Spotify Regression Problem

Group 13: Abby Steedman: 202353510, Benjamin Ashworth: 202385669, Christopher Reilly: 202365778 Haseeb Ramey: 202378564, Kieran McAvoy: 202394078

```
from google.colab import files
files.upload()
```

Introduction

Spotify, one of the most popular music streaming services, provides listeners with songs, albums and podcasts from the latest hits to classics. But what makes a song popular? We aim to build a model that can predict the popularity of a song based on the its musical features such as temperament and loudness. As popularity is a numerical value, we will be treating this as a regression problem. Used as the preferred evaluation metric for regression problems, models performance will be measured by the Root Mean Square Error (RMSE) which will give us an understanding of how much error the model makes when predicting values.

Required Libraries

This notebook uses several python packages which were essential to performing this analysis. These were NumPy, Pandas, Sci-Kit Learn, MatplotLib and Seaborn

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

```
# load training & test datasets
train = pd.read_csv("CS98XRegressionTrain.csv")
test = pd.read_csv("CS98XRegressionTest.csv")
```

Exploring, Cleaning and Preparing the Data

We begin by exploring and visualising the data to understand what preparation and preprocessing needs to take place to ensure our data is clean and capable of being fed into our models.

```
train.head()
```

```
train.info()
```

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(18, 12))
alpha_level = 0.65
axes[0].scatter(train.index, train['pop'], alpha=alpha_level)
axes[0].set_title('Figure 1: Scatterplot of Songs Popularity')
axes[0].set_xlabel('Songs')
axes[0].set_ylabel('Popularity')
train['top genre'].value_counts().plot(kind='bar', ax=axes[1])
axes[1].set_title('Figure 2: Bar Chart of Genres')
plt.show()
```

```
%matplotlib inline
train.hist(bins=50, figsize=(12,7))
plt.suptitle('Figure 3: Histograms of Training Set Variables')
plt.tight_layout()
plt.show()
```

From the graphs a few things are noticeable:

1. Figure 1 shows that the data is messy and not linear, a straight line would never fit this complex data properly, therefore we need to consider how we can combat this to create an effective model.
2. Figure 2 shows the extent of the messiness of the data, and is not very helpful in showcasing the distribution of the top genre attribute in the training set. Clearly there is a significant number of different genres, some with very little instances in each category. This could make the data very difficult for the algorithm to learn from and tricky to generalise. On inspection of the rest of the data, it is assumed that the same issue is present in the 'artist', 'title' and 'year' features.

3. Finally, in figure 3 we can see that many attributes are tail-heavy having most instances on the left handside. This could make it difficult for the algorithm to detect patterns which would produce accurate predictions. Further, the attributes have very different scales. For example, 'dB' ranges from -25 to 0 compared with duration, which ranges from 100 to 500. This could cause potential issues. Therefore, we will need to consider scaling to ensure all attributes are are aligned to the same scale to allow the model to learn effectively.

Thus, it is evident that a significant volume of data cleaning and preperation is needed before we can fit to and predict from our models.

```
train.isnull().sum()
```

```
test.isnull().sum()
```

Initial analysis showed that there was a small number of null values in the 'top genre' feature in the training set, we pondered a few options on how to approach this. Firstly, subsituting the nulls with the mode. We decided this was not an optimal solution as genres differ a lot dependent on the song, the genre with the most number of songs may not necessarily be the genre of the song that is null. Further, as we did not want to create an inaccurate model, dropping these rows to avoid any misclassification seemed like the most ideal solution. As these missing genres were less than 5% of all training instances, we decided to drop these columns to avoid our model being trained with inaccurate and incorrect data.

On the other hand in the test set, only one value was null (again in the 'Top Genre') category. In this case, we filled the null value with the mode of 'Top Genres', which was 'adult standards'. As it was only one value, we judged that filling it with the mode would not have made as much difference as filling the NAs in the training data, of which there were 15.

Splitting the Data

Instead of performing a test-train split, as the dataset was already split into test and train datasets, we decided to test on our target ('pop' from the training dataset). Although this could lead to overfitting in somecases, as our training data was so small, we wanted to include as many instances as possible in the training data to try ensure our model was as accurate to the highest extent.

