STAT230A Final Project: Playlist Membership Prediction for Spotify Tracks

Matthew Seguin
Department of Statistics
University of California, Berkeley
Berkeley, CA 94720
mmseguin2003@berkeley.edu

Ozi Anyanwu
Department of Statistics
University of California, Berkeley
Berkeley, CA 94720
ozi_anyanwu@berkeley.edu

1 Introduction

Avid music listeners typically have a few hand-crafted playlists of songs they enjoy. Typically, the songs in a playlist share some common characteristics. At times, the commonality is clear and precise (e.g. a rap playlist). Other times, the commonality is more difficult to detect. Instead of grouping by genre, artist, or era, a listener may choose to make a playlist of songs that elicit a particular feeling.

Our project will attempt to answer the following question: Given a well-curated Spotify playlist, how effectively can we identify songs that belong in it? An answer to this question could inform the development of automated song-recommendation systems that are tailored to users' specific playlists. We build logistic regression models for a few different playlists to address this question.

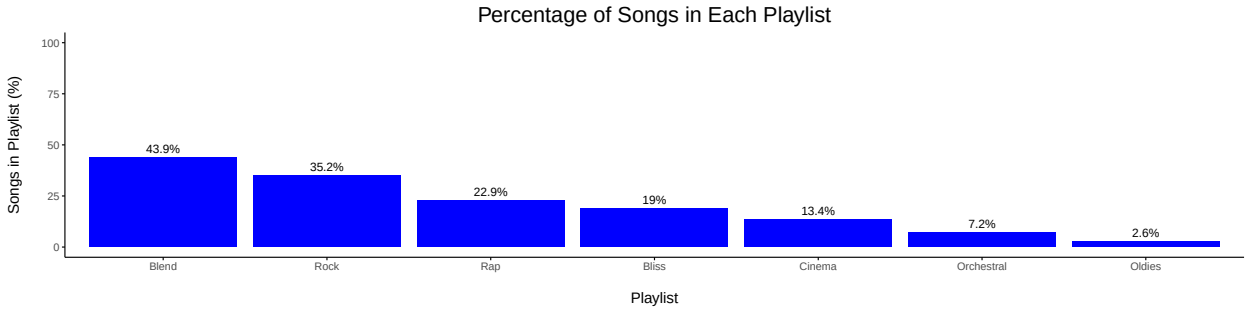
2 Data

The data used in this project comes from our own Spotify playlist and contains song characteristics such as playtime, loudness, popularity, instrumentalness, and danceability to name a few. We first used the Spotify API to retrieve basic information like song names and artists for all of our songs and playlists. Spotify has deprecated its audio feature tool so we had to find an alternative means of retrieving the important song metrics like loudness, popularity, instrumentalness, etc. We reached out to the developer of the website musicstax.com and were able to retrieve most of the data we needed. For the remaining data we were able to webscrape. The code used to collect the basic information from the Spotify API as well as the code used to webscrape any missing data is included but is not runnable without a Spotify API developer key.

Minimal data processing was required to make the raw data into a format useable for logistic regression. First, we featurized the data by adding a membership indicator for each playlist and removing duplicate songs. We then split the key signature column into two variables, key (e.g. A#) and mode (Major or minor). We then used one-hot encoding to split time signature into multiple columns but still kept the original column in case we decide to use it as a non-categorical variable. After that we changed data types to useable formats; Converting the key variable into the frequency (Hz) of the note in the fourth octave (e.g. A# would be 466.16 Hz) and changing data types where necessary. Finally, some of the release dates only had the year of release and not the exact day. In order to fix this we assumed that all songs which had only the year were released on January 1 of that year. We then converted release date to age of the song in months. There were no songs with only the year and month.

The variables are named well but more detailed descriptions can be found in the appendix on the Spotify Developer Tools Page. We did not treat key as categorical so we did not perform one-hot encoding on it. However, we visualized it in the same way as other categorical variables in our exploratory analysis due to the low number of levels.

We used 514 songs for model selection and 128 songs for evaluation of our best models. Below is a graph showing what percentage of songs are in each of our playlists for the full data.



3 Final Models

We present three final models. One for a playlist we expected to have very high predictability (Rap), one for a playlist we expected to have decent predictability (Blend), and one for a playlist we expected to have low predictability (Bliss). To arrive at our best models, we created a 5-fold cross validation environment to serve as our model selection criterion. We used F1 score as the evaluation metric. We began with a parsimonious approach where we hand-selected features that seemed salient based on EDA we performed. We then tried a more systematic approach by performing best feature-subset selection to decide which subset of features leads to the highest cross-validation scores. We applied several heuristics to make this subset selection run faster, such as enforcing the presence of certain features, and placing a minimum number of features allowed. We then tried a more modern approach by using Lasso regularization to automatically hand-pick features by omitting features whose coefficients were set to zero. We eventually found that applying best subset selection, and then regularizing those features with Lasso led to the best cross-validation scores. Our results are consistent with or expectations. The Rap playlist had the best predictability, the Blend playlist came in second, and the Bliss playlist was last. The results are presented in Tables 4, 5, and 6 of the Appendix.

In unpenalized logistic regression models, taking the exponent of a coefficient gives you the multiplicative increase in the odds ratio for a one unit increase in the corresponding predictor. That is

$$\exp \beta_j = \frac{\text{odds}(y_i = 1 \mid \dots, x_{ij} = t+1, \dots)}{\text{odds}(y_i = 1 \mid \dots, x_{ij} = t, \dots)}$$

This means that if a coefficient is negative we expect it to be less likely that $y_i = 1$ when that predictor increases and similarly if a coefficient is positive we expect it to be more likely that $y_i = 1$ when that predictor increases. Below are the coefficients for each model.

