observe what is the best model.



Fit the best boosting model according to the best parameter specification.

```
#Choose parameters
tune_grid <- expand.grid(nrounds=c(200),
                         max_depth = 6,
                         eta = c(0.05),
                         gamma = c(0.01),
                         colsample_bytree = c(0.75),
                         subsample = c(0.50),
                         min_child_weight = c(0))
#Fit best model
boost.best.fit <- train(genre ~., data =
train.spotify_right_lyrics_popularity,
                        method = "xgbTree",
                        trControl=trctrl,
                        tuneGrid = tune_grid,
                        tuneLength = 10)
#Make predictions based on test set
boost.pred <- predict(boost.best.fit,
                      test.spotify_right_lyrics_popularity %>% select(-
genre))
#Compute performance measures of the model
table(true=test.spotify_right_lyrics_popularity$genre, pred=boost.pred)

##             pred
## true         EDM Jazz Metal Pop Rap Rock Songwriter
```

```
##   EDM          32    3     9  39   5   14          2
##   Jazz          2  364     0   8   3   12         62
##   Metal         4    1   233  13   1   24         15
##   Pop           6    6     8 221  59   36         25
##   Rap           2    2     6  63 164    8          5
##   Rock          5   12    27  26   4  158        157
##   Songwriter    0   67    14  16   0   71        468
```

```
class.perf.boost <- NULL
acc.perf.boost <- NULL
err.perf.boost <- NULL
(class.perf.boost <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
                                    predicted = boost.pred))
```

```
##             balanced.accuracy       FNR        FPR precision specificity
## EDM                 0.6480072 0.6923077 0.01167793 0.6274510   0.9883221
## Jazz                0.8702631 0.1929047 0.06656913 0.8000000   0.9334309
## Metal               0.8785897 0.1993127 0.04350782 0.7845118   0.9564922
## Pop                 0.7540108 0.3878116 0.10416667 0.5725389   0.8958333
## Rap                 0.8047442 0.3440000 0.04651163 0.6949153   0.9534884
## Rock                0.6529938 0.5938303 0.10018215 0.4891641   0.8998179
## Songwriter          0.7754350 0.2641509 0.18497914 0.6376022   0.8150209
```

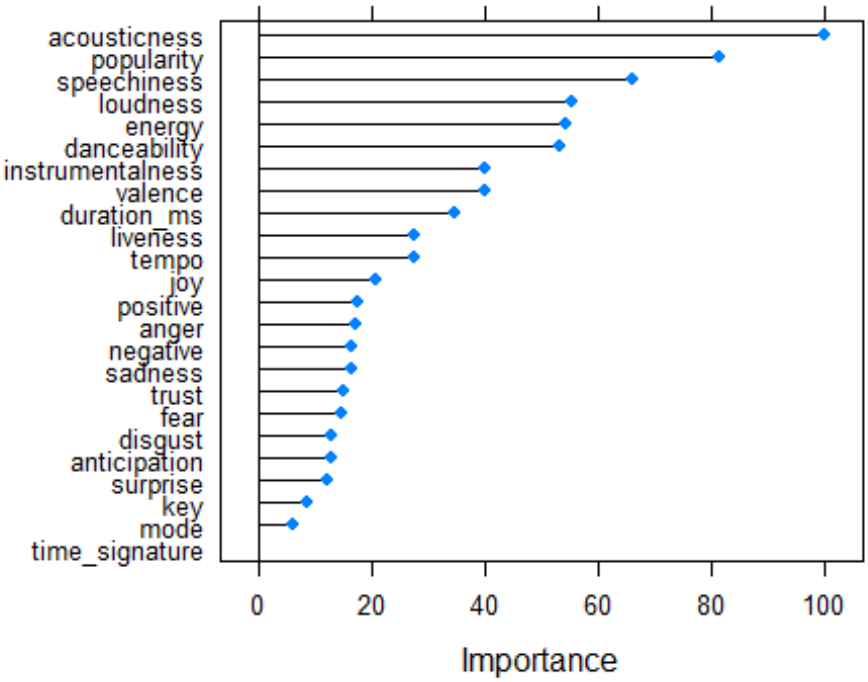
```
(acc.perf.boost <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
                                  predicted = boost.pred, metrics = 2))
```

```
## [1] 0.6607575
```

```
(err.perf.boost <- cm.performance(true =
test.spotify_right_lyrics_popularity$genre,
                                  predicted = boost.pred, metrics = 3))
```

```
## [1] 0.3392425
```