```
train = train.dropna()
train_target = train['pop']
train_data = train.drop(['pop'], axis=1)
test = test.fillna('adult standards')
train_data.info()
```

Irrelevant Features

When developing our algorithim, we decided to drop some relatively irrelavant and arbitrary columns that are not related to a songs popularity. These included *'Id', 'artist'* and '*title*'. *'Id'* was simply to define it within the dataset, thus would render useless in predicting the popularity of a song. Moreover, we decided to drop '*title*' as every song title is different, so having to one-hot encode for example would have added over 300 attributes into the dataset. This would have meant the model was slow and thus ineffcient, and would heavily overfit to the training data, as the titles in the test set would be entirely different so the model would not generalise well. We further chose to drop *'artist'* as it also contained many unique values, which would create difficulty when changing them to numerical values, slowing down the model and causing overfitting to the training data.

This would allow our models to learn easily from the training data and make better predictions. We also explored the structure of the 'top genre' category to see if we wanted to keep it in, as genre is generally a distinct indicator of a songs popularity (E.g. If a genre is popular, artists are likely to release songs into that genre to have a popular song). Thus, we decided to keep top genre within our data, however perform some transformations to allow it to be more accurate.

```
train_data = train_data.drop(['title', 'Id', 'artist'], axis=1)
test = test.drop(['title', 'Id', 'artist'], axis=1)
```

```
train_data['top genre'].value_counts()
```

```
test['top genre'].value_counts()
```

Feature Engineering

To first reduce the number of features in the training set to regularise the model, improve its accurracy and ability to generalise to unseen data, we took a second look at our variables. Looking at correlations between the numerical variables can help with understanding their relationships.

```
train_numerical = train_data.drop(['year', 'top genre'], axis=1)
test_numerical = test.drop(['year', 'top genre'], axis=1)
corr = train_numerical.corr()
sns.heatmap(corr)
plt.title('Figure 4: Heatmap of Correlations between Numerical Variables ')
plt.show()
```

Figure 4, shows that 'nrgy' (Energy) and 'dB' (Decibels) are the most strongly correlated variables ($p = 0.08$). To avoid any multicollinearity issues that could potentially interfere with the models interpretation of the effect of the variables on the target, we decided to combine energy and decibels into one feature. Furthermore, combining highly correlated variables also enables the model to find patterns in the data that may have gone unnoticed without the combining of attributes.

```
train_numerical['LoudEnergy'] = train_numerical['nrgy'] * train_numerical['dB']
test_numerical['LoudEnergy'] = test_numerical['nrgy'] * test_numerical['dB']
```

```
train_num_attribs = list(train_numerical)
train_num_attribs
test_num_attribs = list(test_numerical)
test_num_attribs
```

Years - Grouping and One Hot Encoding

We decided to one-hot encode year. Given the number of years represented in the dataset this would have resulted in many additional features. To reduce this, we first reduced years down to their respective decade and then one-hot encoded this transformed feature. We then created a function that rounds every year down to the starting year of its decade. This grouped years into decades to reduce the number of features in the year attribute. After one hot encoding the manipulated decades feature, we dropped the '1940s' decade from both the test and train sets to avoid the dummy variable trap.

```
train_years = train_data[['year']]
test_years = test[['year']]
```

```
def convert_to_decade(year):
    return (year // 10) * 10
train_decades = train_years[['year']]
train_decades['decade'] = train_decades['year'].apply(convert_to_decade)
train_decades = train_decades.drop(['year'], axis=1)
train_decades
```

```
test_decades = test_years[['year']]
test_decades['decade'] = test_decades['year'].apply(convert_to_decade)
test_decades = test_decades.drop(['year'], axis=1)
test_decades
```

```
###ordinal encoding train data
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()
train_decades_encoded = ordinal_encoder.fit_transform(train_decades)
train_decades_encoded
```

```
train_decade_cat = ordinal_encoder.categories_
train_decade_cat
```

```
###for test...
ordinal_encoder = OrdinalEncoder()
test_decades_encoded = ordinal_encoder.fit_transform(test_decades)
test_decades_encoded
```

```
test_decade_cat = ordinal_encoder.categories_
test_decade_cat
```

```
###one-hot encoding training decades
from sklearn.preprocessing import OneHotEncoder
cat_encoder = OneHotEncoder(sparse=False)
train_decades_1hot = cat_encoder.fit_transform(train_decades)
train_decade_categories = train_decades
train_decades_1hot
```

```
###one-hot encoding testing decades
test_decades_1hot = cat_encoder.fit_transform(test_decades)
test_decade_categories = test_decades
test_decades_1hot
```

