CS986 Spotify Regression Problem 2025

Introduction

In this project, our goal was to build a machine-learning model capable of addressing the regression task of predicting the popularity scores of songs. We used a training dataset focused on Spotify songs from the past few decades; excluding ID and popularity, 13 attributes were included such as genre, release year, beats, loudness, and valance. These attributes guided the model-building process and facilitated effective predictions of song popularity on the testing dataset. A range of techniques was applied, from basic models for baseline comparisons (e.g. Linear, Lasso) to more sophisticated approaches (e.g. Random Forest, CatBoost).

Library

In [11]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder, PowerTransformer, MinMaxScaler, StandardSc
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_selection import SelectKBest, f_regression, RFE
from sklearn.linear_model import Lasso
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, StackingReg
import xgboost as xgb
from catboost import CatBoostRegressor
from datetime import datetime
```

Data exploration

Load data

In [ ]:
```
Reg_test = pd.read_csv("CS98XRegressionTest.csv")
Reg_train = pd.read_csv("CS98XRegressionTrain.csv")
```

Two datasets were used for regression to predict a song's popularity score. The training set has 453 rows and 15 columns, including the target variable pop, and the identification variable, Id. The testing set has 114 rows and 14 columns, excluding pop for prediction.

Preprocessing

Missing value

- **Top genre**: Found 15 missing values in the training set and 1 in the testing set. Missing data was replaced with "Unknown" as a separate category, by using Label Encoding.

In [ ]:
```
Reg_train['top genre'] = Reg_train['top genre'].fillna('Unknown')
Reg_test['top genre'] = Reg_test['top genre'].fillna('Unknown')
```

Skewness

Skewness measures deviation from a normal distribution, impacting model performance. Highly skewed features can reduce prediction accuracy. We calculated skewness for numerical features: year, bpm, nrgy, dnce, dB, live, val, dur, acous, and spch.

In [ ]:
```
numerical_cols = Reg_train.select_dtypes(include=['number']).columns.drop(['pop', 'Id'])
skewness_values = Reg_train[numerical_cols].skew().sort_values(ascending=False)
```

We found spch and live have Highly right-skewed and need Log transformation

In [ ]:
```
highly_skewed_cols = skewness_values[skewness_values > 1.5].index
Reg_train[highly_skewed_cols] = np.log1p(Reg_train[highly_skewed_cols])
Reg_test[highly_skewed_cols] = np.log1p(Reg_test[highly_skewed_cols])
```

We found dur and acous have Moderately Skewed Features to handle skewness. We applied Cube Root Transformation to smooths out skewness without distorting data.

In [ ]:
```
moderately_skewed_cols = skewness_values[(skewness_values > 0.5) & (skewness_values <= 1.5
Reg_train[moderately_skewed_cols] = np.cbrt(Reg_train[moderately_skewed_cols])
Reg_test[moderately_skewed_cols] = np.cbrt(Reg_test[moderately_skewed_cols])
```

Yeo-Johnson Transformation was applied to all numerical columns, regardless of skewness, to ensure a more stable and normally distributed dataset. Even if some columns appeared normally distributed, this transformation helped handle slight deviations and improve model performance.

In [ ]:
```
pt = PowerTransformer(method='yeo-johnson')
Reg_train[numerical_cols] = pt.fit_transform(Reg_train[numerical_cols])
Reg_test[numerical_cols] = pt.transform(Reg_test[numerical_cols])
```

Encode top genre

The Top Genre feature is categorical, requiring encoding for the model to interpret it. Label Encoding was chosen for its lower memory usage compared to One-Hot Encoding. The encoder was fitted on known categories in the training set, while unseen categories in the test set were assigned -1.

In [ ]:
```
label_encoder = LabelEncoder()
Reg_train['top genre_encoded'] = label_encoder.fit_transform(Reg_train['top genre'])
Reg_test['top genre_encoded'] = Reg_test['top genre'].apply(lambda x: label_encoder.transf
label_mapping = dict(zip(label_encoder.classes_, label_encoder.transform(label_encoder.cla
```

Create new columns for year

The year column shows the release year of a song. We compute song_age by subtracting the current year and release year by assuming that newer songs may have different popularity trends compared to older songs.

In [ ]:
```
current_year = datetime.now().year
Reg_train['song_age'] = current_year - Reg_train['year']
Reg_test['song_age'] = current_year - Reg_test['year']
```

Create new columns for artist

Artist names cannot be used in ML. We suggest that more frequent artists might have a higher impact on popularity trends. Therefore, the artist_freq was created to represents how many times an artist appears in the dataset.

In [ ]:
```
artist_counts_train = Reg_train['artist'].value_counts()
artist_counts_test = Reg_test['artist'].value_counts()
Reg_train['artist_freq'] = Reg_train['artist'].map(artist_counts_train)
Reg_test['artist_freq'] = Reg_test['artist'].map(artist_counts_test)
```

We also convert it into numerical values using Label Encoding for using in ML. For unseen Artists in Reg_test, the -1 was assign to avoid errors.

In [ ]:
```
