CS473 Classification Project

Import and transform data

I imported the dataset and did some exploratory analysis. I plotted histograms for every numerical variable and bar plots for every categorical variable (not included here or the python notebook). I propose the following data cleaning and transforming strategies:

1. We're supposed to have 50,000 rows of data, but there are actually 50,005 rows in the dataset. 5 of them are all NaNs. I dropped them.

```
df.dropna(axis=0, how='any', inplace=True)
```

2. Tempo denotes the "speed" of the song, as in how many beats per minute, and some of the tempo values are "?". Since tempo is a numerical variable, I'm replacing the missing values with the mean of the rest of the tempos.

```
average_tempo = df['tempo'][df['tempo'] != '?'].astype('float64').mean()
df['tempo'].replace({'?':average_tempo}, inplace=True)
df['tempo'] = df['tempo'].astype('float64')
```

3. obtained_date, artist_name, and track_name contain no actual information for prediction. Those columns are dropped.

```
df.drop(columns=['obtained_date', 'artist_name', 'track_name'], inplace=True)
```

4. duration_ms is the length of the song, and some of them are -1. Similar to tempo, I'm replacing with mean.

```
average_duration = df['duration_ms'][df['duration_ms'] != -1].mean()
df['duration_ms'].replace({-1:average_duration}, inplace=True)
```

5. Mode and key are two traits of a song. Most of the time, we say that a song is in some key and some mode, but not separately. For example, a song can be in C Major, but we rarely say that it has key C, or it has mode Major. Therefore, I'm combining those two categorical variables with an underscore, and then transforming this single column into dummies.

```
# Combine and make dummy
df['key_mode'] = df['key'] + '_' + df['mode']
df.drop(columns=['key', 'mode'], inplace=True)
df = pd.get_dummies(df, columns=['key_mode'])
```

Here're the columns for the transformed dataset, ready for classification models. Columns 0 through 10 are our numerical features, column 11 is our target variable, and columns 12 through 35 are dummies for key_mode.

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 50000 entries, 32894.0 to 63470.0
```

```
Data columns (total 26 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   popularity        50000 non-null  float64
 1   acousticness      50000 non-null  float64
 2   danceability      50000 non-null  float64
 3   duration_ms       50000 non-null  float64
 4   energy            50000 non-null  float64
 5   instrumentalness  50000 non-null  float64
 6   liveness          50000 non-null  float64
 7   loudness          50000 non-null  float64
 8   speechiness       50000 non-null  float64
 9   tempo             50000 non-null  float64
 10  valence           50000 non-null  float64
 11  music_genre       50000 non-null  object
 12  key_A             50000 non-null  uint8
 13  key_A#            50000 non-null  uint8
 14  key_B             50000 non-null  uint8
 15  key_C             50000 non-null  uint8
 16  key_C#            50000 non-null  uint8
 17  key_D             50000 non-null  uint8
 18  key_D#            50000 non-null  uint8
 19  key_E             50000 non-null  uint8
 20  key_F             50000 non-null  uint8
 21  key_F#            50000 non-null  uint8
 22  key_G             50000 non-null  uint8
 23  key_G#            50000 non-null  uint8
 24  mode_Major        50000 non-null  uint8
 25  mode_Minor        50000 non-null  uint8
dtypes: float64(11), object(1), uint8(14)
memory usage: 5.6+ MB
```

Split Features and Target

Our target variable is represented in strings. Let's transform them into numerical for easier handling. Two dictionaries are created to map those genres to numbers, and also back to genres.

```
translation_dict = pd.Series(index=df['music_genre'].value_counts().index, data=range(10)).to_dict()
translation_back_dict = {v:k for k, v in translation_dict.items()}
```

Splitting the dataset into X and y. The target variable is mapped with the dictionary here.

```
X = df.drop(columns=['music_genre'], inplace=False)
y = df['music_genre'].map(translation_dict)
```

Metrics

This is a classification project, and we'll be using some popular metrics. The four we're using here will be f1_score, accuracy_score, precision_score, and recall_score. I made a DataFrame to store those metrics. It will also be used to store the AUCs.