We can observe that the classification error and accuracy have improved over random forest and bagging, with 0.34 and 0.66 respectively.



To understand what is going on inside the boosting model, we can plot their relative importance. We see how the variables that contribute the most for the separation of classes are mainly those associated to audio features and popularity. The variables associated with the sentiment of the lyrics play a very secondary role.

2.4 Multiclass SVM

As last exercise we have used Support Vector Machine techniques to predict the musical genre of a song. First of all we would need to download the package e071 and to specify our main variable (genre) as a factor variable.

The first step is to divide our database in a training and test set by employing the caTools package

```
#Training set and test set
training_set = spotify_numeric[train.label,]
test_set = spotify_numeric[-train.label,]
```

2.4.1. Linear Kernel, optimal parameters

As a first analysis, we have tried to include a support vector classifier with a linear Kernel. We have used the function tune.svm provided by e1071 to analyse a different combination of the gamma and the cost parameters that modify the flexibility of the model. After having

explored all the combinations, we have retrieved the best model, and we have employed it to make some predictions on the test set.

```
set.seed(12)
tuning <- tune.svm(genre~., data = training_set, kernel='linear', gamma =
10^(-5:-3), cost = 10^(-3:0))
#summary(tuning)
bestmod = tuning$best.model
summary(bestmod)
y_predlin = predict(bestmod, newdata = test_set)
cmlin=table(y_predlin,test_set[,"genre"])
cmlin
# class labels predicted by the classifier model
predicted_labels <- y_predlin

# true labels (test set)
true_labels <- test_set$genre

classifier_metrics <- ml_test(predicted_labels, true_labels, output.as.table
= FALSE)

# overall classification accuracy
acc.svm.lin <- classifier_metrics$accuracy
acc.svm.lin
# F1-measures for classes
err.svm.lin <- classifier_metrics$error.rate
err.svm.lin
```

Our model seems to work better in a setting with gamma=0.0001 and cost=1. In this case the accuracy rate is equal to 0.56 and the error classification rate is equal to 0.44.

2.4.2 Non-Linear Kernels, optimal parameters

We will now try to use the same parameters that we have found by exploiting the tune.svm function but using different types of kernels.

```
set.seed(12)
classifier_rd=svm(formula = genre ~ .,
                  data = training_set,
                  type = 'C-classification',
                  kernel = 'radial',gamma=0.0001, cost=1)

set.seed(12)
classifier_sgm=svm(formula = genre ~ .,
                  data = training_set,
                  type = 'C-classification',
                 kernel = 'sigmoid',gamma=0.0001, cost=1)


y_pred_rd = predict(classifier_rd, newdata = test_set)
```

```
cm_rd=table(y_pred_rd,test_set[,"genre"])
cm_rd #confusion table with radial kernel


y_pred_sgm = predict(classifier_sgm, newdata = test_set)
cm_sgm=table(y_pred_sgm,test_set[,"genre"])
cm_sgm #confusion table with sigmoid kernel

# class labels predicted by the classifier model
predicted_labels2 <- y_pred_rd
predicted_labels3 <-y_pred_sgm


# true labels (test set)
true_labels2 <- test_set$genre
true_labels3 <-test_set$genre

classifier_metrics2 <- ml_test(predicted_labels2, true_labels2,
output.as.table = FALSE)
classifier_metrics3 <- ml_test(predicted_labels3, true_labels3,
output.as.table = FALSE)
# overall classification accuracy
acc.svm.rad <- classifier_metrics2$accuracy
acc.svm.sig <-classifier_metrics3$accuracy

# F1-measures for classes
err.svm.rad <- classifier_metrics2$error.rate
err.svm.sig <-classifier_metrics3$error.rate

acc.svm.rad #accuracy radial kernel
acc.svm.sig #accuracy sigmoid

err.svm.rad #error.rate radial kernel
err.svm.sig #error.rate sigmoid
```