label_encoder = LabelEncoder()
Reg_train['artist_encoded'] = label_encoder.fit_transform(Reg_train['artist'])
Reg_test['artist_encoded'] = Reg_test['artist'].apply(lambda x: label_encoder.transform([x
```

Create new columns for Title

The title column is text based to make this feature useful for ML models, we apply TF-IDF to captures word importance and reduces dimensionality.

In [ ]:
```
vectorizer = TfidfVectorizer(max_features=50)
title_tfidf_train = vectorizer.fit_transform(Reg_train['title'].astype(str)).toarray()
title_tfidf_test = vectorizer.transform(Reg_test['title'].astype(str)).toarray()
title_tfidf_train = pd.DataFrame(title_tfidf_train, columns=[f"title_{i}" for i in range(
title_tfidf_test = pd.DataFrame(title_tfidf_test, columns=[f"title_{i}" for i in range(ti
Reg_train = pd.concat([Reg_train.reset_index(drop=True), title_tfidf_train], axis=1)
Reg_test = pd.concat([Reg_test.reset_index(drop=True), title_tfidf_test], axis=1)
```

Check for outlier

Outlier might distort relationships in the dataset. So, the IQR was used to detecting outliers.

In [ ]:
```
cols = ['bpm', 'nrgy', 'dnce','dB', 'live', 'val', 'dur', 'acous', 'spch','song_age', 'ar
def detect_outliers_iqr(df, columns):
    outlier_summary = {}
    for col in columns:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        outliers = df[(df[col] < lower_bound) | (df[col] > upper_bound)]
        outlier_summary[col] = len(outliers)
    return pd.DataFrame.from_dict(outlier_summary, orient='index', columns=['Outlier Coun
outlier_counts = detect_outliers_iqr(Reg_train, cols)
```

The result found that artist_freq has a large number (48) and dur(4), bpm(3), dB(2) also live (1) have a small number of outliers. While nrgy, dnce, val, acous, spch, and song_age don't have extreme values.

Min-Max Scaling (0 to 1)

Different types of features require different scaling techniques based on their distribution and range. Min-Max Scaling preserves the original distribution while bringing values between 0 and 1. This works best for features with a fixed range and no extreme outliers. Then 'nrgy', 'dnce', 'val', 'acous','spch', and 'song_age' are applied for Min-Max Scaling.

In [ ]:
```
min_max_cols = ['nrgy', 'dnce', 'val', 'acous','spch','song_age']
minmax_scaler = MinMaxScaler()
Reg_train[min_max_cols] = minmax_scaler.fit_transform(Reg_train[min_max_cols])
Reg_test[min_max_cols] = minmax_scaler.transform(Reg_test[min_max_cols])
```

Standardisation (Z-score)

When data has outliers, Z-score Standardization is useful for normally distributed or highly variable data. 'live', 'dur', 'artist_freq', and 'dB' are applied for Standardisation.

In [ ]:
```
standardize_cols = ['live', 'dur', 'artist_freq','dB']
standard_scaler = StandardScaler()
Reg_train[standardize_cols] = standard_scaler.fit_transform(Reg_train[standardize_cols])
Reg_test[standardize_cols] = standard_scaler.transform(Reg_test[standardize_cols])
```

Feature Selection

In this feature selection step, pop is the target variable, we store it separately before removing it from Reg_train, ensuring we don't lose it. Afterward, dropping Id which is an identifier that doesn't provide useful information. Moreover, the title, and artist are not directly usable, we use the encoded one instead. Lastly, the pop column is dropped because it is the target variable that is already stored separately.

In [ ]:
```
y = Reg_train['pop']
drop_columns_train = ['Id', 'title', 'artist', 'top genre', 'pop']
drop_columns_test = ['title', 'artist', 'top genre']
Reg_train.drop(columns=drop_columns_train, inplace=True)
Reg_test.drop(columns=drop_columns_test, inplace=True)
```

Check how useful of title Encoding

The title column was transformed using TF-IDF, but not all features were useful. Variance analysis and correlation with pop were used to assess whether the encoded features had enough variation and impact on model predictions.

In [ ]:
```
title_features = [col for col in Reg_train.columns if 'title_' in col]
Variance_analysis_output = Reg_train[title_features].var().sort_values()
correlation_with_target = Reg_train[title_features].corrwith(y).abs().sort_values(ascendi
```

- Features with very low variance (<0.01) provide little variability.
- Features with a correlation >0.08 have stronger predictive power for pop.
- title_0 (0.1006) and title_1 (0.1046) were kept due to their high correlation.

In [ ]:
```
drop_title_features = [col for col in title_features if col not in ['title_0', 'title_1']]
Reg_train.drop(columns=drop_title_features, inplace=True)
Reg_test.drop(columns=drop_title_features, inplace=True)
```

Final Features

- For train set: Contains 16 features excluding the target variable pop
- Test Set: Contains 17 features, including Id as identifier. The Id column is present in Reg_test but not in Reg_train.

Correlation Analysis

Measures linear correlation between features & target (pop). This method is simple and interpretable for identifies relationships

In [ ]:
```
X = Reg_train
```

In [ ]:
```
correlation = X.corrwith(y).abs().sort_values(ascending=False)
correlation_threshold = 0.1
top_features_corr = correlation[correlation > correlation_threshold].index.tolist()
print("Selected Features (Correlation Analysis):", top_features_corr)
```

