CS985/6 Assignment 1 Report

Members:

- Anja Wyzdak - anja.wyzdak.2019@uni.strath.ac.uk 201990375
- Michel Scharnitzki - michel.scharnitzki.2019@uni.strath.ac.uk 201987665
- Shivling Gawli - shivling.gawli.2019@uni.strath.ac.uk 201992168
- Theodor Ivanov - theodor.ivanov.2019@uni.strath.ac.uk 201989531

I. Regression Problem

1. Introduction

Spotify is the second biggest provider of music streaming services in the world in terms of monthly user base shadowed only by Apple Music. With their vast collection of tracks spanning across genres on the edge of the creative frontier, predicting musical taste and providing salient recommendations to the people who use Spotify's services has never been more challenging. A good music recommender system focuses on recognising the genre and the corresponding mood of the song so that it can make situation-specific suggestions for the next one. As a purely abstract medium, music proves very hard to classify in terms of genre because of its many nuances. These nuances ultimately contribute to its high dimensionality in terms of attributes that can be explored as classifiers. What is more, suggesting the most representative song from a corpus of titles within a genre is what sets the scene of the listening session thus allowing for its gradual exploration. In order to do that the recommender system is interested in quantifying the absolute popularity of a song. Apart from the main features of the system, recently Spotify has introduced the service Tastebreakers which allows for the exploration of the complement to the taste of the listener thus pushing them outside of their comfort zone – another feature relying heavy on classification and quantifying. Based on these concepts, the current work is interested in exploring different machine learning approaches to predicting a song's genre and popularity.

1.1 Description of the data set

For the purposes of the analysis the following libraries were used: pandas, numpy, seaborn, statsmodels, and sklearn. The environment sellected for the development and testing of the script was Colab.

In [0]:

```
# Loading the Data
from google.colab import files # insert json token
files.upload()
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/kaggle.json
!chmod 600 /root/.kaggle/kaggle.json
!pip install kaggle
!kaggle competitions download -c cs98x-spotify-regression
!kaggle competitions download -c cs98xspotifyclassification
# Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
import sklearn as skl
from sklearn import preprocessing
from sklearn.cluster import DBSCAN
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn import ensemble
from sklearn import linear_model
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
```

The data for the analysis was taken from the Spotify Songs by Decade list and consists of 4 sets divided into data for regression and classification each subdivided into sets for training and testing. Both training sets include 453 songs with 15 attributes and the test sets consisted of 114 (regression set) and 113 (classification set) with 14 attributes. The attributes provided included:

- Id - an arbitrary unique track identifier
- title - track title
- artist - singer or band
- top genre - genre of the track
- year - year of release (or re-release)
- bpm - beats per minute (tempo)
- nrgy - energy: the higher the value the more energetic
- dnce - danceability: the higher the value, the easier it is to dance to this song
- dB - loudness (dB): the higher the value, the louder the song
- live - liveness: the higher the value, the more likely the song is a live recording
- val - valence: the higher the value, the more positive mood for the song
- dur - duration: the length of the song
- acous - acousticness: the higher the value the more acoustic the song is
- spch - speechiness: the higher the value the more spoken word the song contains
- pop - popularity: the higher the value the more popular the song is (and the target variable for this problem)

In [0]:

```
# This is what we train the regression model on
training_data_reg = pd.read_csv('/content/CS98XRegressionTrain.csv', low_memory = False
)
test_data_reg = pd.read_csv('/content/CS98XRegressionTest.csv', low_memory = False)
# This is what we train the classification model on
training_data_class = pd.read_csv('/content/CS98XClassificationTrain.csv', low_memory =
False)
test_data_class = pd.read_csv('/content/CS98XClassificationTest.csv', low_memory = Fals
e)
```

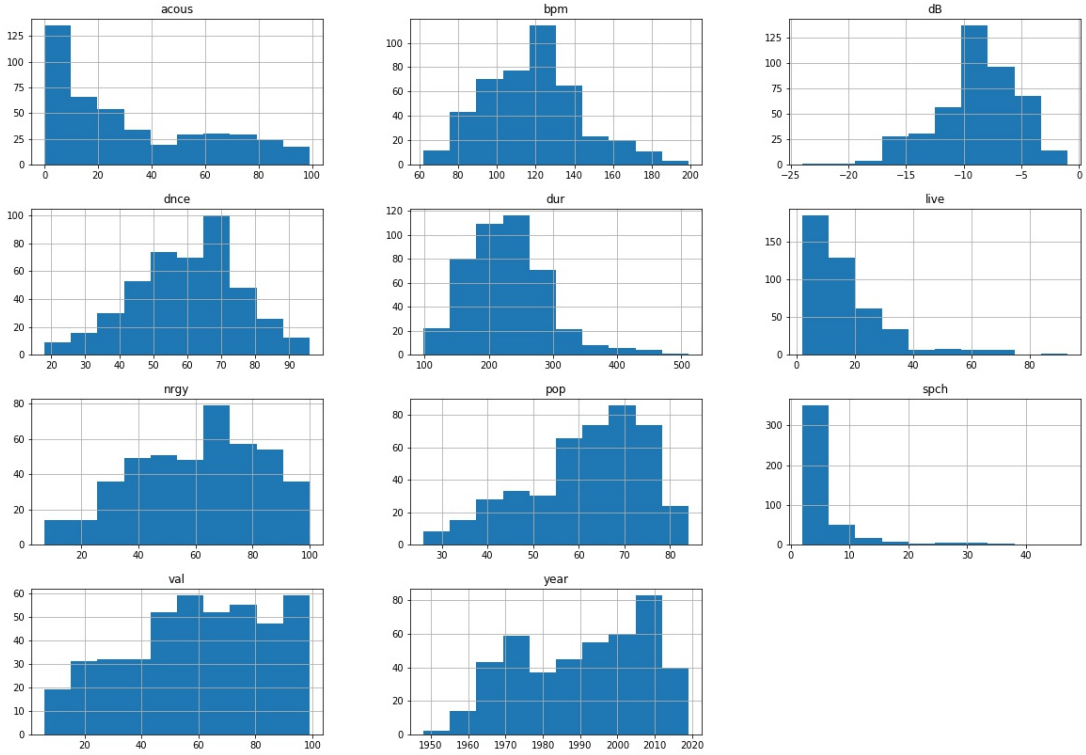
During the exploration of the data it was noted that some of the attributes were unevenly distributed (Fig. 1), with live and speechiness resembling a logarithmic distribution, and year having strong positive skew. These observations were explained by the facts that all songs have a minimum of 0:01s playtime, some songs are instrumental, and that modern technology allows for more music to be made more affordably thus causing a greater creative output which ultimately pushes the number of new songs up as data moves towards the present day.