```
###assigning correct columns to training encoded data
train_decade_enc = pd.DataFrame(train_decades_1hot)
train_decade_enc.columns = train_decade_cat[0]
train_decade_enc.index = train_data.index
train_decade_enc
```

```
###assigning correct columns to testing encoded data
test_decade_enc = pd.DataFrame(test_decades_1hot)
test_decade_enc.columns = test_decade_cat[0]
test_decade_enc.index = test.index
test_decade_enc


### DROPPING 1940 FROM the encoded datasets, to avoid the dummy trap
train_decade_enc.columns = train_decade_enc.columns.astype(str)
test_decade_enc.columns = test_decade_enc.columns.astype(str)
train_decade_enc = train_decade_enc.drop(['1940'],axis=1)
test_decade_enc = test_decade_enc.drop(['1940'], axis=1)


train_decade_enc.info()


test_decade_enc.info()
```

Genre - One Hot Encoding

Additionally, rather than dropping the nominal category of *'top genre'*, which we wanted to keep within the model's data, we recoded it to be represented numerically. One function we could have used was Sci-Kit Learn's Ordinal Encoder, which converts nominal values to numerical ones. However, given that the categories we want to convert are nominal, recoding them to numerical values within the same feature would potentially cause our model to detect relationships between instances that do not exist. Instead, we used Sci-Kit Learn's OneHotEncoding function. This creates a single binary feature per nominal category in the original feature (e.g. An attribute equal to 1 when the genre is *'adult standards'* and 0 when another genre). The code for our one-hot encoding of *'top genre'* in both the test and train is shown below.

```
train_genre = train_data[['top genre']]
test_genre = test[['top genre']]


###ordinal encoding genre for the train
ordinal_encoder = OrdinalEncoder()
train_genre_encoded = ordinal_encoder.fit_transform(train_genre)
train_genre_encoded


train_genre_cat = ordinal_encoder.categories_
train_genre_cat


###and for the test
ordinal_encoder = OrdinalEncoder()
test_genre_encoded = ordinal_encoder.fit_transform(test_genre)
test_genre_encoded


test_genre_cat = ordinal_encoder.categories_
test_genre_cat


###one hot encoding for the train
cat_encoder = OneHotEncoder(sparse=False)
train_genre_1hot = cat_encoder.fit_transform(train_genre)
train_genre_categories = train_genre
train_genre_1hot


###and for the test
test_genre_1hot = cat_encoder.fit_transform(test_genre)
test_genre_categories = test_genre
test_genre_1hot


###matching columns for the train
train_genre_enc = pd.DataFrame(train_genre_1hot)
train_genre_enc.columns = train_genre_cat[0]
train_genre_enc.index = train_data.index
train_genre_enc


###matching columns for the test
test_genre_enc = pd.DataFrame(test_genre_1hot)
test_genre_enc.columns = test_genre_cat[0]
test_genre_enc.index = test.index
test_genre_enc
```

However, one-hot encoding *'top genre'* resulted in an extremely large number of input features. The feature had over 300 unique values in the training data, which would have potentially slowed down our model if it remained in one-hot encoded form. To combat this, we manipulated the

one-hot encoded genre data into fewer categories to speed up training whilst still having genres as features our model could learn from. To reduce the feature numbers, we retained the top 3 most numerous genres from the test and training sets (which happpened to be the same three genres) and collapsed the remaining genre columns into one single column named 'Other Genres'. This reduced the number of genre columns down to 4. Finally, we dropped the feature 'dance pop' from both the train and test datasets. The code for collapsing the genre data and reducing the features is shown below.

```
train_top_genres = train_genre_enc[['adult standards', 'album rock', 'dance pop']]
train_top_genres
```

```
train_collapsed_data = train_genre_enc.drop(['adult standards', 'album rock', 'dance pop'], axis=1)
train_collapsed_data
```

```
start_column = 'acoustic blues'
end_column = 'yodeling'
start_index = train_collapsed_data.columns.get_loc(start_column)
end_index = train_collapsed_data.columns.get_loc(end_column)
```

```
train_collapsed_genre = train_collapsed_data.loc[:, start_column:end_column].sum(axis=1)
train_collapsed_data['Other Genres'] = train_collapsed_genre
train_collapsed_data
train_collapsed_genres = train_collapsed_data[['Other Genres']]
train_collapsed_genres
```

```
train_genre_collapsed = train_top_genres.join(train_collapsed_genres)
train_genre_collapsed
```