```
Selected Features (Correlation Analysis): ['dur', 'acous', 'dB', 'nrgy', 'dnce', 'artist_f
req', 'top genre_encoded', 'live', 'title_1', 'title_0']
```

SelectKBest (f_regression) Uses ANOVA F-statistic to select features that best explain target variance. This method is good for linear relationships and Fast computation

In [ ]:
```
# Select top 10 features based on f_regression
selector = SelectKBest(score_func=f_regression, k=10)
selector.fit(X, y)

# Get selected feature names
selected_features_f_reg = X.columns[selector.get_support()]
print("Selected Features (f_regression):", selected_features_f_reg)
```

```
Selected Features (f_regression): Index(['nrgy', 'dnce', 'dB', 'live', 'dur', 'acous', 'to
p genre_encoded',
       'artist_freq', 'title_0', 'title_1'],
      dtype='object')
```

Recursive Feature Elimination (RFE)

Recursive Feature Elimination (RFE) works by iteratively removing the least important features using a machine learning model.

In [ ]:
```
rf = RandomForestRegressor(n_estimators=100, random_state=42)
rfe = RFE(rf, n_features_to_select=10)
rfe.fit(X, y)
selected_features_rfe = X.columns[rfe.support_]
print("Selected Features (RFE):", selected_features_rfe)
```

```
Selected Features (RFE): Index(['year', 'bpm', 'nrgy', 'dB', 'dur', 'acous', 'top genre_en
coded',
       'song_age', 'artist_freq', 'artist_encoded'],
      dtype='object')
```

Random Forest Feature Importance Uses Random Forest to rank features based on how much they reduce prediction error. This method is good for handling non-linear relationships and works well for high-dimensional data.

In [ ]:
```
# Train Random Forest and get feature importances
rf.fit(X, y)
feature_importances = pd.Series(rf.feature_importances_, index=X.columns)

# Select top 10 features
top_features_rf = feature_importances.nlargest(10).index.tolist()
print("Selected Features (Random Forest Importance):", top_features_rf)
```