To identify relationships between the attributes for the regression task, a correlation matrix was created and the correlations between popularity and the rest of the predictors were used as a guide when constructing the models. In the course of the analysis only the strongest predictors were used.

In [0]:

```
# Histograms
for_hist = training_data_reg.select_dtypes(exclude=['object'])
for_hist.drop(columns='Id', inplace=True)
for_hist.hist(column= for_hist.columns, figsize=(20,14))
plt.savefig('hist.jpg')
```

Figure 1: Distributions of the attributes



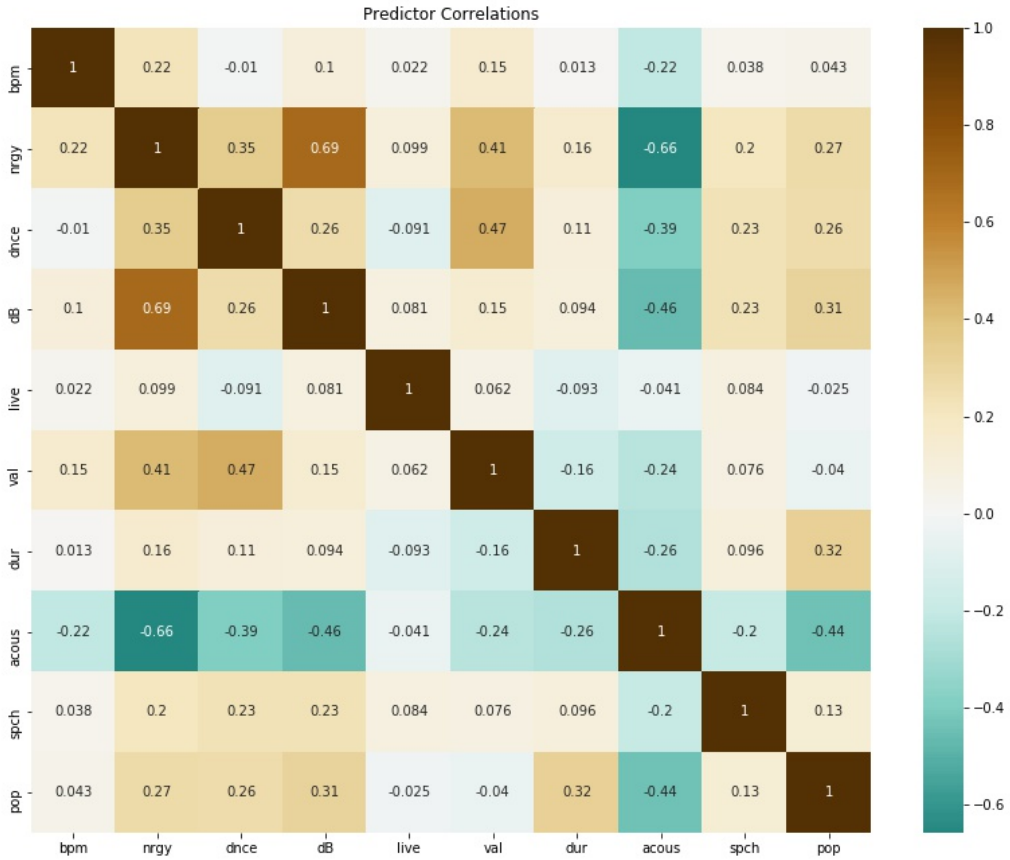
2. Regression Models & Results

This section introduces the regression models. Regression tasks usually try to predict an outcome (the dependent variable) in this case the popularity of songs, by employing other attributes (the independent variable). After carefully analysing the data set three central independent variables were identified: dB, dur and acous. DB and duration showed positive correlations of 0.31 and 0.32 respectively. Acousticness showed the highest correlation of the set with -0.44 (Fig. 2).

In [0]:

```
# Removing NaNs
training_data_reg = training_data_reg.dropna()
norm_training_data_reg = pd.DataFrame(preprocessing.normalize(training_data_reg.select_
dtypes(['number']))) # Normalising
# Correlation heatmap
training_data_reg_cov = training_data_reg.drop(columns=['Id','year']).corr() # # Removi
ng non-predictor columns
plt.figure(figsize =(14,11))
plt.title('Predictor Correlations')
sns.heatmap(training_data_reg_cov, annot=True, center=0, cmap='BrBG_r')
plt.savefig('corr.jpg')
```

Figure 2: Correlation matrix between the attributes



2.1 Linear Regression

Linear Regression is one of the most basic regression models. The idea of the linear regression method is to model the relationship between a dependent (endogenous) variable and one or multiple independent (exogenous) variables. This is done by the construction of a regression line that best fits into the data. The best fit is determined by minimising the error between the predicted and observed values. Ultimately, the model tries to minimise the errors (residuals). The residuals are the distance from the regression line to the observed data points. The Kaggle Public RMSE score of the Linear Regression was 8.23155. This number was used as a benchmark for the following models.