Feature	Coefficient
Explicit	2.0293496
Age	-1.8447523
Danceability	1.7853854
Speechiness	1.5895343
Time Signature 5	-1.2230394
Playtime	0.6716019
Energy	-0.6596255
Time Signature 3	-0.3547538
Time Signature 1	0.2560657
Mode	-0.1347241

Table 1: Coefficients for Rap Playlist

Feature	Coefficient
Instrumentalness	-1.5988264
Explicit	-0.7397082
Positiveness	-0.5508191
Acousticness	-0.5034825
Energy	-0.3841088
Popularity	0.3600873
Key	0.1441046
Speechiness	0.1034590
Tempo	-0.0205006

Table 2: Coefficients for Blend Playlist

Feature	Coefficient
Speechiness	-2.0158168
Instrumentalness	-1.7093763
Explicit	-1.1921352
Danceability	0.9614139
Positiveness	-0.7700187
Time Signature 1	-0.4825381
Time Signature 3	0.3228491
Liveliness	-0.2668071
Loudness	-0.2300533
Time Signature 5	0.1815700
Popularity	0.1318618
Key	-0.0930573
Energy	0.0212633

Table 3: Coefficients for Bliss Playlist

4 Discussion

With the Lasso, the above interpretation of coefficients as the multiplicative factor on odds ratio breaks down. But since we standardized our features, the coefficients can still indicate relative importance of various features. Features with higher magnitude coefficients will be more salient.

For the Rap playlist, we inspect the top four features. "Explicit" indicates whether or not a song contains explicit language. "Age" measures how recently the song was released. "Speechiness" refers to the presence of spoken words in a track, and "danceability" is quantifies how suitable a track is for dancing, ranging from 0.0 (least danceable) to 1.0 (most danceable). These are all features we would expect to be important in identifying rap songs. Rap songs tend to be explicit, more recent, have a lot of lyrics, and are very suitable for dancing. We also bring attention to the negative sign on the age coefficient, which indicates that an increase in the age of a song decreases the odds that it belongs in the Rap playlist.

Blend and Bliss are far less predictable than Rap. It therefore comes as no surprise that our best-subset selection algorithm chose more features for these two playlists than for the Rap playlist (counting the Time Signature as categorical). It is natural that the model would need to look at more features to come up with a good decision boundary for playlists that are not genre-specific. We notice that the coefficient on the "Explicit" feature is negative in the Bliss playlist, which makes sense since a playlist eliciting feelings of delight and elation is less likely to have songs with with profanity. Speechiness had a positive coefficient in every model except for Blend. We attribute this to the fact that Blend contains a larger breadth of songs, and therefore includes many songs with little to no vocals or lyrics.

One limitation of our work in predicting songs for the Blend and Bliss playlists is a lack of expressive feature transformations for the numeric features. Adding splines or polynomial expansions may help carve out a better decision boundary. We leave that as future work.

There is an even greater limitation that affects all three playlist models. From a song listener's perspective, a very important determinant for whether or not a song makes into any of their playlists is whether or not they like the song in the first place. This is essentially a pre-requisite. If a song clearly has all the features of a rap song, is still does not mean the user will add it to their Rap playlist. This is one major area our analysis could be extended. Our dataset did not include examples of rap songs that the curator chose not to add to their "Rap" playlist. And so our models are currently not directly built to capture how much a user like a particular song.

5 Conclusion

Our analysis shows that whether or not we can identify if a song belongs in a well-curated playlist highly depends on the nature of the playlist. Playlists made up of songs with highly identifiable features like genre are much easier to make recommendations for than playlists whose songs have less tangible commonalities. For the latter case, we may still be able to build. recommendation systems if we find expressive transformations of the audio features and have a lot more data. In either case, any good song recommendation system will also need to capture information as to whether or not a user will enjoy a song in the first place, before even deciding if it has the qualities to belong in particular playlist.

6 Additional Work

There are many ways to assess model performance. We opted to focus on the prediction confusion matrix which shows how often we predicted which category in relation to what the actual category was and ROC curves which is a measure of how well our model is performing based on false positive and true positive rates. The ROC curves are provided as Figures 4, 5, and 6 in the Appendix.