```
Selected Features (Random Forest Importance): ['dur', 'acous', 'nrgy', 'song_age', 'year',
'dB', 'artist_freq', 'top genre_encoded', 'bpm', 'artist_encoded']
```

Lasso Regression Feature Selection

Lasso Regression Feature Selection Uses L1 regularisation to shrink less important feature coefficients to zero.

In [ ]:
```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
lasso = Lasso(alpha=0.1)
lasso.fit(X_scaled, y)
selected_features_lasso = X.columns[lasso.coef_ != 0]
print("Selected Features (Lasso Regression):", selected_features_lasso)
```

```
Selected Features (Lasso Regression): Index(['year', 'bpm', 'nrgy', 'dnce', 'dB', 'live',
'val', 'dur', 'acous',
       'top genre_encoded', 'song_age', 'artist_freq', 'artist_encoded',
       'title_0', 'title_1'],
      dtype='object')
```

Compare Feature Selection Methods

After apply 16 Features to all methods above to determine the most important features.

In [ ]:
```
common_features = set(top_features_corr) & set(selected_features_rfe) & set(selected_featu
print("Selected Features (common features):", common_features)
```

```
Selected Features (common features): {'dB', 'acous', 'top genre_encoded', 'dur', 'nrgy',
'artist_freq'}
```

Common Features is the most important feature and consistent across methods

There are 4 metric were used to evaluate. There are MAE (Mean Absolute Error), MSE (Mean Squared Error),RMSE (Root Mean Squared Error), andR^2 Score (Coefficient of Determination)

In [ ]:
```
def evaluate_feature_selection(X, y, feature_sets):
    results = []
    for method, features in feature_sets.items():
        valid_features = list(features.intersection(X.columns))
        if not valid_features:
            continue
        X_selected = X[valid_features]
        X_train, X_test, y_train, y_test = train_test_split(X_selected, y, test_size=0.2,
        model = RandomForestRegressor(random_state=42)
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        mae = mean_absolute_error(y_test, y_pred)
        mse = mean_squared_error(y_test, y_pred)
        rmse = np.sqrt(mse)
        r2 = r2_score(y_test, y_pred)
        results.append({
            "Feature Selection Method": method, "MAE": mae, "MSE": mse, "RMSE": rmse, "R²
        })
    return pd.DataFrame(results)
feature_sets = {
    "Correlation Analysis": set(top_features_corr),"Recursive Feature Elimination (RFE)":
    "Lasso Regression": set(selected_features_lasso),"Common Features (Multiple Methods)"
}
```

In [ ]:
```
results_df = evaluate_feature_selection(X, y, feature_sets)
```

In [ ]:
```
results_df
```

Out[ ]:

	Feature Selection Method	MAE	MSE	RMSE	R² Score
0	Correlation Analysis	8.609231	122.813398	11.082121	0.450936
1	Recursive Feature Elimination (RFE)	8.429890	114.620719	10.706107	0.487563
2	Lasso Regression	8.566703	114.516696	10.701247	0.488028
3	Common Features (Multiple Methods)	8.569011	122.946719	11.088134	0.450340

- The best Feature Selection Methods is Recursive Feature Elimination (RFE) which has the Lowest RMSE (10.60)and the Highest R^2 (0.4970).
- The worst Feature Selection Methods is Correlation Analysis with Highest RMSE (11.13) and Lowest R^2 Score (0.4461)

Model Selection

Final features to use in model are bpm, dur, dB, top genre_encoded, song_age, year, artist_freq, artist_encoded, acous, and nrgy from RFE for better model performance. Then, split into train and test sets to train the model on one part and evaluate it on another to avoid overfitting and use random_state=42, to ensure consistent results every time the code runs.

In [ ]:
```
X = Reg_train[selected_features_rfe]
X_test_final = Reg_test[selected_features_rfe]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Evaluation Function

The key evaluation metrics were calculated to evaluate the models. There are MAE (Mean Absolute Error), MSE (Mean Squared Error), RMSE (Root Mean Squared Error), and R^2 Score (Coefficient of Determination).

In [ ]:
```
def evaluate_model(model, X_train, X_test, y_train, y_test, model_name):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    mae = mean_absolute_error(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_test, y_pred)
    result = pd.DataFrame([{"Model": model_name,"MAE": mae,"MSE": mse,"RMSE": rmse,"R² Sc
    print(f"\n Performance of {model_name}:")
    print(result)
```

Baseline

Linear Regression

Linear Regression is the simplest model that assumes a linear relationship between features and target. It is fast and easy to interpret but Fails on non-linear data.