In [0]:

```
#Instalation of the relevant packages
#Pandas is a package for spreadsheet analysis
#Numpy is a package for computations
#Matplotl is a package for data visualization
#sklearn is a package for machine learning computation
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np
import sklearn
from sklearn import datasets
from sklearn import preprocessing

# importing the train and the test data
#drop the nan values
regtraindf = training_data_reg
regtraindf.dropna()
regtestdf = test_data_reg
regtestdf.dropna()
# split the sets in train set and test set and remove variables not used for the regres
sion problems
X_train = regtraindf.loc[:, "nrgy" : "acous"]
X_train = X_train.drop(['live','val','nrgy','dnce'], axis=1)
Y_train = regtraindf.loc[:, "pop":"pop"]
X_test = regtestdf.loc[:, "nrgy" : "acous"]
X_test = X_test.drop(['live','val','nrgy','dnce'], axis=1)

X_test = preprocessing.normalize(X_test)
X_train = preprocessing.normalize(X_train)

# train the model
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X_train, Y_train)
Y_pred = lin_reg.predict(X_test)

# create the file in the right format for upload
Y_pred = pd.DataFrame(data=Y_pred)
Y_pred.columns = ['pop']
Id = regtestdf.loc[:, "Id" : "Id"]
result = pd.concat([Id, Y_pred,], axis=1, sort=False)
result = result.set_index('Id')
result.to_csv('LinReg.csv')
```

2.2 Gradient Boosting Methods

Gradient Boosting is one of the most popular hypothesis boosting techniques and it aims at sequentially introducing predictors to an ensemble of weak learners. Unlike its close relative the Adaptive Boosting which focuses on altering the weight of the misclassified instances during training, the Gradient Boosting alters the weight of the residual errors of the previous predictor. It uses a number of decision trees as base learners and can be applied to both classification and regression problems.

The performance of the deployed Gradient Boosting Regressor for the popularity prediction showed somewhat less than satisfactory results. Converging on 50 estimators at a learning rate of 0.5, with a total of 3 features and a tree depth of 1, this model scored 51.6% accuracy on the test data and 20.1% on the validation, yielding a Kaggle Public RMSE score of 8.66544.

In [0]:

```
## Gradient Boosting: Regression
train =  pd.read_csv('/content/CS98XRegressionTrain.csv', low_memory = False)
test = pd.read_csv('/content/CS98XRegressionTest.csv', low_memory = False)
train.set_index("Id", inplace=True)
test.set_index("Id", inplace=True)
train.dropna(inplace=True)
y_train = train["pop"]

# dropping labels
train.drop(labels="pop", axis=1, inplace=True)
train_test =  train.append(test)

# delete columns that are not used as features for training and prediction
columns_to_drop = ['title', 'year','top genre','artist']
train_test.drop(labels=columns_to_drop, axis=1, inplace=True)

# Hot one encoding
train_test_dummies = train_test

# replace nulls with 0.0
train_test_dummies.fillna(value=0.0, inplace=True)

# generate feature sets (X)
X_train = train_test_dummies.values[0:438]
X_test = train_test_dummies.values[438:]

# scaling the data
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_scale = scaler.fit_transform(X_train)
X_test_scale = scaler.transform(X_test)
from sklearn.model_selection import train_test_split

# splitting the data
X_train_sub, X_validation_sub, y_train_sub, y_validation_sub = train_test_split(X_train
_scale, y_train, random_state=0, shuffle=None)

# importing machine learning algorithms
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc

# computing the accuracy scores on training and validation sets for different learning
 rates
learning_rates = [0.05, 0.1, 0.25, 0.5, 0.75, 1]
for learning_rate in learning_rates:
    gb = GradientBoostingRegressor(n_estimators=50, learning_rate = learning_rate, max_
features=3, max_depth = 1, random_state = 0)
    gb.fit(X_train_sub, y_train_sub)
    print("Learning rate: ", learning_rate)
    print("Accuracy score (training): {0:.3f}".format(gb.score(X_train_sub, y_train_sub
)))
    print("Accuracy score (validation): {0:.3f}".format(gb.score(X_validation_sub, y_va
lidation_sub)))
    print()

# Outputing confusion matrix and classification report of Gradient Boosting algorithm o
n validation set
gb = GradientBoostingRegressor(n_estimators=50, learning_rate = 0.5, max_features=3, ma
x_depth = 1, random_state = 0)
```

```
gb.fit(X_train_sub, y_train_sub)
predictions = gb.predict(X_test_scale)
print("Accuracy score (training): {0:.3f}".format(gb.score(X_train_sub, y_train_sub)))
print("Accuracy score (validation): {0:.3f}".format(gb.score(X_validation_sub, y_valida
tion_sub)))

reg_result = pd.DataFrame()
reg_result['pop'] = predictions.T.round(decimals=4)
reg_result['Id'] = test.index
reg_result.index = reg_result['Id']
reg_result.drop(columns='Id',inplace=True)
reg_result.to_csv('reg_enc_boost.csv')
#!kaggle competitions submit -c cs98x-spotify-regression -f reg_enc_boost.csv -m "Boost
with Encoding"
```

2.3 Random Forest

The third model used for the regression task was the Random Forest Regression technique. The Random Forest Regression uses multiple decision trees and bootstrap aggregation (bagging). Bagging performs sampling with replacement on different random subsets of the dataset. In the Random Forest Regression each decision tree is trained on a different subset. The multiple decision trees are combined to calculate the final output. In contrast to the Gradient Boosting, Random Forest Regression builds each decision tree independently and combines the results in the end. Grid search was used to find the optimal hyperparameters for the regression model. Grid search cross validates different pre-defined parameters to find the parameters which deliver the highest accuracy levels. Hyperparameter tuning reduced the RMSE score from 8.09130 to 7.9538. In the last step different data scaling techniques were implemented. At this point it is important to notice that this whole process was done in multiple iterations. After each scaling method the grid search function was applied to find the optimal parameters. The optimal parameters for the final Random Forest Regression model were a maximum split of 4 (max_depth = 4) and a number of 50 different trees (n_estimators = 50). The final Kaggle Public RMSE was 7.23191 after employing the normalisation data scaling technique.