What we can notice is that the more we add flexibility to our model, the more the predictions tend to converge toward the category “Songwriter” that as we have seen in our best subset selection analysis has a strong predictive power over the genre. One possible reason for this pattern is that our model with Linear Kernel and the best parameters capture a good equilibrium between bias and variance that is broken by the use of a more flexible Kernel.

2.4.3 Non-linear Kernels, standard parameters

Finally, we have tried to analyse the performance of non-linear kernel (radial and sigmoid) by coming back to the standard parameter of cost=10. We have tried to reproduce the results obtained for tune.svm for a radial and a sigmoid kernel but the code took too much time and did not seem to work correctly.

```r
#RADIAL Kernel
set.seed(12)
classifier_rd_std = svm(formula = genre ~ .,
                 data = training_set,
                 type = 'C-classification',
                 kernel = 'radial', cost=10)

y_rd_std = predict(classifier_rd_std, newdata = test_set)


# class labels predicted by the classifier model
predicted_labels_std <-y_rd_std

# true labels (test set)
true_labels_std <- test_set$genre

classifier_metrics_std <- ml_test(predicted_labels_std, true_labels_std,
output.as.table = FALSE)

# overall classification accuracy
accuracy_std <- classifier_metrics_std$accuracy

# F1-measures for classes
error.rate_std <- classifier_metrics_std$error.rate
accuracy_std
error.rate_std

#SIGMOID Kernel
set.seed(12)
classifier_sgm_std = svm(formula = genre ~ .,
                 data = training_set,
                 type = 'C-classification',
                 kernel = 'sigmoid', cost=10)

y_sgm_std = predict(classifier_sgm_std, newdata = test_set)

# class labels predicted by the classifier model
predicted_labels_sgmsd <-y_sgm_std
# true labels (test set)
true_labels_sgmsd <- test_set$genre

classifier_metrics_sgmsd <- ml_test(predicted_labels_sgmsd,
true_labels_sgmsd, output.as.table = FALSE)

# overall classification accuracy
accuracy_sgmsd <- classifier_metrics_sgmsd$accuracy

# F1-measures for classes
error.rate_sgmsd <- classifier_metrics_sgmsd$error.rate
```