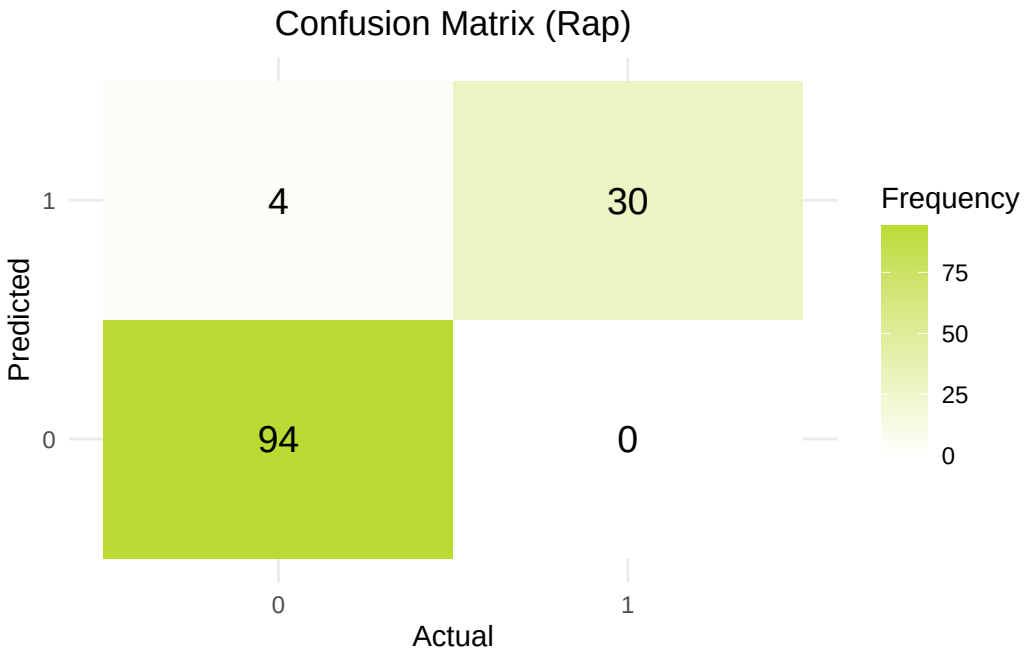


Figure 1: Confusion Matrix for Rap Playlist

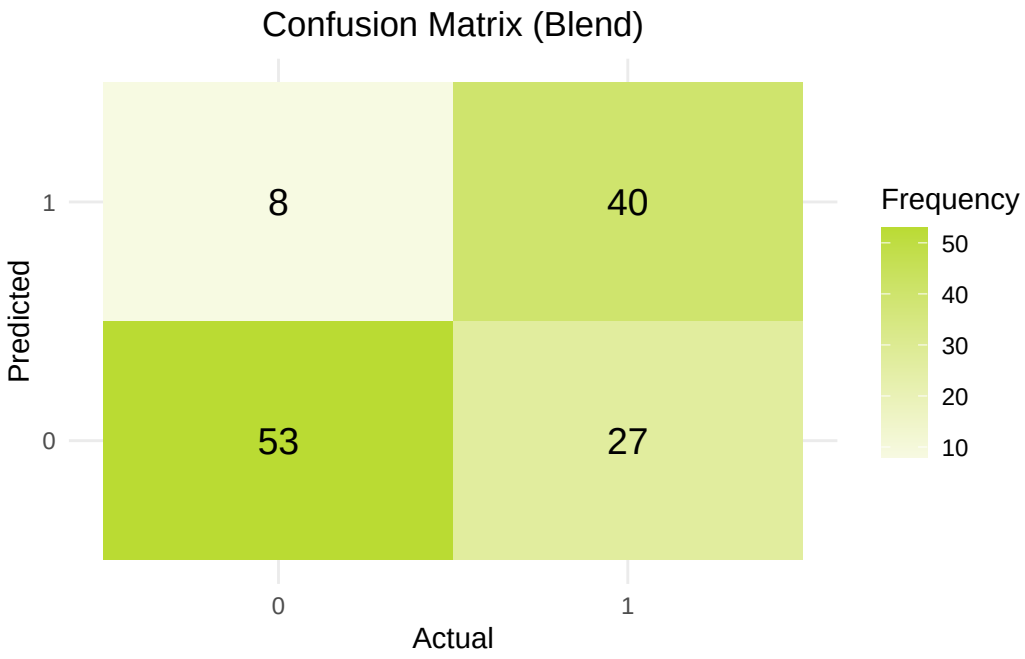


Figure 2: Confusion Matrix for Blend Playlist

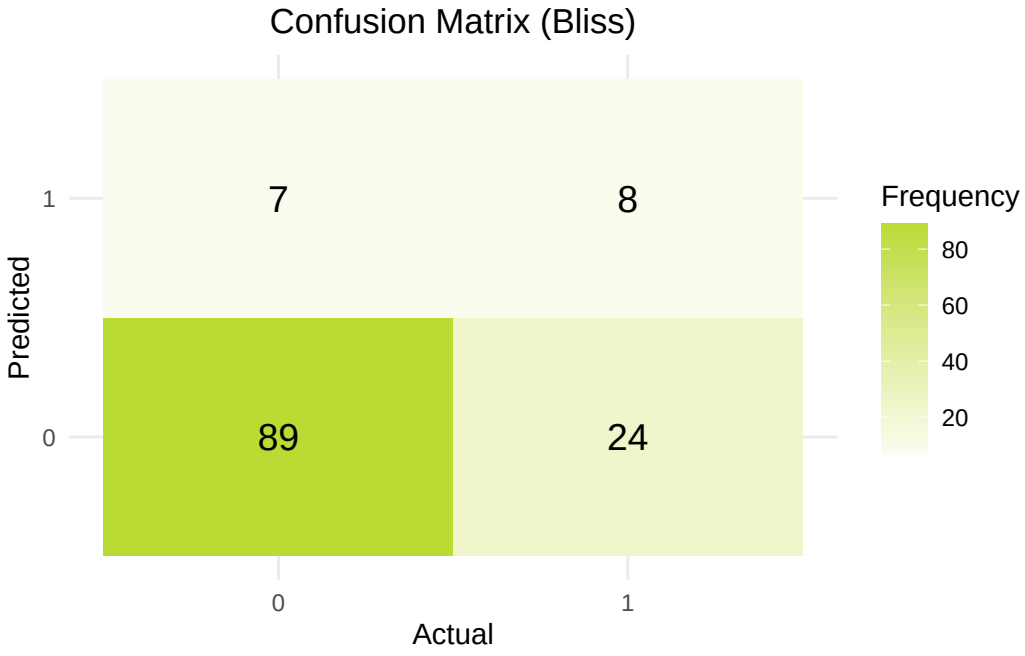


Figure 3: Confusion Matrix for Bliss Playlist

It is clear that the Rap playlist is predicting the most songs correctly overall, having only four false predictions in total and no false negatives. This indicates that this model is exceptional especially when it comes to determining what songs do not belong. The Blend playlist makes far more errors, having 35 in total, where its strong suit is determining songs that belong and it struggles a bit more at determining songs that do not belong. This makes sense for this playlist since it is a mix of genres so the model is recognizing that and simply thinking everything belongs. It is not initially clear that the Bliss playlist is the worse based on this graph because it has higher total accuracy compared to the Blend playlist, having 31 errors. However, since our main goal is prediction of belonging to the playlist the model the false negative rate has a big impact here which reduces our F1 score dramatically.

Contributions

Matthew was responsible for retrieving the data, cleaning the data, and performing exploratory data analysis. We both did an equal amount of work when choosing our desired modeling approach and on the initial proposal, this was more of a discussion type stage and had no formal work distribution. Ozi did the initial modeling, creating relevant functions for hyperparameter selection and model evaluation. Matthew modified some of these functions to allow for feature selection as well then Ozi cleaned up the functions as we moved on to our final testing. Matthew wrote the Introduction, Data, and Final Models section while Ozi wrote the rest and reviewed Matthew's work.