In [0]:

```
#2.3 Random Forest  Regression
#importing the train and the test data
#drop the nan values
regtraindf = training_data_reg
regtraindf.dropna()
regtestdf = test_data_reg
regtestdf.dropna()
#split the sets in train set and test set and remove variables not used for the regress
ion problems

X_train = regtraindf.loc[:, "nrgy" : "acous"]
X_train = X_train.drop(['live','val','nrgy','dnce'], axis=1)
Y_train = regtraindf.loc[:, "pop":"pop"]
X_test = regtestdf.loc[:, "nrgy" : "acous"]
X_test = X_test.drop(['live','val','nrgy','dnce'], axis=1)


#Employ different data scaling techniques
#normalize scaler results in RMSE of 7.23191
X_test = preprocessing.normalize(X_test)
X_train = preprocessing.normalize(X_train)

# ================================================================================
# #MinMaxScaler results in RMSE of 8.06374
# from sklearn.preprocessing import MinMaxScaler
# scaler = MinMaxScaler()
# X_train = scaler.fit_transform(X_train)
# X_train = pd.DataFrame(data=X_train)
# X_test = scaler.fit_transform(X_test)
# X_test = pd.DataFrame(data=X_test)
#
# #Normalizer results in RMSE of 7.42832
# from sklearn.preprocessing import Normalizer
# scaler = Normalizer()
# X_train = scaler.fit_transform(X_train)
# X_train = pd.DataFrame(data=X_train)
# X_test = scaler.fit_transform(X_test)
# X_test = pd.DataFrame(data=X_test)
#
# #RobustScaler results in RMSE of 7.92187
# from sklearn.preprocessing import RobustScaler
# scaler = RobustScaler()
# X_train = scaler.fit_transform(X_train)
# X_train = pd.DataFrame(data=X_train)
# X_test = scaler.fit_transform(X_test)
# X_test = pd.DataFrame(data=X_test)
# ================================================================================

#import the RandomForestRegressor package
from sklearn.ensemble import RandomForestRegressor
#find the optimal parameters
rfc = RandomForestRegressor()
#define parameters for optimisation
parameters = {
    "n_estimators":[5,10,50,100,250],
    "max_depth":[1,2,4,8,16,32,None]

}
```

```
#import the packages for the GridSearchCV optimisaion and import mean_squared_error sco
rer
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
from sklearn.metrics import mean_squared_error
scorer = make_scorer(mean_squared_error, greater_is_better=False)
cv = GridSearchCV(rfc, parameters, scoring=scorer, cv=5)
cv.fit(X_train,Y_train.values.ravel())

#print the optimisation results
def display(results):
    print(f'Best parameters are: {results.best_params_}')
    print("\n")
    mean_score = results.cv_results_['mean_test_score']
    std_score = results.cv_results_['std_test_score']
    params = results.cv_results_['params']
    for mean,std,params in zip(mean_score,std_score,params):
        print(f'{round(mean,3)} + or -{round(std,3)} for the {params}')

display(cv)

#train the model with the optimised parameters
model = RandomForestRegressor(max_depth = 4, n_estimators = 50, random_state =0)
model.fit(X_train, Y_train)
Y_pred = model.predict(X_test)

#create the file in the right format for upload
Y_pred = pd.DataFrame(data=Y_pred)
Y_pred.columns = ['pop']
Id = regtestdf.loc[:, "Id" : "Id"]
result = pd.concat([Id, Y_pred,], axis=1, sort=False)
result = result.set_index('Id')
result.to_csv('randomforest_normalize.csv')
```

2.4 Support Vector Regression

The Support Vector Regression (SVR) is built on a similar approach as the Support Vector Machine (SVM) Classification model. The goal of the SVM is to find the hyperplane that maximises the margin between the observations and avoiding margin violations (hard margin) or limiting them (soft margin). The SVR tries to fit the error rates within the margin and maximise the margin. The grid search method was used to find the optimal hyperparameters for the model. The optimal parameters for the SVR model were a polynomial kernel (kernel='poly'), a kernel coefficient of 0.6 (gamma = 0.6) and a strength of regularisation of 1000 (C = 1000). Again, the normalisation data scaling technique was applied. The final Kaggle Public RMSE was 7.36986 in comparison to a RMSE of 7.84204 using the Robust Scaler method.

In [0]:

```
#4.4 Support Vector Regression
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np
import sklearn
from sklearn import datasets
from sklearn import preprocessing

#importing the train and the test data
#drop the nan value
regtraindf = training_data_reg
regtraindf.dropna()
regtestdf = test_data_reg
regtestdf.dropna()

#split the sets in train set and test set and remove variables not used for the regress
ion problems
X_train = regtraindf.loc[:, "nrgy" : "acous"]
X_train = X_train.drop(['live','val','nrgy','dnce'], axis=1)
Y_train = regtraindf.loc[:, "pop":"pop"]
X_test = regtestdf.loc[:, "nrgy" : "acous"]
X_test = X_test.drop(['live','val','nrgy','dnce'], axis=1)

#Different data scaling techniques
#results in RMSE of 7.36986
X_test = preprocessing.normalize(X_test)
X_train = preprocessing.normalize(X_train)

# ================================================================================
# #Scale the data
# #results in RMSE of 7.93718
# from sklearn.preprocessing import MinMaxScaler
# scaler = MinMaxScaler()
# X_train = scaler.fit_transform(X_train)
# X_train = pd.DataFrame(data=X_train)
# X_test = scaler.fit_transform(X_test)
# X_test = pd.DataFrame(data=X_test)
#
# #results in RMSE of 7.66126
# from sklearn.preprocessing import RobustScaler
# scaler = RobustScaler()
# X_train = scaler.fit_transform(X_train)
# X_train = pd.DataFrame(data=X_train)
# X_test = scaler.fit_transform(X_test)
# X_test = pd.DataFrame(data=X_test)
# ================================================================================

from sklearn.svm import SVR
#find the optimal parameters
SVR_M = SVR()
#define parameters for optimisation
parameters = {
    "gamma" : [1e-4, 1e-3, 0.01, 0.1, 0.2, 0.5, 0.6, 0.9],
    "kernel" : ['rbf', 'poly', 'linear', 'sigmoid'],
    "C": [1, 10, 100, 1000, 10000]

}

#import the packages for the GridSearchCV optimisaion and import mean_squared_error sco
```

```
rer
from sklearn.metrics import make_scorer
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV
scorer = make_scorer(mean_squared_error, greater_is_better=False)
cv = GridSearchCV(SVR_M, parameters, scoring=scorer)
cv.fit(X_train,Y_train.values.ravel())

#print the optimisation results
def display(results):
    print(f'Best parameters are: {results.best_params_}')
    print("\n")
    mean_score = results.cv_results_['mean_test_score']
    std_score = results.cv_results_['std_test_score']
    params = results.cv_results_['params']
    for mean,std,params in zip(mean_score,std_score,params):
        print(f'{round(mean,3)} + or -{round(std,3)} for the {params}')

display(cv)

#train the model with the optimised parameters
SVR_model = SVR(kernel='poly', C = 1000, gamma = 0.6)
SVR_model.fit(X_train, Y_train)
Y_pred = SVR_model.predict(X_test)

#create the file in the right format for upload
Y_pred = pd.DataFrame(data=Y_pred)
Y_pred.columns = ['pop']
Id = regtestdf.loc[:, "Id" : "Id"]
result = pd.concat([Id, Y_pred,], axis=1, sort=False)
result = result.set_index('Id')
result.to_csv('SVR_normalize.csv')
```