```
###doing the same for the test data
test_top_genres = test_genre_enc[['adult standards', 'album rock', 'dance pop']]
test_top_genres
```

```
test_collapsed_data = test_genre_enc.drop(['adult standards', 'album rock', 'dance pop'], axis=1)
test_collapsed_data
```

```
start_column = 'alternative country'
end_column = 'neo mellow'
start_index = test_collapsed_data.columns.get_loc(start_column)
end_index = test_collapsed_data.columns.get_loc(end_column)
```

```
test_collapsed_genre = test_collapsed_data.loc[:, start_column:end_column].sum(axis=1)
test_collapsed_data['Other Genres'] = test_collapsed_genre
test_collapsed_data
test_collapsed_genres = test_collapsed_data[['Other Genres']]
test_collapsed_genres
```

```
test_genre_collapsed = test_top_genres.join(test_collapsed_genres)
test_genre_collapsed
```

```
###dropping 'dance pop' from the encoded, collapsed datasets in order to avoid the dummy variable trap
train_genre_collapsed = train_genre_collapsed.drop(['dance pop'], axis=1)
train_genre_collapsed
```

```
test_genre_collapsed = test_genre_collapsed.drop(['dance pop'], axis=1)
test_genre_collapsed
```

Feature Scaling

As seen in the Figure 3 (above), for the numerical attributes of the training data, much of the data is placed along different scales. For example, in decibels ('dB') values can run from -25 to 0, whilst in beats per minute (BPM) values can run from 0 to 200. Scales of varying length can prove problematic for certain ML algorithms like Support Vector Machines (SVM) we decided to scale the numerical attributes of our data. To scale our numeric data, we employed Sci_Kit Learn's StandardScaler. StandardScaler utilises standardization, which involves subtracting the mean from values and then dividing by the standard deviation. This results in values that are not constrained to a specific range (such as in normalization, where values will end up between 0 and 1). Standardization was chosen because it is less sensitive to outliers than normalization. This was useful as certain features of our data, such as *'live'* and *'spch'*, have instances that could be considered outlying.

```
from sklearn import preprocessing
std_scaler = preprocessing.StandardScaler()
train_numerical[train_num_attribs] = std_scaler.fit_transform(train_numerical[train_num_attribs])
```

```
train_numerical
```

```
###additionally scaling the test data
test_numerical[test_num_attribs] = std_scaler.fit_transform(test_numerical[test_num_attribs])
```

```
test_numerical
```

After conducting a significant amount of preprocessing, we needed to rejoin our numerical and catergorical back together to create our final training and test sets that would be applied to our models.

```
train_prepped = train_numerical.join(train_genre_collapsed)
train_prepped
```

```
test_prepped = test_numerical.join(test_genre_collapsed)
test_prepped
```

```
train_prepped = train_prepped.join(train_decade_enc)
train_prepped
```

```
test_prepped = test_prepped.join(test_decade_enc)
test_prepped
```

```
train_prepped_FINAL = train_prepped
train_prepped_FINAL
```

```
test_prepped_FINAL = test_prepped
test_prepped_FINAL = test_prepped
```

Building Models

After preparing our data, we then had to choose the various ML model(s) to use in our regression problem. We also had to paramatize these models, and potentially include them in ensemble methods.

Support Vector Machine (SVM)

The first model we built and evaluated was a support vector machine (SVM). An SVM regressor (SVR) was chosen because of two reasons. First, SVMs are useful when dealing with small-to-medium sized datasets and we thought that this could also apply to our regression problem given that our training data contained less than 500 rows. Second, SVMs are capable of dealing with both linear and nonlinear data. This versatility meant that we could also apply polynomial features to our data if necessary.SVR operates on the opposite objective to SVM classification. Rather than attempting to limit margin violations within the margin and using the street to divide the dataset, SVR attempts to fit as many datapoints within the street whilst limiting margin violations outwith the margin. A SVR model was built and fit to the training data. The model and evaluation metric (RMSE) are coded below.

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon = 1.5)
svm_reg.fit(train_prepped_FINAL, train_target)
```

```
from sklearn.metrics import mean_squared_error
y_pred_svm = svm_reg.predict(train_prepped_FINAL)
mse_svm = mean_squared_error(train_target, y_pred_svm)
rmse_svm = np.sqrt(mse_svm)
rmse_svm
```

The RMSE score implies that the model, on average, is predicting a songs popularity at approximately 10 points away from the correct popularity. Although not a bad first attempt, it is evident that parameter tuning or another model would be required to achieve a lower RMSE.