References

[1] Spotify Developer Page. Source

[2] Spotify Developer Tools, Get Basic Song Details. Source

[3] Spotify Developer Tools, Audio Features Description. Source

[4] Musicstax Website. Source

[5] Penalized Logistic Regression. Source

7 Appendix

7.1 Figures

Metric	Value
Accuracy	124/128 (96.88%)
Predicted in Playlist / In playlist	30/30 (100.0%)
In playlist / Predicted in Playlist	30/34 (88.24%)
F1	0.9375

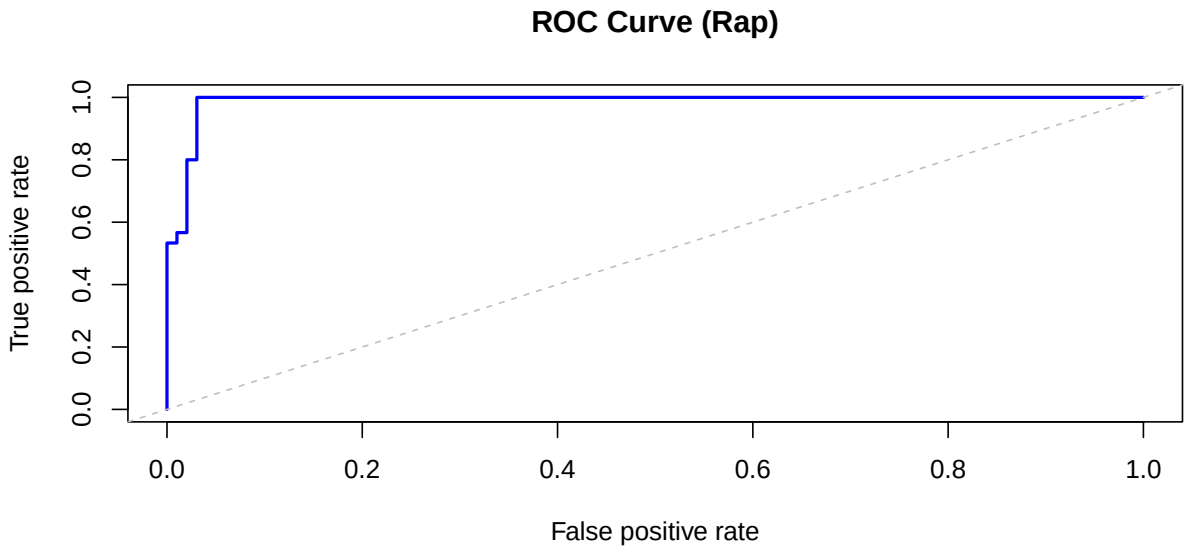
Table 4: Rap Model Test Set Performance

Metric	Value
Accuracy	93/128 (72.66%)
Predicted in Playlist / In playlist	40/67 (59.7%)
In playlist / Predicted in Playlist	40/48 (83.33%)
F1	0.6957

Table 5: Blend Model Test Set Performance

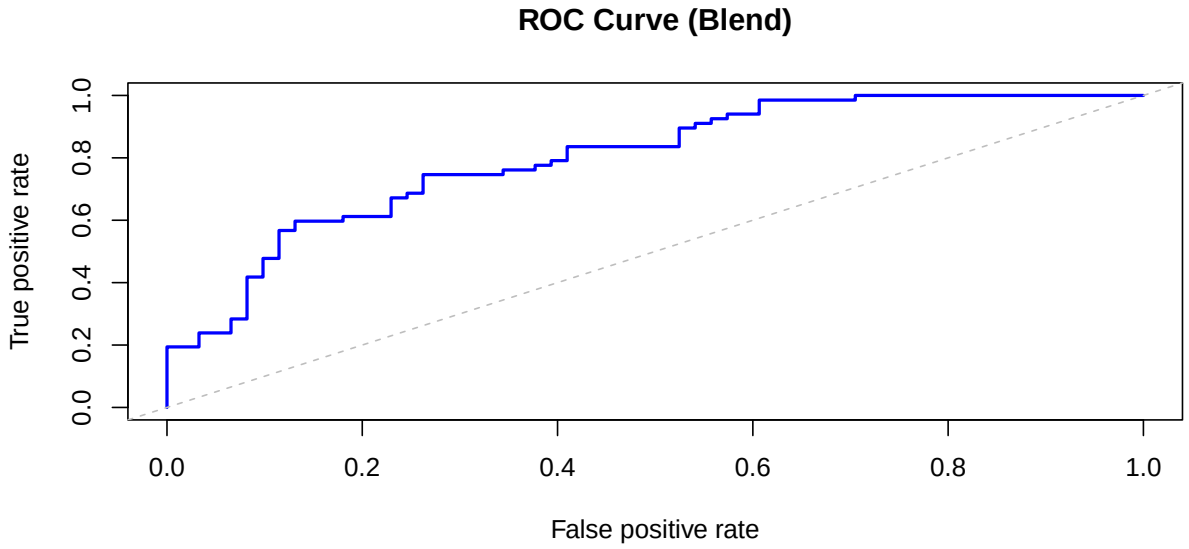
Metric	Value
Accuracy	97/128 (75.78%)
Predicted in Playlist / In playlist	8/32 (25.0%)
In playlist / Predicted in Playlist	8/15 (53.33%)
F1	0.3404