```
# metrics must accept parameters y_true and y_pred
metrics = [f1_score, accuracy_score, precision_score, recall_score]
# Create a DataFrame to store the metrics
metric_frame = pd.DataFrame(dtype='float64')
```

Custom Function

The API for scikit-learn models are consistent, so I made a custom function to automate the process. It does hyperparameter tuning, trains the model with the best hyperparameters, and plots the AUC for all classes.

```
def evaluate(classifier, metrics, metric_frame, X, y, classifier_name=None, classifier_params={}, param_grid={}):

    name = classifier_name if classifier_name else classifier.__name__

    # Use GridSearchCV for hyperparameter tuning
    grid = GridSearchCV(classifier(**classifier_params), param_grid, n_jobs=-1)
    grid.fit(X, y)
    if grid.best_params_:
        print(grid.best_params_)

    # Split training and testing, with equal class counts
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=seed, stratify=y)

    # Train the model on all features
    clf = classifier(**classifier_params, **grid.best_params_)
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    y_score = clf.predict_proba(X_test)

    # Code below for plotting is from
    # https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

    # Binarize target
    label_binarizer = LabelBinarizer().fit(y_train)
    y_onehot_test = label_binarizer.transform(y_test)
    # y_onehot_test.shape  # (n_samples, n_classes)

    # store the fpr, tpr, and roc_auc for micro averaging strategy
    fpr, tpr, roc_auc = dict(), dict(), dict()
    # Compute micro-average ROC curve and ROC area
    fpr["micro"], tpr["micro"], _ = roc_curve(y_onehot_test.ravel(), y_score.ravel())
    roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

    fig, ax = plt.subplots(figsize=(10, 10))

    # Plot the micro average ROC
    plt.plot(
        fpr["micro"],
        tpr["micro"],
        label=f"micro-average ROC curve (AUC = {roc_auc['micro']:.2f})",
        color="deeppink",
        linestyle=":",
        linewidth=4,
    )

    # Plot the ROC for each class
    cmap = plt.get_cmap('tab10')  # type: matplotlib.colors.ListedColormap
    colors = cycle(cmap.colors)  # type: list
    for class_id, color in zip(range(y_onehot_test.shape[1]), colors):
        RocCurveDisplay.from_predictions(
            y_onehot_test[:, class_id],
            y_score[:, class_id],
            name=f"ROC curve for {translation_back_dict[class_id]}",
            color=color,
            ax=ax,
        )

    # Save and show the plot
    plt.plot([0, 1], [0, 1], "k--", label="ROC curve for chance level (AUC = 0.5)")
    plt.axis("square")
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title(f"Extension of Receiver Operating Characteristic to One-vs-Rest multiclass\n{name}")
    plt.legend()
    plt.savefig(f'{name}.png')
    plt.show()
```

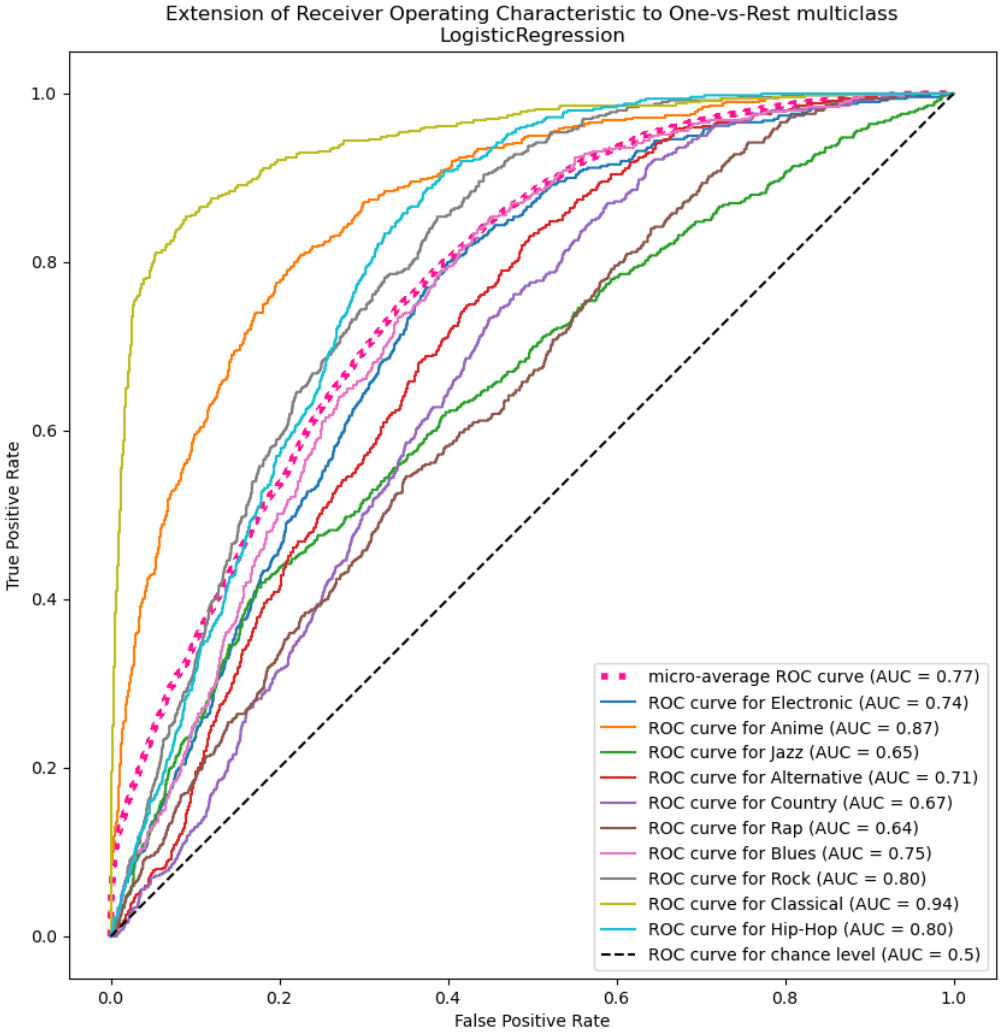
```
    # End of Reference