3. Conclusion & Reflection

What this analysis clearly shows is that the results of the regression models can only be as good as the input data. Before starting this report, we did not expect that scaling of the data can improve the results so significantly. What this report demonstrates is that extensive pre-processing of the input data can go a long way.

Deciding on the right model is definitely important but also fine tuning the model is a crucial step. By tuning the SVR and Random Forest model the final results fall in the same area of 7.36986 and 7.23191 respectively. It is important to highlight that the focus of the regression task clearly laid on minimising the RMSE. However, it is also essential to consider other parameters like the MAE, R-Squared or F-Test to evaluate the models in greater detail. Additionally, employing other techniques as deep learning via neural networks might further increase the precision in predicting the popularity of Spotify songs. The regression task is an up-to-date problem that many companies and institutions face at the moment. As Quantitative Finance students we will definitely be able to transfer our experience to other point estimation exercises as predicting stock prices or forecasting company revenues.

II. Classification

1. Introduction

Spotify is the second biggest provider of music streaming services in the world in terms of monthly user base shadowed only by Apple Music. With their vast collection of tracks spanning across genres on the edge of the creative frontier, predicting musical taste and providing salient recommendations to the people who use Spotify's services has never been more challenging. A good music recommender system focuses on recognising the genre and the corresponding mood of the song so that it can make situation-specific suggestions for the next one. As a purely abstract medium, music proves very hard to classify in terms of genre because of its many nuances. These nuances ultimately contribute to its high dimensionality in terms of attributes that can be explored as classifiers. What is more, suggesting the most representative song from a corpus of titles within a genre is what sets the scene of the listening session thus allowing for its gradual exploration. In order to do that the recommender system is interested in quantifying the absolute popularity of a song. Apart from the main features of the system, recently Spotify has introduced the service Tastebreakers which allows for the exploration of the complement to the taste of the listener thus pushing them outside of their comfort zone – another feature relying heavy on classification and quantifying. Based on these concepts, the current work is interested in exploring different machine learning approaches to predicting a song's genre and popularity.

1.1 Description of the data set

For the purposes of the analysis the following libraries were used: pandas, numpy, seaborn, statsmodels, and sklearn. The environment sellected for the development and testing of the script was Colab.

The data for the analysis was taken from the Spotify Songs by Decade list and consists of 4 sets divided into data for regression and classification each subdivided into sets for training and testing. Both training sets include 453 songs with 15 attributes and the test sets consisted of 114 (regression set) and 113 (classification set) with 14 attributes. The attributes provided included:

- Id - an arbitrary unique track identifier
- title - track title
- artist - singer or band
- top genre - genre of the track
- year - year of release (or re-release)
- bpm - beats per minute (tempo)
- nrgy - energy: the higher the value the more energetic
- dnce - danceability: the higher the value, the easier it is to dance to this song
- dB - loudness (dB): the higher the value, the louder the song
- live - liveness: the higher the value, the more likely the song is a live recording
- val - valence: the higher the value, the more positive mood for the song
- dur - duration: the length of the song
- acous - acousticness: the higher the value the more acoustic the song is
- spch - speechiness: the higher the value the more spoken word the song contains
- pop - popularity: the higher the value the more popular the song is (and the target variable for this problem)

During the exploration of the data it was noted that some of the attributes were unevenly distributed, with live and speechiness resembling a logarithmic distribution, and year having strong positive skew (Fig. 1). These observations were explained by the facts that all songs have a minimum of 0:01s playtime, some songs are instrumental, and that modern technology allows for more music to be made more affordably thus causing a greater creative output which ultimately pushes the numbers up as data moves towards the present day.

To identify relationships between the attributes for the classification task, a correlation matrix was created and the correlations between popularity and the rest of the predictors were used as a guide when constructing the models. In the course of the analysis only the strongest classifiers were used (Fig. 2).