Random Forest

Random forests are an ensemble of decision trees trained via bagging method. We chose to explore the use of a Random Forest for many reasons. Firstly, it allowed us to test the effect of a decision tree regressor and bagging regressor on our data in a more efficent way. Further, random forests also provide a generally better overall model due to searching for the best feature amongst a random subset of features. Finally, despite their simplicity, Random Forests are one of the most powerful predicitive algorithms available, which makes it an ideal model to explore.

```
from sklearn.ensemble import RandomForestRegressor
Rf_reg = RandomForestRegressor()
Rf_reg.fit(train_prepped_FINAL, train_target)


Rf_reg_ypred = Rf_reg.predict(train_prepped_FINAL)
Rf_reg_mse = mean_squared_error(train_target, Rf_reg_ypred)
Rf_reg_rmse = np.sqrt(Rf_reg_mse)
Rf_reg_rmse
```

The Random Forest Regressor has given us our best score yet. This may be because Random Forests employ bagging, which allows for training instances to be used several times. This could have been beneficial due to the (relatively) small size of our training data and may have allowed for a reduction in both bias and variance when compared to a single model used on its own. With a low score using the default parameters, our next course of action is to find the best parameters for the data to see if it improves our RMSE score.

Polynomial Regression

As shown above (Figure 1), our data was incredibly noisy and not as straightforward as predicting results on a simple straight line. Thus, it was of interest to investigate the effect of creating polynomial features by adding powers of each feature as new features. This allows the model to find relationships between features that a simple Linear Regression is not cabable of.

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(train_prepped_FINAL)


from sklearn.linear_model import LinearRegression
poly_lin_reg = LinearRegression()
poly_lin_reg.fit(X_poly, train_target)


y_pred_poly = poly_lin_reg.predict(X_poly)
poly_mse = mean_squared_error(train_target, y_pred_poly)
poly_rmse = np.sqrt(poly_mse)
poly_rmse
```

The polynomial regression model has a lower rmse than the SVM, indicating that that polynomial features have performed well with the non-linear data, and reduced the RMSE. However, it is still nearly double the score of that produced by the random forest. Despite, the polynomial features it has still not produced a lower score than the random forest, however keeping in mind that poly features does reduce the RMSE will be useful when investigating ensemble methods. The results of our individual models have provided a good starting point for further investigation. With the random forest performing best, it is of interest to see how it performs with different parameters and within ensemble methods.

Finding the Best Parameters

Despite finding a fairly low RMSE score for the Random Forest Regressor, this was using the default parameters. To find the parameters which fit our data best we utilised the GridSearchCV() function to ensure that our model performed to the best extent. The parameters to search were obtained from Sci-Kit Learn.

```
from sklearn.model_selection import GridSearchCV
parameters = {'n_estimators': [1, 100],'max_depth': [1, 100],'min_samples_leaf': [1, 100],'max_leaf_nodes': [2, 100],
    'n_jobs':[1, 100]}
rf_reg2 = RandomForestRegressor()
reg = GridSearchCV(rf_reg2, parameters, error_score='raise')
reg.fit(train_prepped_FINAL, train_target)
print("Best parameters found:", reg.best_params_)


### re-run models to see if it improves
rf_reg2 = RandomForestRegressor(max_depth =100, max_leaf_nodes= 100, min_samples_leaf= 1, n_estimators= 100, n_jobs= 100)
rf_reg2.fit(train_prepped_FINAL, train_target)
rf_reg2_ypred = rf_reg2.predict(train_prepped_FINAL)
rf_reg2_mse = mean_squared_error(train_target, rf_reg2_ypred)
rf_reg2_rmse = np.sqrt(rf_reg2_mse)
rf_reg2_rmse
```

The model run on the training set does not improve with the parameters found by the grid search, therefore indicating that the default parameters perform better. This could be a result of overfitting, which means the model performs well on the training data but generalises poorly to new instances. However, when uploading to kaggle our RMSE decreased by 0.01. Therefore, when assigning these default parameters, the model is better at generalising, which is ideal for when unseen data is applied to the model. As our strongest model so far, we decided to apply ensemble methods to see if it reduced the RMSE further.

Ensemble Methods

Ensemble methods aim to aggregate the predicions of models to get better predictions than an individual predictor. Ensembles are found to work best when the models used are as independent from each other as possible, as different models are likely to make different types of errors and improve the ensembles overall accuracy. To explore the effects of ensembles in an attempt to lower our RMSE score, we created a boosted model and a stacked model to see if we could produce a better regressor.

Boosting

Boosting aims to combine a few weak learners into a singular strong learner by training each model sequentially where each model learns from the mistakes of its predecessor. This should create a more accurate predictor with a lower RMSE. As our original models performed fairly poorly, combining them alongside a strong random forest regressor could create a strong model.

A Gradient Boosting Regressor (GBR) utilises boosting as described above, adding models successively in order to create a stronger overall model. However, rather than increase the importance of underfit training instances which is used by boosting models such as AdaBoost, GBR focuses on the residuals of the component models.