In [ ]:
```
lr_model = LinearRegression()
evaluate_model(lr_model, X_train, X_test, y_train, y_test, "Linear Regression")
```

```
 Performance of Linear Regression:
               Model       MAE         MSE       RMSE  R² Score
0  Linear Regression  9.454644  134.907251  11.614958  0.396868
```

Lasso Regression

Lasso Regression is similar to Ridge, but also performs feature selection by shrinking some coefficients to zero. This model prevents overfitting but performs poorly if features are highly correlated.

In [ ]:
```
lasso_model = Lasso(alpha=0.1)
evaluate_model(lasso_model, X_train, X_test, y_train, y_test, "Lasso Regression")
```

```
Performance of Lasso Regression:
              Model       MAE         MSE       RMSE  R² Score
0  Lasso Regression  9.475763  134.905331  11.614875  0.396877
```

Decision Tree Regression

Decision Tree Regression is a simple tree-based model that captures non-linear relationships in data. This model Handles both linear & non-linear data and works well with missing values. However,it is overfits easily.

In [ ]:
```
dt_model = DecisionTreeRegressor(random_state=42)
evaluate_model(dt_model, X_train, X_test, y_train, y_test, "Decision Tree Regression")
```

```
Performance of Decision Tree Regression:
                      Model        MAE         MSE       RMSE  R² Score
0  Decision Tree Regression  11.406593  209.054945  14.458732  0.065375
```

Advance

Random Forest Regression

Random Forest Regression is an ensemble of decision trees that reduces overfitting by averaging multiple predictions, it handles non-linearity well and is less sensitive to outliers.

In [ ]:
```
rf_model = RandomForestRegressor(n_estimators=200, random_state=42)
evaluate_model(rf_model, X_train, X_test, y_train, y_test, "Random Forest Regression")
```

```
Performance of Random Forest Regression:
                      Model       MAE         MSE       RMSE  R² Score
0  Random Forest Regression  8.408846  114.088122  10.681204  0.489944
```

Gradient Boosting Regression

Gradient Boosting Regression sequentially builds trees, focusing on correcting previous errors to improve accuracy.

In [ ]:
```
gb_model = GradientBoostingRegressor(n_estimators=200, random_state=42)
evaluate_model(gb_model, X_train, X_test, y_train, y_test, "Gradient Boosting Regression")
```

```
Performance of Gradient Boosting Regression:
                          Model       MAE         MSE       RMSE  R² Score
0  Gradient Boosting Regression  8.449046  109.791038  10.478122  0.509155
```

CatBoost Regression

CatBoost Regression is a Gradient Boosting model optimized for categorical data. It required less hyperparameter tuning.

In [ ]:
```
cat_model = CatBoostRegressor(iterations=200, learning_rate=0.1, depth=5, verbose=0, rando
evaluate_model(cat_model, X_train, X_test, y_train, y_test, "CatBoost Regression")
```

```
Performance of CatBoost Regression:
                Model       MAE       MSE      RMSE  R² Score
0  CatBoost Regression  7.898911  96.79184  9.838284  0.567271
```

Stacking Regression

Stacking Regression is a meta-learning approach that combines multiple models to improve performance. This can leverage multiple models' strengths, however is complex to implement and is computationally expensive.

In [ ]:
```
base_models = [
    ('rf', RandomForestRegressor(n_estimators=200, random_state=42)),
    ('gb', GradientBoostingRegressor(n_estimators=200, random_state=42)),
    ('xgb', xgb.XGBRegressor(n_estimators=200, learning_rate=0.1, max_depth=5, random_sta
    ('cat', CatBoostRegressor(iterations=200, learning_rate=0.1, depth=5, verbose=0, rando
]
meta_model = Ridge(alpha=1.0)
stacking_model = StackingRegressor(estimators=base_models, final_estimator=meta_model)
evaluate_model(stacking_model, X_train, X_test, y_train, y_test, "Stacking Regression")
```

```
Performance of Stacking Regression:
                Model       MAE        MSE      RMSE  R² Score
0  Stacking Regression  7.887805  99.633482  9.981657  0.554567
```

In [ ]:
```
X_test_final = Reg_test[selected_features_rfe]
```

Final Model and Implementation

As a result, the Lasso regressor performs as a strong baseline model, whereas CatBoost delivers superior performance for challenging predictions. Lasso regression produced the lowest RMSE of 11.61 and an R^2 score of 0.396 amongst other baseline models. It acts as an effective starting point as it identifies important features in predicting song popularity while minimising overfitting. However, it may not capture the complexity of data patterns.