In [0]:

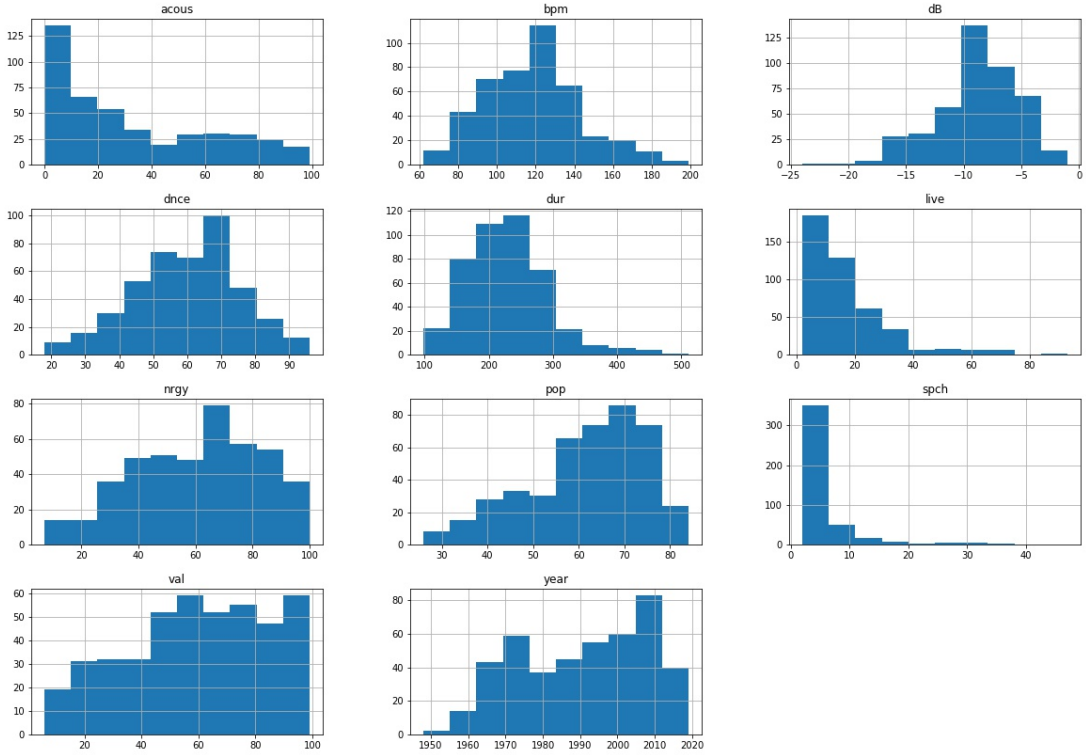
```
# Loading the Data
# from google.colab import files # upload json token
# files.upload()
# !mkdir -p ~/.kaggle
# !cp kaggle.json ~/.kaggle/kaggle.json
# !chmod 600 /root/.kaggle/kaggle.json
# !pip install kaggle
# !kaggle competitions download -c cs98x-spotify-regression
# !kaggle competitions download -c cs98xspotifyclassification

# Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
import sklearn as skl
from sklearn import preprocessing
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn import ensemble
from sklearn import linear_model
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# This is what we train the regression model on
training_data_reg = pd.read_csv('/content/CS98XRegressionTrain.csv', low_memory = False
)
test_data_reg = pd.read_csv('/content/CS98XRegressionTest.csv', low_memory = False)
# This is what we train the classification model on
training_data_class = pd.read_csv('/content/CS98XClassificationTrain.csv', low_memory =
False)
test_data_class = pd.read_csv('/content/CS98XClassificationTest.csv', low_memory = Fals
e)

# Histograms
for_hist = training_data_reg.select_dtypes(exclude=['object'])
for_hist.drop(columns='Id', inplace=True)
for_hist.hist(column= for_hist.columns, figsize=(20,14))
plt.savefig('hist.jpg')
```

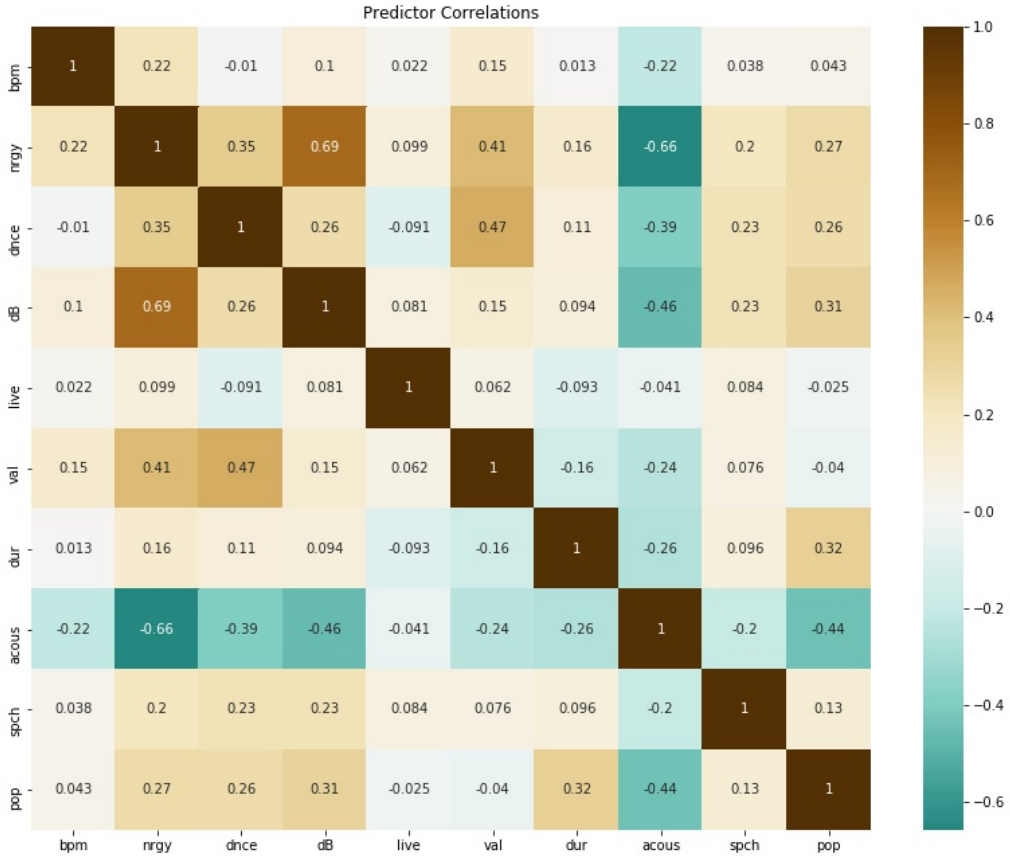
Figure 1: Distributions of the attributes