    # Calculate metrics
    metric_frame.at[name, 'AUC_micro'] = roc_auc["micro"]
    for metric in metrics:
        if 'average' in list(inspect.signature(metric).parameters.keys()):
            metric_frame.at[name, metric.__name__] = metric(y_true=y_test, y_pred=y_pred, average='micro')
        else:
            metric_frame.at[name, metric.__name__] = metric(y_true=y_test, y_pred=y_pred)

    # return the trained classifier for future use
    return clf
```

Implementing the Models

Sections below will use this function to skip a lot of redundant code. Parameter grids will still be included. Each function call draws a plot with one ROC curve for each class, as well as the micro-averaged AUC. The plots will be included in this report.

Logistic Regression

```
evaluate(LogisticRegression, metrics, metric_frame, X, y, None,
         classifier_params={'multi_class':'ovr', 'random_state':seed, 'n_jobs':-1, 'max_iter':10000},
         param_grid={})
```

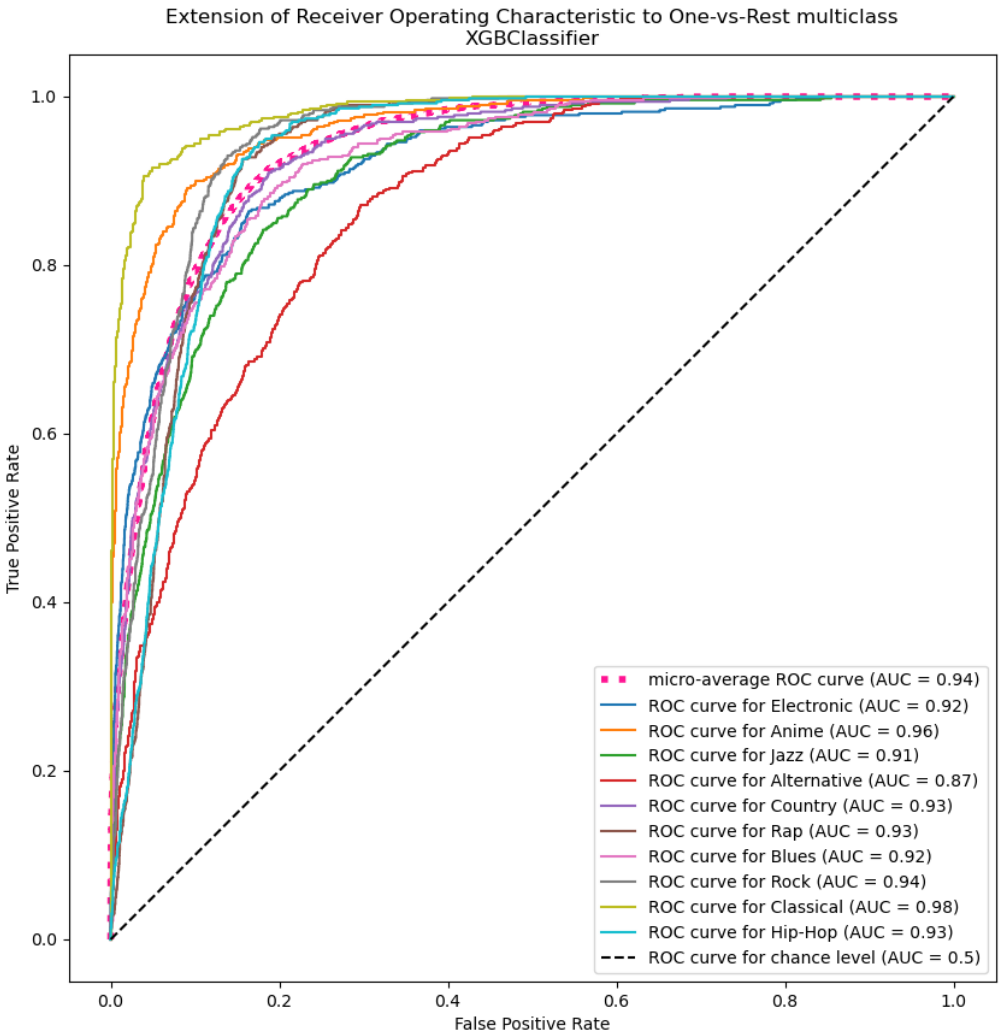


XGBoost

```
gpu_id = 0 if gpu else -1
tree_method = 'gpu_hist' if gpu else 'auto'
```

```
param_grid_xgb = {
    'n_estimators':[10, 100, 1000],
    'max_depth':[1, 10, 100, None]
}
```

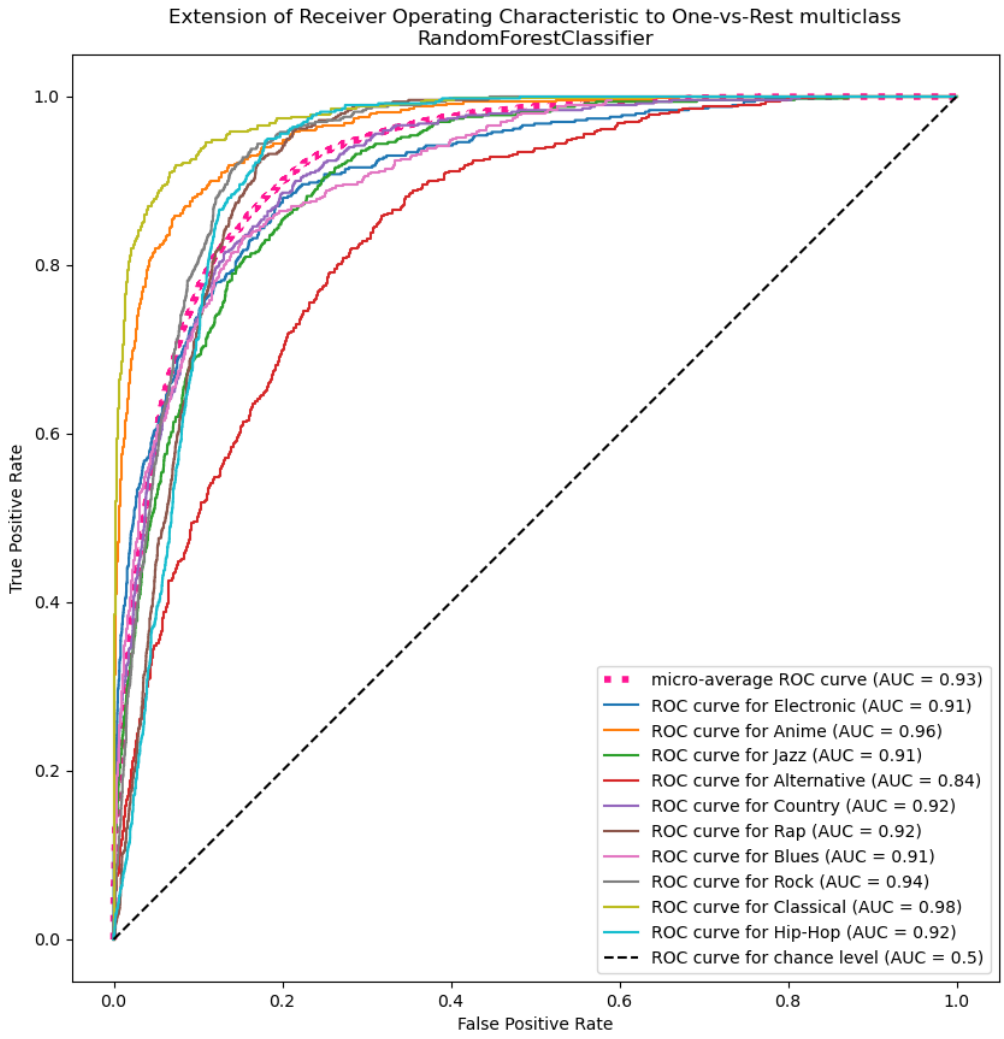
```
evaluate(XGBClassifier, metrics, metric_frame, X, y, None,
         classifier_params={'n_jobs':-1, 'random_state':seed, 'gpu_id':gpu_id, 'tree_method':tree_method},
         param_grid=param_grid_xgb)
```



Random Forest