Table 6: Bliss Model Test Set Performance



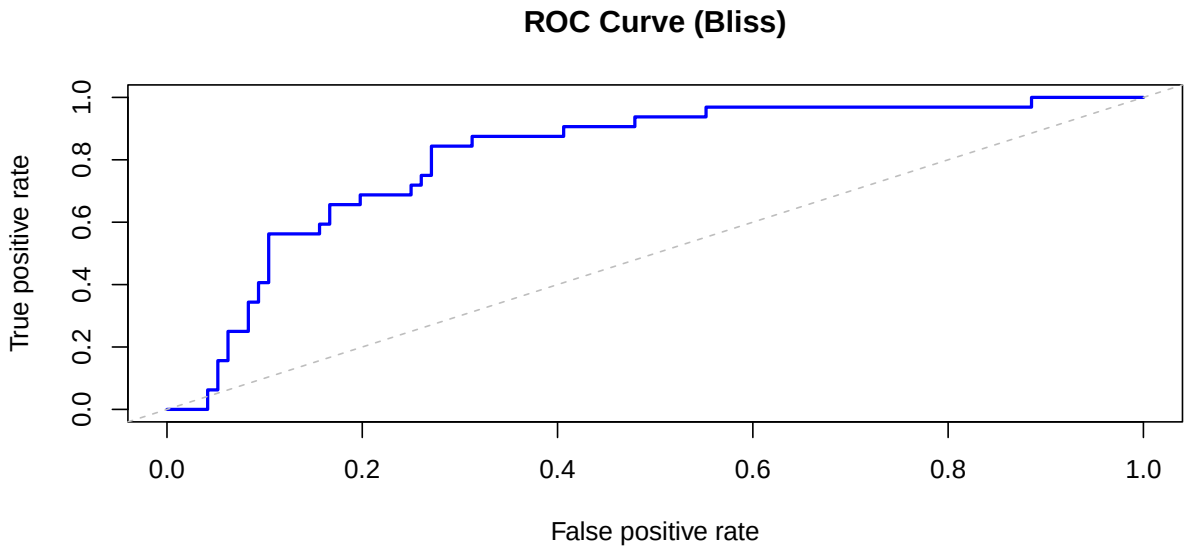

```
[1] 0.9887755
```

Figure 4: Rap Playlist ROC Curve (Higher Area Under Curve Better)




```
[1] 0.7998532
```

Figure 5: Blend Playlist ROC Curve (Higher Area Under Curve Better)




```
[1] 0.8115234
```

Figure 6: Bliss Playlist ROC Curve (Higher Area Under Curve Better)

7.2 Reproduction

The get-songs.py script is provided purely for completeness, it will not work without a Spotify API Key which we did not provide. The scrape-details.py is similarly provided purely for completeness, it will not work without a .env file as well as the songs.txt file which we did not provide. The resulting raw data (Spotify.csv) that is a result of these scripts is provided. The steps to reproduce our results are below.

- Run clean.R to create the cleaned data Spotify_clean.csv
- View exploratory data analysis in data_analysis.R
- Run test-train-split.py in order to create test and train data
- View modeling process in modeling.ipynb
- Run best_model_results.py to write results to csv files
- View results visualization in visualize_results.R
- Render Final-Report.qmd in order to generate this report