```
from sklearn.ensemble import GradientBoostingRegressor
gbr = GradientBoostingRegressor()
gbr.fit(train_prepped_FINAL, train_target)
gbr_ypred = gbr.predict(train_prepped_FINAL)
gbr_mse = mean_squared_error(train_target, gbr_ypred)
gbr_rmse = np.sqrt(gbr_mse)
gbr_rmse
```

The Gradient Boosting Regressor performed significantly better than the linear SVR and linear regression models on their own, however was still not as strong as the Random Forest Regressor on its own. Gradient Boosting Regressor shares some parameters with the Random Forest regressor. We ran a second GBR with the parameters set in our second Random Forest Regressor, to see if that would improve performance.

```
gbr = GradientBoostingRegressor(max_depth = 100, n_estimators = 100)
gbr.fit(train_prepped_FINAL, train_target)
gbr_ypred = gbr.predict(train_prepped_FINAL)
gbr_mse = mean_squared_error(train_target, gbr_ypred)
gbr_rmse = np.sqrt(gbr_mse)
gbr_rmse
```

An RMSE this low potentially indicates that the second GBR model is dramatically overfitting the data. Even though a Random Forest Regressor with similar parameters was performing relatively well, this was not the case with boosting. This may be because the bagging method employed by a RF Regressor allows it to generalize to data better.

Stacking

With success on our individual models only found in the random forest model, we attempted to stack our models to aggagate the predictions. Stacking takes several model's predictions and uses them to make a final prediction utilising a *meta learner*. The training data is split into subsets, with one being used to train the component predictors. The component predictors then predict on the second subset of the training data, and these predictions are fed into a new set as features. The meta learner is then trained on this resulting dataset. Our base models were those originally examined indivdually, again for the reasons discussed above. These are very different models, which would allow them to make different errors from which the meta regressor could learn from and to make accurate predictions. For our 'meta' regressor, we selected a the random forest regressor to predict the songs popularity. As our best performing model, it made sense to have this as the meta learner.

```
from sklearn.ensemble import StackingRegressor
from sklearn.linear_model import LinearRegression
base_models = [
    ('svm', LinearSVR()),
    ('gradient_boosting', GradientBoostingRegressor()),
    ('lin_regressor', LinearRegression())]
meta_regressor = RandomForestRegressor(max_depth = 100 , max_leaf_nodes= 100, min_samples_leaf= 1, n_estimators= 100, n_jobs=
stacked_model = StackingRegressor(estimators=base_models, final_estimator=meta_regressor)
stacked_model.fit(train_prepped_FINAL, train_target)
```

```
stacked_ypred = stacked_model.predict(train_prepped_FINAL)
stacked_mse = mean_squared_error(train_target, stacked_ypred)
stacked_rmse = np.sqrt(stacked_mse)
stacked_rmse
```

Despite, stacking different models, it has not improved our RMSE score and has in fact increased it. This may imply that the model is not learning from its predessors and is struggling to make more accurate predictions. As shown earlier, using polynomial features decreased the RMSE score, so it may be useful try this in the stacked regressor to see if this reduces the RMSE.