```
param_grid_rf = {
    'n_estimators':[10, 100, 1000],
    'max_depth':[1, 10, 100, None]
}
```

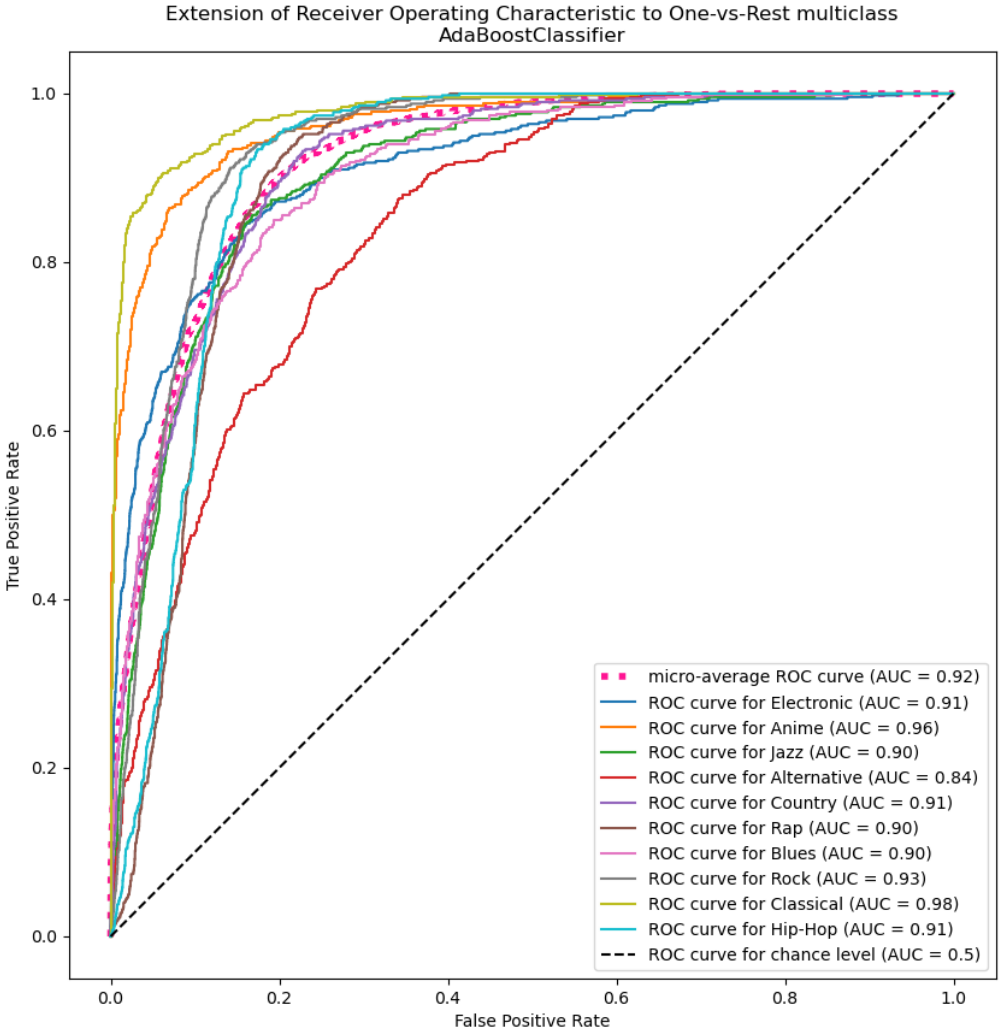
```
evaluate(RandomForestClassifier, metrics, metric_frame, X, y, None,
         classifier_params={'n_jobs':-1, 'random_state':seed},
         param_grid=param_grid_rf)
```



AdaBoost