7.3 Code

7.3.1 Python (Model Fitting)

```
#########################
###     Modeling      ###
#########################

## Read Data ##
train_data = pd.read_csv('Train.csv')
test_data = pd.read_csv('Test.csv')

## Select Features ##
ALL_FEATURES = [
  'Explicit', 'Playtime', 'Loudness', 'Popularity',
```

```
  'Energy', 'Positiveness', 'Speechiness', 'Liveliness',
  'Acousticness', 'Instrumentalness', 'Danceability',
  'Tempo', 'Key', 'Mode','Age', 'Time.Signature',
  'TimeSignature_1', 'TimeSignature_3', 'TimeSignature_5'
]
ALL_FEATURES_WITHOUT_TIME = [feat for feat in ALL_FEATURES if 'Time' not in feat]
ALL_FEATURES_WITH_TIME_AS_CATEGORY = [
  feat for feat in ALL_FEATURES if feat != 'Time.Signature'
]
ALL_FEATURES_WITH_TIME_AS_NUMERIC = ALL_FEATURES_WITHOUT_TIME + ['Time.Signature']

## Define Functions ##
def get_folds_for_playlist(features_to_include, K=5,
                           playlist='in_Rap', quiet=False):
    FOLDS = []
    fold_size = len(train_data) // K
    for i in range(K):
        val = train_data.iloc[i*fold_size: (i+1)*fold_size]
        train = pd.concat([train_data.iloc[:i*fold_size]
                           , train_data[(i+1)*fold_size:]],
                          axis=0)
        X_train, y_train = train[features_to_include], train[playlist]
        y_train = y_train.to_numpy()
        X_val, y_val = val[features_to_include], val[playlist]
        y_val = y_val.to_numpy()
        FOLDS.append({
            'X_train': X_train,
            'y_train': y_train,
            'X_val': X_val,
            'y_val': y_val
        })
    if not quiet:
        print("Num folds:", K)
        print("Fold Xy Train Shape:",
              FOLDS[0]['X_train'].shape, FOLDS[0]['y_train'].shape
              )
        print("Fold Xy Val Shape:",
              FOLDS[0]['X_val'].shape, FOLDS[0]['y_val'].shape
              )
    return FOLDS

def get_feature_combos(feature_list=ALL_FEATURES,
                       necessary_features=[],
                       minimum_number_features = 4):
    candidate_combinations = []

    # Get all possible feature combinations
    for k in range(minimum_number_features, len(feature_list) + 1):
        combinations_of_size_k = list(itertools.combinations(feature_list, k))
        candidate_combinations.extend(combinations_of_size_k)
    candidate_combinations = [list(combo) for combo in candidate_combinations]
```

```
    final_combinations = []
    # Filter combinations for only good combos
    for comb in candidate_combinations:
        # skip this combination if it doesn't have
        # all the necessary features.
        has_all_necessary = all([feat in comb for feat in necessary_features])
        if not has_all_necessary:
            continue # continue to next combination

        # time signiature should be completely used or not at all
        num_time_sig_feats = len([feat for feat in comb if 'Time' in feat])
        if num_time_sig_feats == 0:
            final_combinations.append(comb)
        # If there are time signatures in there,
        # Keep this combination only if it's the single numerical feature,
        # or the 3 categories.
        else:
            if num_time_sig_feats == 1 and 'Time.Signature' in comb:
                final_combinations.append(comb)
            elif num_time_sig_feats == 3 and 'Time.Signature' not in comb:
                final_combinations.append(comb)

    return final_combinations

def cv_workflow(hyperparams_to_try,
                feature_sets_to_try,
                playlist, K=5, quiet=False):
    # Loops through hyperparameter options
    # and returns the hyperparameters that achieved
    # the best k-fold avg f1 score, along with the score.
    def eval_model_on_folds(model, quiet=False):
        scaler = StandardScaler()

        f1_scores = []
        for i, fold in enumerate(FOLDS):
            X_train, y_train = fold['X_train'], fold['y_train']
            X_val, y_val = fold['X_val'], fold['y_val']

            # Standardize
            X_train = scaler.fit_transform(X_train)
            # Apply same transformation to validation set
            X_val = scaler.transform(X_val)

            # Train and evaluate
            model.fit(X_train, y_train)
            preds = model.predict(X_val)

            f1_scores.append(evaluate(preds, y_val, quiet=quiet))
        avg = np.mean(f1_scores)
        if not quiet:
            print(f"Avg f1 score over {len(FOLDS)} folds: {avg}")
        return avg
```

```
    if not quiet:
        print(
          f"===Performing {K}-fold cross validation for playlist '{playlist}'==="
          )

    best_score = -1
    best_params = {}
    best_feats = []
    total_iterations = len(hyperparams_to_try) * len(feature_sets_to_try)
    it = 1
    for feature_list in feature_sets_to_try:
        FOLDS = get_folds_for_playlist(feature_list, K=K,
                                       playlist=playlist, quiet=quiet
                                       )
        for params_dict in hyperparams_to_try:
            if not quiet:
                print(f"{it}/{total_iterations}")
            model = LogisticRegression(**params_dict)
            k_fold_f1_score = eval_model_on_folds(model, quiet=quiet)
            if k_fold_f1_score > best_score:
                if not quiet:
                    print("Updating best model:", params_dict, feature_list)
                best_score = k_fold_f1_score
                best_params = params_dict
                best_feats = feature_list
            it += 1
    if not quiet:
        print(f"Best {K}-fold f1 score was", best_score)
        print("Best model were", best_params, feature_list)
    return best_params, best_score, best_feats

def recall(y_hat, y):
    # Computes the number of songs in the playlist
    # that were correctly predicted to belong in the playlist.
    # Eg. Out of all the songs in the rap playlist, how many did get right.
    # 'y': array of labels
    # 'y_hat': array of predictions

    in_playlist_idx = np.where(y==1)[0]
    labels = y[in_playlist_idx]
    predictions = y_hat[in_playlist_idx]
    num_correct = sum(labels == predictions)
    total = len(labels)
    return num_correct, total

def precision(y_hat, y):
    # Computes the number songs predicted to be in the playlist
    # that were actually in the playlist.
    # Eg. Out of all the songs we predicted to be rap, how many were actually rap?
    # 'y': array of labels
    # 'y_hat': array of predictions
```

```
    predicted_in_playlist_idx = np.where(y_hat==1)[0]
    labels = y[predicted_in_playlist_idx]
    predictions = y_hat[predicted_in_playlist_idx]
    num_correct = sum(labels == predictions)
    total = len(labels)
    return num_correct, total

def evaluate(y_hat, y, quiet=False, return_all=False):
    # Prints summary of performance
    num_correct = sum(y_hat == y)
    total = len(y)
    acc = num_correct/total

    rec, rec_total = recall(y_hat, y)
    rec_pct = rec/rec_total


    prec, prec_total = precision(y_hat, y)
    if prec_total != 0:
        prec_pct = prec/prec_total
    else:
        prec_pct = 1

    f1 = (2 * prec_pct * rec_pct)/(prec_pct + rec_pct)
    if not quiet:
        print(
          f"Accuracy: {round(100*acc, 2)}% ({num_correct}/{total})"
          )
        print(
          "Num Predicted in Playlist/Num In playlist: " + \
          f"{round(100*rec_pct, 2)}% ({rec}/{rec_total})"
          )
        if prec_total != 0:
            print(
              "Num in Playlist/Num Predicted In playlist: " + \
              f"{round(100*prec_pct, 2)}% ({prec}/{prec_total})"
              )
        else:
            print(f"Num in Playlist/Num Predicted In playlist: 100% (0/0)")
        print(f"F1 Score: {round(100*f1, 2)}%")
    if not return_all:
        return f1
    else:
        return num_correct, total, rec, rec_total, prec, prec_total, f1

## Defining Parameters ##
base_params = {'penalty': 'l1', 'solver': 'liblinear'}
hyperparams_to_try = [
    {**base_params, 'C':0.001},
    {**base_params, 'C':0.01},
    {**base_params, 'C':0.1},
    {**base_params, 'C':1},
```

```
    {**base_params, 'C':10},
    {**base_params, 'C':100},
    {**base_params, 'C':10e6}, # Almost no regularization
    {'penalty': None, 'solver': 'newton-cg'}, # No regularization
]

## Automatic Selection ##
## (Manual Selection Provided in Notebook) ##


## Rap

# First testing all combinations of features with no regularization.
necessary_rap = ['Explicit', 'Danceability', 'Speechiness', 'Age']
feature_combos_rap = get_feature_combos(necessary_features=necessary_rap)
best_params_rap, best_score_rap, selected_feats = \
cv_workflow([{'penalty': None, 'solver': 'newton-cg'}],
            feature_combos_rap, playlist='in_Rap',
            K=5, quiet=True)
print("Best features:", selected_feats)
print("Best score:", best_score_rap)

# Then testing all features with regularization.
best_params_rap, best_score_rap, best_lasso_features = \
cv_workflow(hyperparams_to_try,
            [ALL_FEATURES_WITH_TIME_AS_CATEGORY,
             ALL_FEATURES_WITH_TIME_AS_NUMERIC],
            playlist='in_Rap', K=5, quiet=True)
print("Best features:", best_lasso_features)
print("Best score:", best_score_rap)

# Okay, we found the best lasso regularization strength.
# Let's see which features it deemed important.
# We retrain on all the training data
X_train, y_train = train_data[best_lasso_features], train_data['in_Rap']
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

rap_model = LogisticRegression(**best_params_rap)
rap_model.fit(X_train_scaled, y_train)
coefs = pd.DataFrame({"Feature": best_lasso_features,
                      "Coef": rap_model.coef_[0],
                      "Abs_Coef": abs(rap_model.coef_[0])}).sort_values(
                        by="Abs_Coef",
                        ascending=False
                        )
lasso_most_important_feats = coefs[["Feature"]].iloc[:len(selected_feats)].to_numpy()
agree = set(lasso_most_important_feats[:,0]) & set(selected_feats)
agree

# Now let's try running Lasso on top of the best
# selected features from best subset selection
```

```
best_params_rap, best_f1, _ = cv_workflow(hyperparams_to_try,
                                          [selected_feats], playlist='in_Rap',
                                          K=5, quiet=True)
print("Best features:", selected_feats)
print("Best params:", best_params_rap)
print("Best score:", best_f1)

# Evaluate Best Model Performance
X_train, y_train = train_data[selected_feats], train_data['in_Rap']
X_val, y_val = test_data[selected_feats], test_data['in_Rap']
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

rap_model = LogisticRegression(**{'penalty': None})
rap_model.fit(X_train_scaled, y_train)

X_test, y_test = test_data[selected_feats], test_data['in_Rap'].to_numpy()
# Apply same transformation to validation set
X_test_scaled = scaler.transform(X_val)
preds = rap_model.predict(X_test_scaled)

evaluate(preds, y_test)


## Blend

# First testing all combinations of features with no regularization.
necessary_blend = ['Positiveness', 'Instrumentalness']
feature_combos_blend = get_feature_combos(necessary_features=necessary_blend)
best_params_blend, best_score_blend, selected_feats = \
cv_workflow([{'penalty': None, 'solver': 'newton-cg'}],
            feature_combos_blend, playlist='in_Blend',
            K=5, quiet=True)
print("Best features:", selected_feats)
print("Best score:", best_score_blend)

# Then testing all features with regularization.
best_params_blend, best_score_blend, best_lasso_features = \
cv_workflow(hyperparams_to_try,
            [ALL_FEATURES_WITH_TIME_AS_CATEGORY,
             ALL_FEATURES_WITH_TIME_AS_NUMERIC],
             playlist='in_Blend', K=5, quiet=True)
print("Best features:", best_lasso_features)
print("Best score:", best_score_blend)

# Okay, we found the best lasso regularization strength.
# Let's see which features it deemed important.
# We retrain on all the training data
X_train, y_train = train_data[best_lasso_features], train_data['in_Blend']
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