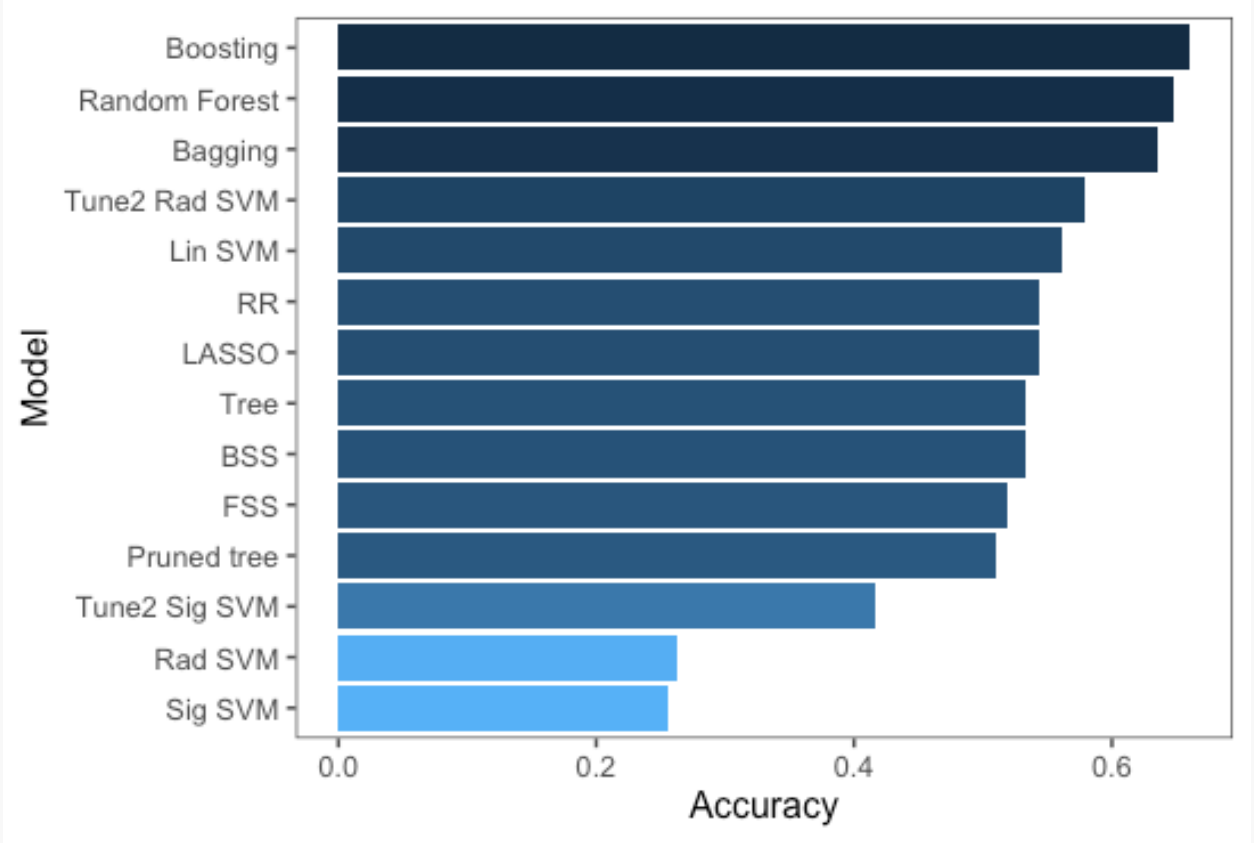
```
accuracy_sgmsd
error.rate_sgmsd
```

When we change the parameters, we can notice that both the Radial and Sigmoid Kernel perform better than in the previous case. In particular, the model with the Radial Kernel gives us the best result in terms of accuracy with a measure that reaches 'r round(accuracy_std, 2)'. Also, the sigmoid seems to perform better results than for the linear kernel with the best parameters. The first one, indeed, delivers an accuracy equal to 'r round(accuracy_sgmsd, 2)' versus the linear kernel that gives us an accuracy of 'r round (accuracy, 2)'.

2.5. Conclusion *genre*

We observed how the predicting ability of the models range from about 50% error rate to less than 35%. This is a very significant improvement! Down below we can observe the table summarizing all the results.

```
##                accuracy     error
## FSS           0.5193392 0.4806608
## BSS           0.5322321 0.4677679
## RR            0.5443191 0.4556809
## LASSO         0.5443191 0.4556809
## Tree          0.5326350 0.4673650
## Pruned tree   0.5104754 0.4895246
## Bagging       0.6361805 0.3638195
## Random Forest 0.6486704 0.3513296
## Boosting      0.6607575 0.3392425
## Lin SVM       0.5608380 0.4391620
## Rad SVM       0.5785657 0.4214343
## Sig SVM       0.4161966 0.5838034
## Tune2 Rad SVM 0.2634972 0.7365028
## Tune2 Sig SVM 0.2562450 0.7437550
```


Boosting
Random Forest
Bagging
Tune2 Rad SVM
Lin SVM
RR
LASSO
Tree
BSS
FSS
Pruned tree
Tune2 Sig SVM
Rad SVM
Sig SVM
Model
0.0
0.2
0.4
0.6
Accuracy