In [0]:

```
# Removing NaNs
training_data_reg = training_data_reg.dropna()
norm_training_data_reg = pd.DataFrame(preprocessing.normalize(training_data_reg.select_
dtypes(['number']))) # Normalising
# Correlation heatmap
training_data_reg_cov = training_data_reg.drop(columns=['Id','year']).corr() # # Removi
ng non-predictor columns
plt.figure(figsize =(14,11))
plt.title('Predictor Correlations')
sns.heatmap(training_data_reg_cov, annot=True, center=0, cmap='BrBG_r')
plt.savefig('corr.jpg')
```

Figure 2: Correlation matrix between the attributes



2. Classification Models and Results

2.1 Support Vector Machine

A very prominent supervised machine learning algorithm that performs classification is the Support Vector Classifier (SVC). It is used to find a decision boundary in more than two dimensions that divides the given data into different categories (here: top genres) and is suitable for small datasets. The decision boundary is found through the maximum margin, i.e. the distance between data points of different classes. To find the most accurate prediction score, the three hyper-parameters of the SVC (C, kernel and gamma) were optimized by importing the GridSearchCV from sklearn.modelapproach. Grid Search evaluates several combinations of algorithm parameters. To transform the input from a low-dimensional space into a higher-dimensional space, the Grid Search determines a sigmoid kernel approach to separate the data with a non-linear hyperplane. In addition, Grid Search proposes a low 0.01 gamma value, meaning that far away data points also be considered to get the decision boundary. A low gamma can also help to prevent overfitting. Lastly, the parameter tuning recommends a regularization parameter C of 10, which again is quite low and tries to stop overfitting. The Classifier has been trained with 11 scaled numerical features only (the columns 'title', 'artist' were removed from the training and test dataset) and achieved a performance Kaggle Public score of 0.33928. Further adjustments such as removing different columns (i.e. year or val), increasing the value of gamma and encoding the column ‘artist’ to use it for prediction did not provide a higher score.

In [0]:

```
#Set-Up
#import libraries and datasets
import pandas as pd
import numpy as np
from sklearn.preprocessing import scale
from sklearn import preprocessing
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score

#training_data_class
train = pd.read_csv('/content/CS98XClassificationTrain.csv', low_memory = False)
#test_data_class
test = pd.read_csv('/content/CS98XClassificationTest.csv', low_memory = False)
train.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)

#rename the column to avoid error
train = train.rename(columns = {'top genre' : 'topgenre'})

##remove some columns that are not used in this project
train = train.drop('artist', axis = 1)
train = train.drop('title', axis = 1)
test = test.drop('artist', axis = 1)
test = test.drop('title', axis = 1)

#scale train and test data
train[['topgenre']] = train[['topgenre']].astype(str)
trainfeat = train.drop('topgenre', axis =1)
scaledfeat = preprocessing.scale(trainfeat)
trainoutcome = train.topgenre #what we predict
scaledtest = preprocessing.scale(test)

#find optimal parameters for svc
param_grid = {'C': [0.1,1, 10, 100], 'gamma': [1,0.1,0.01,0.001],'kernel': ['rbf', 'pol
y', 'sigmoid']}

#apply param_grid
grid = GridSearchCV(SVC(),param_grid,refit=True,verbose=2)
grid.fit(scaledfeat, trainoutcome)

print(grid.best_estimator_)

svc = SVC(C=10, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.01, kernel='sigmoid',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)

svc.fit(scaledfeat, trainoutcome)
cross_val_score(svc, scaledfeat, trainoutcome, cv = 5, scoring = 'accuracy' )

#predict genre
clean_test_class = test.dropna()
y_pred = svc.predict(scaledtest)
yresult = pd.DataFrame()
yresult['top genre'] = y_pred.T
predict = pd.merge(yresult['top genre'], clean_test_class, left_index=True, right_index
=True)
predict = predict[['Id',  'top genre']]
predict.to_csv('svc.csv')
```

```
#!kaggle competitions submit -c cs98x-spotify-regression -f svc.csv -m "SVC"
```

2.2 Decision Trees

Decision Trees are a popular approach in supervised machine learning when it comes to classification since they are far less sensitive to noise and irrelevant/missing values than other classifiers. The Decision Tree Classifier (DTC) tries to find a way to split the training data until all the data has been classified into certain classes (here: top genres). To measure the quality of those splits, the DTC's parameter 'criterion' can either be Gini (for Gini impurity) or entropy (for information gain). To decide which criterion fits best to predict the top genre of a song, the Grid Search was used. It recommends applying the entropy criteria which may compute a little slower than the Gini index since entropy is computing a logarithmic function. Secondly, Grid Search proposes a maximum depth of 4 meaning that the length of the paths from the root of the tree to the leaf is no greater than 4. To see whether the parameter tuning increases the accuracy score, the DTC was applied with and without adjusted parameters. Given the same data and attributes which were already used with the SVC, the DTC scores an accuracy between 28% and 29% at most. After applying a maximum depth of 4 and an entropy criterion, the accuracy indeed increases. Given the optimized parameters, the DTC escalates to 32%, which still less than the SVC score.