```
blend_model = LogisticRegression(**best_params_blend)
blend_model.fit(X_train_scaled, y_train)
coefs = pd.DataFrame({"Feature": best_lasso_features,
                      "Coef": blend_model.coef_[0],
                      "Abs_Coef": abs(blend_model.coef_[0])}).sort_values(
                        by="Abs_Coef",
                        ascending=False
                        )
lasso_most_important_feats = coefs[["Feature"]].iloc[:len(selected_feats)].to_numpy()
agree = set(lasso_most_important_feats[:,0]) & set(selected_feats)
agree

# Now let's try running Lasso on top of the best
# selected features from best subset selection
best_params_blend, best_f1, _ = cv_workflow(hyperparams_to_try,
                                            [selected_feats], playlist='in_Blend',
                                            K=5, quiet=True)
print("Best features:", selected_feats)
print("Best params:", best_params_blend)
print("Best score:", best_f1)

# Evaluate Best Model Performance
X_train, y_train = train_data[selected_feats], train_data['in_Blend']
X_val, y_val = test_data[selected_feats], test_data['in_Blend']
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

blend_model = LogisticRegression(**{'penalty': None})
blend_model.fit(X_train_scaled, y_train)

X_test, y_test = test_data[selected_feats], test_data['in_Blend'].to_numpy()
# Apply same transformation to validation set
X_test_scaled = scaler.transform(X_val)
preds = blend_model.predict(X_test_scaled)

evaluate(preds, y_test)


## Bliss

# First testing all combinations of features with no regularization.
necessary_bliss = ['Instrumentalness', 'Danceability']
feature_combos_bliss = get_feature_combos(necessary_features=necessary_bliss)
best_params_bliss, best_score_bliss, selected_feats = \
cv_workflow([{'penalty': None, 'solver': 'newton-cg'}],
            feature_combos_bliss, playlist='in_Bliss',
            K=5, quiet=True)
print("Best features:", selected_feats)
print("Best score:", best_score_bliss)

# Then testing all features with regularization.
best_params_bliss, best_score_bliss, best_lasso_features = \
```

```
cv_workflow(hyperparams_to_try,
            [ALL_FEATURES_WITH_TIME_AS_CATEGORY,
             ALL_FEATURES_WITH_TIME_AS_NUMERIC],
            playlist='in_Bliss', K=5, quiet=True)
print("Best features:", best_lasso_features)
print("Best score:", best_score_bliss)

# Okay, we found the best lasso regularization strength.
# Let's see which features it deemed important.
# We retrain on all the training data
X_train, y_train = train_data[best_lasso_features], train_data['in_Bliss']
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

bliss_model = LogisticRegression(**best_params_bliss)
bliss_model.fit(X_train_scaled, y_train)
coefs = pd.DataFrame({"Feature": best_lasso_features,
                      "Coef": bliss_model.coef_[0],
                      "Abs_Coef": abs(bliss_model.coef_[0])}).sort_values(
                        by="Abs_Coef",
                        ascending=False
                        )
lasso_most_important_feats = coefs[["Feature"]].iloc[:len(selected_feats)].to_numpy()
agree = set(lasso_most_important_feats[:,0]) & set(selected_feats)
agree

# Now let's try running Lasso on top of the best
# selected features from best subset selection
best_params_bliss, best_f1, _ = cv_workflow(hyperparams_to_try,
                                            [selected_feats], playlist='in_Bliss',
                                            K=5, quiet=True)
print("Best features:", selected_feats)
print("Best params:", best_params_bliss)
print("Best score:", best_f1)

# Evaluate Best Model Performance
X_train, y_train = train_data[selected_feats], train_data['in_Bliss']
X_val, y_val = test_data[selected_feats], test_data['in_Bliss']
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

bliss_model = LogisticRegression(**{'penalty': None})
bliss_model.fit(X_train_scaled, y_train)

X_test, y_test = test_data[selected_feats], test_data['in_Bliss'].to_numpy()
# Apply same transformation to validation set
X_test_scaled = scaler.transform(X_val)
preds = bliss_model.predict(X_test_scaled)

evaluate(preds, y_test)
```

7.3.2 R (Visualization)

```r
############################
### Import Cleaned Data ###
############################

train_data <- read.csv("Train.csv")
test_data <- read.csv("Test.csv")

########################
### Import Libraries ###
########################

library(dplyr)
library(ggplot2)
library(ROCR)