```
param_grid_adb = {
    'estimator':[
        DecisionTreeClassifier(max_depth=1),
        DecisionTreeClassifier(max_depth=10),
        DecisionTreeClassifier(max_depth=None)
    ],
    'n_estimators':[10, 50, 100],
    'learning_rate':[0.1, 1.0, 10]
}
```

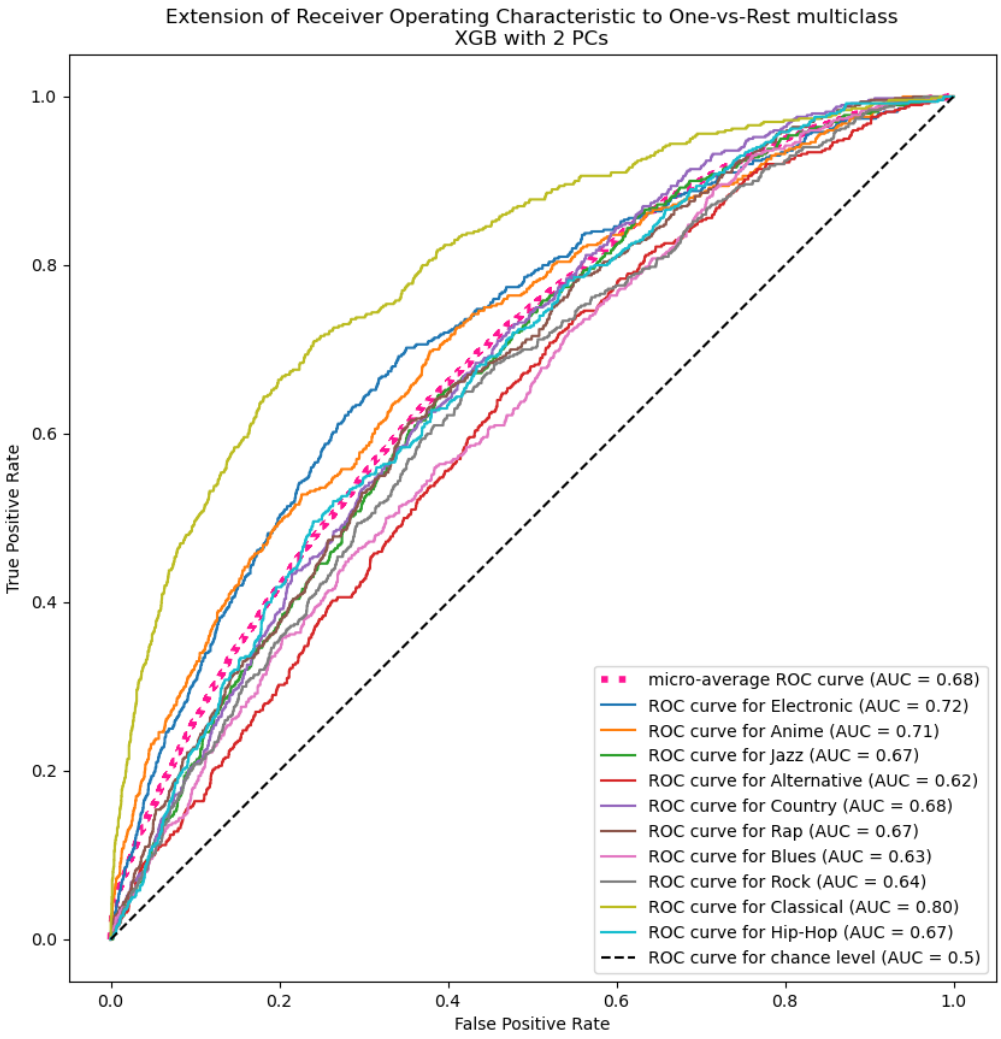
```
evaluate(AdaBoostClassifier, metrics, metric_frame, X, y, None,
         classifier_params={'random_state':seed},
         param_grid=param_grid_adb)
```



XGBoost with PCA