```
from sklearn.ensemble import StackingRegressor
from sklearn.linear_model import LinearRegression
base_models2 = [
    ('svm', LinearSVR()),
    ('gradient_boosting', GradientBoostingRegressor()),
    ('lin_regressor', LinearRegression())]
meta_regressor2 = RandomForestRegressor(max_depth = 100 , max_leaf_nodes= 100, min_samples_leaf= 1, n_estimators= 100, n_jo
stacked_model2 = StackingRegressor(estimators=base_models, final_estimator=meta_regressor)
stacked_model2.fit(X_poly, train_target)
```

```
stacked_ypred2 = stacked_model2.predict(X_poly)
stacked_mse2 = mean_squared_error(train_target, stacked_ypred2)
stacked_rmse2 = np.sqrt(stacked_mse2)
stacked_rmse2
```

Using a stacked model with polynomial features did improve the RMSE score, decreasing by approximately 3 from the non-polynomial stacked model. However, the score is still not as efficient as our simple random forest regressor. This could be because in our original stacked model, the component predictors, such as Linear Regression and SVR, were perhaps too simple for the data they were being fit to. Adding polynomial features improved their performance, but not as well as the RFR. Another reason for the disparity in performance may be because the RFR can already model nonlinear data, without the need for feature transformation, meaning it can operate on data with lower dimensionality.

Even with polynomial features added, a stacked model still underpeformed in comparison to our Random Forest. One reason may be that the stacking algorithm involves splitting the dataset into subsets. This may have been counterproductive in our case given that our training data already had a small number of instances.

Final Solution

After trying, testing and tuning a variety of models, it is evident from the RMSE scores that the Random Forest Regressor, with the parameters found via grid search, was the most accurate in producing the predicted popularity of the songs. Despite polynomial features performing well in previous models, Random Forests are already suited to handling non linear data without the need to add extra features. There, we decided to not create our final model with polynomial features. Although simple, the Random Forest Regressor has produced the most effective score to predict the popularity of songs. This implies that although ensemble methods can improve a models performance, it is not necessarily always true and depending on the type and structure of the data, some ensemble learning methods will perform better than others.The code from our final model is shown below, followed by the predictions that were used to submit to Kaggle.

```
final_model = RandomForestRegressor(max_depth =100, max_leaf_nodes= 100, min_samples_leaf= 1, n_estimators= 100, n_jobs= 10
final_model.fit(X_poly, train_target)
final_model_ypred = final_model.predict(X_poly)
final_model_mse = mean_squared_error(train_target, final_model_ypred)
final_model_rmse = np.sqrt(final_model_mse)
final_model_rmse
```

To assess our model's generalisability, we created a learning curve to assess our model's performance on training data versus validation data. To do this we had to split our existing training data into a test and train set, because we lack the test data's targets.

```
from sklearn.model_selection import learning_curve
from sklearn.model_selection import train_test_split
# Split the training data into training and testing sets because we don't have the test target
X_train, X_val, y_train, y_val = train_test_split(train_prepped_FINAL, train_target, test_size=0.2, random_state=42)
```

```
# Defining our Random Forest Regressor
rf_reg = RandomForestRegressor(max_depth=100, max_leaf_nodes=100, min_samples_leaf=1, n_estimators=100, n_jobs=100)
```

```
# Instantiate the learning curve
train_sizes, train_scores, test_scores = learning_curve(
    rf_reg, X_train, y_train, cv=5, scoring='neg_mean_squared_error', train_sizes=np.linspace(0.1, 1.0, 10))
```

```
# Calculating mean and standard deviation of training and validation scores
train_mean = -np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = -np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)
```

```
# Plotting learning curve
plt.figure(figsize=(10, 6))
plt.plot(train_sizes, train_mean, label='Training Error', marker='o')
plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.15, color='blue')
plt.plot(train_sizes, test_mean, label='Validation Error', marker='o')
plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std, alpha=0.15, color='green')
plt.title('Learning Curve')
plt.xlabel('Training Set Size')
plt.ylabel('Negative Mean Squared Error')
plt.legend()
plt.show()
```

The learning curve shows that the model is significantly overfitting to the split training data. This meant that our model is detecting patterns within the dataset and thus will not generalise well to new instances. It should be noted that this graph, whilst informative, is not plotted against the *actual test data*, because the test targets are not available. Whilst this does reduce the overall relevance of the learning curve in the context of our test data, it is still a potentially informative indicator of the performance of our regression model.

Kaggle Performance

```
final_kaggle = final_model.predict(X_poly)
test_data_set = pd.read_csv("CS98XRegressionTest.csv")
final_submission = pd.DataFrame(columns=['Id', 'pop'])
final_submission['pop'] = kaggle
final_submission['Id'] = test_data_set['Id']
excel_filename = 'Final_submission.xlsx'
final_submission.to_excel(excel_filename, index=False)
print(f'Data exported to {excel_filename}')
```

When uploading our final model to Kaggle, our RMSE score was: 7.36.