Therefore, a final advanced model was chosen - CatBoost (Categorical Boosting) Regressor. It is an ensemble learning approach built on a gradient-boosting algorithm that combines multiple models to handle categorical data efficiently. It was chosen because it produced the lowest RMSE of 9.83 and an R^2 score of 0.567 compared to other advanced models, effectively capturing patterns in the dataset while reducing prediction errors. A mix of numerical attributes such as loudness and energy and categorical features such as top genre predicts song popularity. CatBoost's ability to automatically handle different data types and its strong performance metrics allowed a more streamlined and accurate model.

A further reason the CatBoost model might have performed so well in comparison to baseline models is its ability to handle non-linear data. Some of the variables within the dataset are non-linear such as danceability or energy, meaning that simpler models like LASSO regression struggle to provide accurate predictions.

Once the Lasso and CatBoost models were trained, it was then applied to the test dataset to produce final predictions of the song's popularity.

- The best model as the baseline model is Lasso Regression with lowest RMSE(11.6149) with highest R^2 (0.3969)
- The best model as the advance model is CatBoost Regression with lowest RMSE(9.8383) with highest R^2 (0.5673)

Lasso Regression is the best model for Basic

In [ ]:
```
lasso_model.fit(X_train, y_train)
y_pred_test = lasso_model.predict(X_test_final)
```

In [ ]:
```
submission_df_lasso_model = pd.DataFrame({"Id": Reg_test["Id"],  "pop": y_pred_test })
```

CatBoost Regression is the best model for Advanced

In [ ]:
```
cat_model.fit(X_train, y_train)
y_pred_test = cat_model.predict(X_test_final)
```

In [ ]:
```
submission_df_cat_model = pd.DataFrame({ "Id": Reg_test["Id"], "pop": y_pred_test})
```

Insights from the Model

As it has been outlined, the CatBoost model was the best-performing model created for this project. Its low RMSE means that the predicted values from the model were very similar to the actual values provided with the model not overfitting and maintaining generalisation. As well as the tuning of the models through the use of methods such as feature selection we were further able to improve the predictive capabilities of our regression model. More specifically the feature selection processes such as Recursive Feature Elimination (RFE) and Random Forest Importance were proven to be the most effective with RMSE scores of 10.54 respectively.

The CatBoost Regressor outperformed the baseline models with an RMSE of 9.84 and an improved RMSE of 7.5 on the Kaggle test dataset. This demonstrates that our model generalised very well to the unseen samples based on this reduction in RMSE. The model successfully captured patterns from the main dataset proving that our steps taken to build the model ie feature selection as well as model selection were successful. However, it should be noted that this RMSE score from the Kaggle dataset is only based on 50% of the test data. The final result may vary from this 7.5 as the entire dataset was not fully utilised. Nonetheless, a positive outcome overall.

Concluding Remarks

Overall, the CatBoost regression model created was very successful in predicting the popularity score of a song. The feature selection process showed that the RFE and Random Forest Importance were the most effective at helping narrow down the attribute selection. Not only this but the process of trialling several different regression models and comparing their RMSE score allowed for a wide range of models to be analysed and broken down. These two main processes undertaken proved to be very effective as the RMSE score from the Kaggle leaderboard was 7.50631, much lower than the validation set meaning our model fit the data very well with minimal overfitting.

A downside to this process was the time taken to create all these separate models. The selection of the models and subsequent fine-tuning of them was a very time-consuming process with the majority of the models not even being used in the end when their RMSE score was calculated to be high. It would have been much easier and faster to simply create one model from the start and stick with it. However, we would not have been able to compare different models and thus perhaps create a sub-optimal algorithm.