```
pca = PCA(n_components=2, random_state=seed)
X_pca = pca.fit_transform(X)
```

```
evaluate(XGBClassifier, metrics, metric_frame, X_pca, y, 'XGB with 2 PCs',
         classifier_params={'n_jobs':-1, 'random_state':seed, 'gpu_id':gpu_id, 'tree_method':tree_method},
         param_grid=param_grid_xgb)
```



Performance for Models

	AUC_micro	f1_score	accuracy_score	precision_score	recall_score
LogisticRegression	0.773492	0.2796	0.2796	0.2796	0.2796
XGBClassifier	0.935002	0.5768	0.5768	0.5768	0.5768
RandomForestClassifier	0.925246	0.5534	0.5534	0.5534	0.5534
AdaBoostClassifier	0.920657	0.5190	0.5190	0.5190	0.5190
XGB with 2 PCs	0.684540	0.2078	0.2078	0.2078	0.2078

Summary on Performance

For this project, XGBoost performs the best with an AUC of 0.935. Random Forest and AdaBoost are not far behind, with AUC of about 0.925 and 0.921 respectively. Logistic Regression requires very little time to train, but significantly lacks performance compared to the other three models. If we fit a PCA with 2 components on the features, and train a XGBoost model on those 2 PCs, the AUC drops from 0.935 to 0.685, which is even worse than Logistic Regression. This suggests the 2 PCs are not able to capture a lot of variance in the original feature set, and the PCA loses a lot of information in its transformation phase.

The four metrics are just for reference here, since we're primarily focusing on AUC as the main metric for this project. However, it's weird to see the four metrics have exactly the same value for each model. This could be because the class balances are perfect for the test set (and the training set as well), and the "average" parameter for the metrics are set to "micro".

Clustering

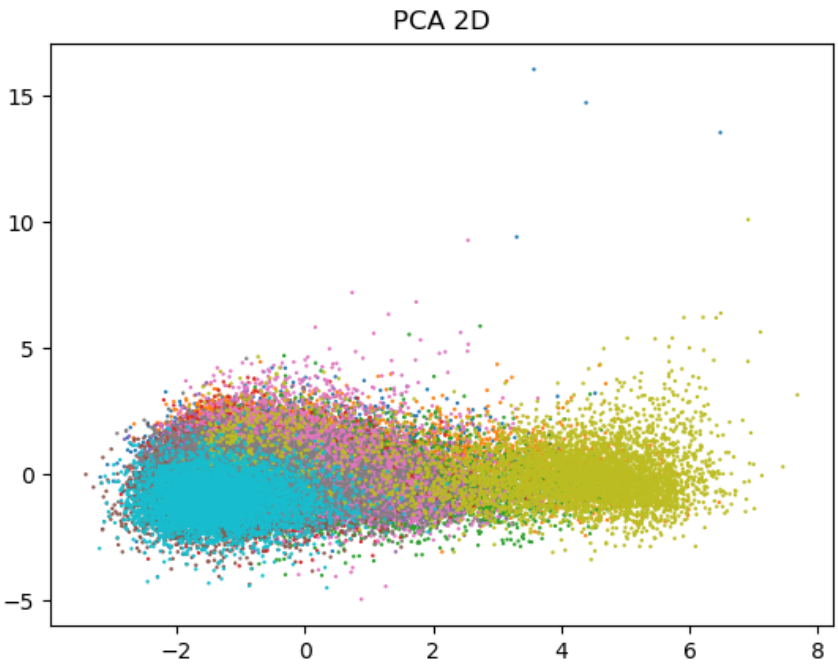
We have the labels for this project, so doing clustering isn't that necessary at all. Nevertheless, I still tried it to see if we can arrive at very separable clusters with respect to the labels. If we can identify the labels just by clustering, then it's almost unnecessary to train the classification models for classification; we just transform the datapoint with the fitted clustering model, and see which cluster it belongs to. That's not the case though, as we'll see.

I extracted and normalized the numerical features from the cleaned dataset.