In [0]:

```
#import libraries and Classifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn import decomposition, datasets
from sklearn import tree

#define variables
X = scaledfeat
y = trainoutcome
tree_clf = DecisionTreeClassifier(max_depth=5)
tree_clf.fit(X,y)

sc = StandardScaler()
pca = decomposition.PCA()
decisiontree = tree.DecisionTreeClassifier()
pipe = Pipeline(steps=[('sc', sc),
                       ('decisiontree', decisiontree)])

#define parameters which we want to tune
criterion = ['gini', 'entropy']
max_depth = [4,6,8,12]
parameters = dict(decisiontree__criterion=criterion,
                  decisiontree__max_depth=max_depth)

tree_clf = GridSearchCV(pipe, parameters)
tree_clf.fit(X, y)

#print best parameter values
print('Best Criterion:', tree_clf.best_estimator_.get_params()['decisiontree__criterio
n'])
print('Best max_depth:', tree_clf.best_estimator_.get_params()['decisiontree__max_dept
h'])
print(); print(tree_clf.best_estimator_.get_params()['decisiontree'])

#adapt best parameters
tree.clf = DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=4, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')

CV_Result = cross_val_score(tree.clf, X, y, cv=3, n_jobs=-1)
print(); print(CV_Result)
print(); print(CV_Result.mean())
print(); print(CV_Result.std())

#show accuracy of DecisionTreeClassifier
cross_val_score(svc, scaledfeat, trainoutcome, cv = 5, scoring = 'accuracy' )

#Predict top genre using the test dataset
y_pred = svc.predict(scaledtest)
yresult = pd.DataFrame()
yresult['top genre'] = y_pred.T
predict = pd.merge(yresult['top genre'], clean_test_class, left_index=True, right_index
=True)
predict = predict[['Id',  'top genre']]
predict.to_csv('decision.csv')
```

```
#!kaggle competitions submit -c cs98xspotifyclassification -f decision.csv -m
```

2.3 Gradient Boosting: Classification

Gradient Boosting is a hypothesis boosting technique which aims at sequentially introducing predictors to an ensemble of weak learners. Unlike its close relative the Adaptive Boosting which focuses on altering the weight of the misclassified instances during training, the Gradient Boost alters the weight of the residual errors of the previous predictor. It uses a number of decision trees as base learners and can be applied to both classification and regression problems.

In terms of predicting the genre of the songs in our case, the Gradient Boosting Classifier class was used by training it with the most of the numeric data set attributes (excluding year, id, valence, and liveliness) as well as hot one encoded version of the artist name since as established earlier in the preliminary analysis, artists have high predictive power due to their genre loyalty. After performing iterative optimisation over the parameters of the model, the final criteria converged on 1000 estimators at a learning rate of 0.1, with a total of 8 features and a tree depth of 2. The performance score for this model was 100% for the training set and 47.3% for the validation. The superior performance of this model to the others deployed throughout the research process yielded a Kaggle Public score of 50% and is the one chosen to represent this work.

In [0]:

```
## Gradient Boosting: Classification
train =  pd.read_csv('/content/CS98XClassificationTrain.csv', low_memory = False)
test = pd.read_csv('/content/CS98XClassificationTest.csv', low_memory = False)
train.set_index("Id", inplace=True)
test.set_index("Id", inplace=True)
train.dropna(inplace=True)
test.dropna(inplace=True)
y_train = train["top genre"] # assigning training results

train.drop(labels="top genre", axis=1, inplace=True)
train_test =  train.append(test)

# delete columns that are not used as features for training and prediction
columns_to_drop = ['title', 'year', 'val', 'live']
train_test.drop(labels=columns_to_drop, axis=1, inplace=True)

# converting objects to numbers (hot-one encoding)
train_test_dummies = pd.get_dummies(train_test, columns=["artist"])

# replace nulls with 0.0
train_test_dummies.fillna(value=0.0, inplace=True)

# generate feature sets (X)
X_train = train_test_dummies.values[0:438]
X_test = train_test_dummies.values[438:]
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_scale = scaler.fit_transform(X_train)
X_test_scale = scaler.transform(X_test)
from sklearn.model_selection import train_test_split

# splitting the data
X_train_sub, X_validation_sub, y_train_sub, y_validation_sub = train_test_split(X_train
_scale, y_train, random_state=0, shuffle=None)

# import machine learning algorithms
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc

# train with Gradient Boosting algorithm compute the accuracy scores on train and valid
ation sets when training with different learning rates
learning_rates = [0.05, 0.1]
for learning_rate in learning_rates:
    gb = GradientBoostingClassifier(n_estimators=200, learning_rate = learning_rate, ma
x_features=8, max_depth = 2, random_state = 0)
    gb.fit(X_train_sub, y_train_sub)
    print("Learning rate: ", learning_rate)
    print("Accuracy score (training): {0:.3f}".format(gb.score(X_train_sub, y_train_sub
)))
    print("Accuracy score (validation): {0:.3f}".format(gb.score(X_validation_sub, y_va
lidation_sub)))
    print()

# outputing confusion matrix and classification report of Gradient Boosting algorithm o
n validation set
gb = GradientBoostingClassifier(n_estimators=1000, learning_rate = 0.1, max_features=8,
max_depth = 2, random_state = 0) # set estimator to 1000
gb.fit(X_train_sub, y_train_sub)
predictions = gb.predict(X_test_scale)
```

```
print("Accuracy score (training): {0:.3f}".format(gb.score(X_train_sub, y_train_sub)))
print("Accuracy score (validation): {0:.3f}".format(gb.score(X_validation_sub, y_valida
tion_sub)))

# Submitting results to Kaggle
class_result = pd.DataFrame()
class_result['top genre'] = predictions.T
class_result['Id'] = test.index
class_result.index = class_result['Id']
class_result.drop(columns='Id',inplace=True)
class_result.to_csv('class_enc_boost.csv')

#!kaggle competitions submit -c cs98xspotifyclassification -f class_enc_boost.csv -m "E
ncoded boosting"
```

---

3. Conclusion & Reflection

The wide variety of models used during this analysis yielded various results. It must be noted that despite its promising performance, Gradient Boosting comes at a high computational cost. This is due to the fact that in its core Gradient Boosting is a linear algorithm which requires sequential processing and is thus hard to spread over a multithreaded architecture. Thinking from a scalability perspective, this adds an extra layer of complexity when deploying in a real-life scenario, and in the context of song classification it can prove difficult to implement in a dynamic environment with new songs being released daily. What is more, as in the regression example, the quality and scaling of the data is of paramount importance to the performance of the model.

Looking forward, a replication of this work will greatly benefit from exploring the application of neural network models to the classification of song genres. Being far more dynamic in terms of accommodating new instances and taking the full advantage of cutting-edge multithreaded hardware, neural networks are a solution that an ever-growing number of companies focus their resources on, one which can be even better suited to the current task.