##########################
### Defining Functions ###
##########################

compute_performance_metrics <- function(actual, predicted) {
  cm <- table(Predicted = predicted, Actual = actual)
  accuracy <- sum(diag(cm)) / sum(cm)

  TP <- cm[2, 2]
  FN <- cm[1, 2]
  FP <- cm[2, 1]
  TN <- cm[1, 1]

  sensitivity <- TP/(TP + FN)
  specificity <- TN/(TN + FP)
  precision <- TP/(TP + FP)
  f1 <- 2*(precision*sensitivity)/(precision + sensitivity)

  return(list(
      ConfusionMatrix = cm,
      Accuracy = accuracy,
      Sensitivity = sensitivity,
      Specificity = specificity,
      Precision = precision,
      F1 = f1
  ))
}

visualize_for <- function(model, type, quiet = FALSE){
  if (model == "Rap"){
    pred_filename = "RapPredictions.csv"
    prob_filename = "RapProbabilities.csv"
    coef_filename = "RapCoefficients.csv"
  } else if (model == "Bliss") {
    pred_filename = "BlissPredictions.csv"
    prob_filename = "BlissProbabilities.csv"
```

```
  coef_filename = "BlissCoefficients.csv"
} else {
  pred_filename = "BlendPredictions.csv"
  prob_filename = "BlendProbabilities.csv"
  coef_filename = "BlendCoefficients.csv"
}

Predictions <- read.csv(pred_filename)
Probabilities <- read.csv(prob_filename)
Coefficients <- read.csv(coef_filename)
if (quiet == FALSE){
  print(Coefficients)
}

predictions <- Predictions[[paste0(model, ".Predictions")]]
probs <- Probabilities[[paste0(model, ".Probabilities")]]
Coefficients <- Coefficients %>%
  mutate(fill = ifelse(Coefficient < 0, "red", "blue"))
Coefficients$Feature <- gsub("_", "\n", gsub("TimeSignature", "Time\nSignature", Coefficients

y_val <- test_data[[paste0("in_", model)]]

metrics <- compute_performance_metrics(y_val, predictions)
cm_df <- as.data.frame(metrics$ConfusionMatrix)
pred <- prediction(probs, y_val)

# Confusion Matrix
if (type == "ConfusionMatrix"){
  cm_df %>%
    ggplot(aes(x = Actual, y = Predicted, fill = Freq)) +
    geom_tile() +
    geom_text(aes(label = Freq), color = "black", size = 5) +
    labs(title = paste0("Confusion Matrix (", model, ")"), x = "Actual", y = "Predicted", fi
    scale_fill_gradient2(low = "red", mid = "white", high = "#badb33") +
    theme_minimal() +
    theme(plot.title = element_text(hjust = 0.5))
} else if (type == "Coefficient"){
  # Coefficient Bar Chart
  # So it displays in descnding order of magnitude
  Coefficients$Feature <- factor(
    Coefficients$Feature,
    levels = Coefficients$Feature[order(-abs(Coefficients$Coefficient))]
  )
  Coefficients %>%
    ggplot(aes(x = Feature, y = Coefficient)) +
    geom_col(aes(fill = fill), width = 0.8) +
    scale_fill_identity() +
    labs(
      title = paste0("Feature Coefficients (", model, ")\n"),
      x = "\nFeature",
      y = "Coefficient"
    ) +
```

```
      theme_minimal() +
      theme(plot.title = element_text(hjust = 0.5),
            axis.line = element_line(color = "black"),
            axis.ticks = element_line(color = "black"))
  } else if (type == "Lift"){
    # Lift Chart
    # Rate of positive predictions lift over random guessing
    perf_lift <- performance(pred, "lift", "rpp")

    lift_df <- data.frame(
      rpp = perf_lift@x.values[[1]] * 100,
      lift = perf_lift@y.values[[1]]
    )

    ggplot(lift_df, aes(x = rpp, y = lift)) +
      geom_line(col = "blue", linewidth = 0.5) +
      labs(
        title = paste0("Lift Chart (", model, ")"),
        x = "% of Sample (Ranked by Score)",
        y = "Lift"
      ) +
      theme_minimal() +
      theme(plot.title = element_text(hjust = 0.5),
            axis.line = element_line(color = "black"),
            axis.ticks = element_line(color = "black"))
  } else if (type == "ROC"){
    # ROC plot
    perf <- performance(pred, "tpr", "fpr")  # true pos rate vs false pos rate

    # Plot ROC curve
    plot(perf, col = "blue", lwd = 2, main = paste0("ROC Curve (", model, ")"))
    abline(a = 0, b = 1, lty = 2, col = "gray")
    # AUC
    auc <- performance(pred, measure = "auc")
    return(auc@y.values[[1]])
  } else {
    return(NA)
  }
}

#########################
### Results for Rap  ###
#########################

visualize_for("Rap", "ConfusionMatrix")
visualize_for("Rap", "Coefficient")
visualize_for("Rap", "Lift")
visualize_for("Rap", "ROC")

#########################
### Results for Bliss ###
#########################
```

```r
visualize_for("Bliss", "ConfusionMatrix")
visualize_for("Bliss", "Coefficient")
visualize_for("Bliss", "Lift")
visualize_for("Bliss", "ROC")

#########################
### Results for Blend ###
#########################

visualize_for("Blend", "ConfusionMatrix")
visualize_for("Blend", "Coefficient")
visualize_for("Blend", "Lift")
visualize_for("Blend", "ROC")

#########################
###       Tables      ###
#########################

# Rap
RapMetrics <- read.csv("RapMetrics.csv", header=FALSE)
colnames(RapMetrics) <- c("Metric", "Value")
knitr::kable(RapMetrics)

# Blend
BlendMetrics <- read.csv("BlendMetrics.csv", header=FALSE)
colnames(BlendMetrics) <- c("Metric", "Value")
knitr::kable(BlendMetrics)

# Bliss
BlissMetrics <- read.csv("BlissMetrics.csv", header=FALSE)
colnames(BlissMetrics) <- c("Metric", "Value")
knitr::kable(BlissMetrics)
```