```
df_numerical = df.select_dtypes(include='float64')
# Normalize the dataset
df_numerical = (df_numerical-df_numerical.mean())/df_numerical.std()
```

PCA with 2 PCs

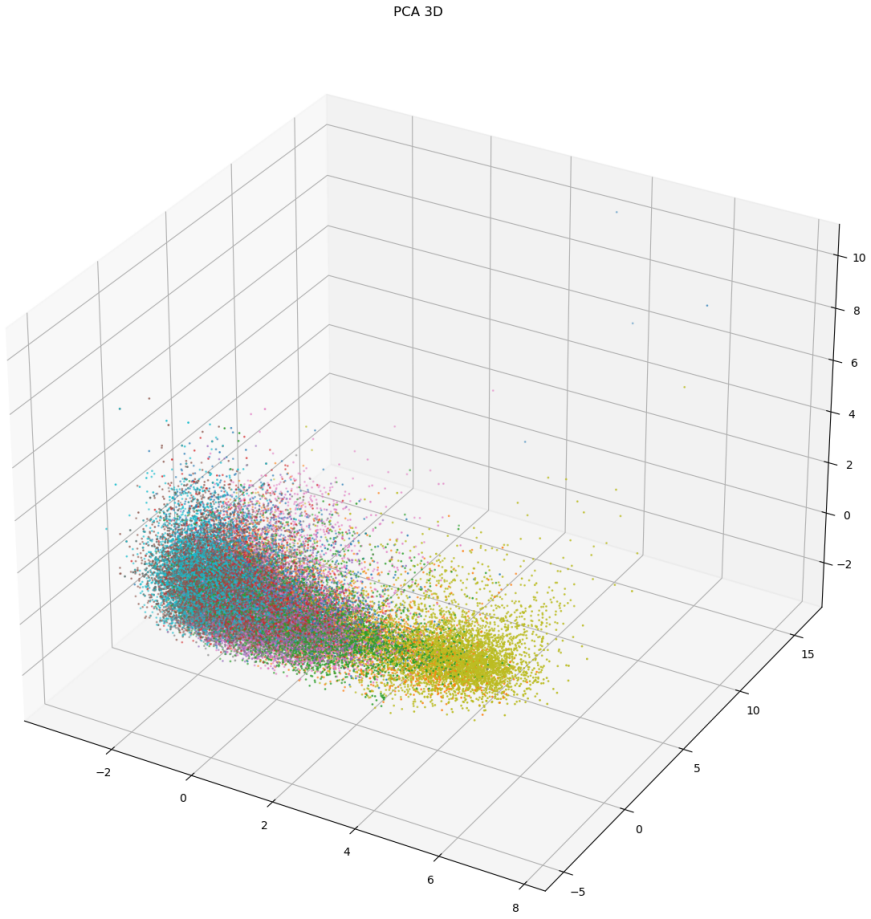
```
pca = PCA(n_components=2, random_state=seed)
pca_df = pd.DataFrame(pca.fit_transform(df_numerical))
plt.scatter(pca_df.loc[:, 0], pca_df.loc[:, 1], s=.5, c=y, cmap='tab10')
plt.title('PCA 2D')
plt.savefig('pca_2d.png')
plt.show()
```




```
Explained Variances: [3.70493226 1.34611637]
Explained Total Variance Ratio: 0.45918623888777443
```

PCA with 3 PCs

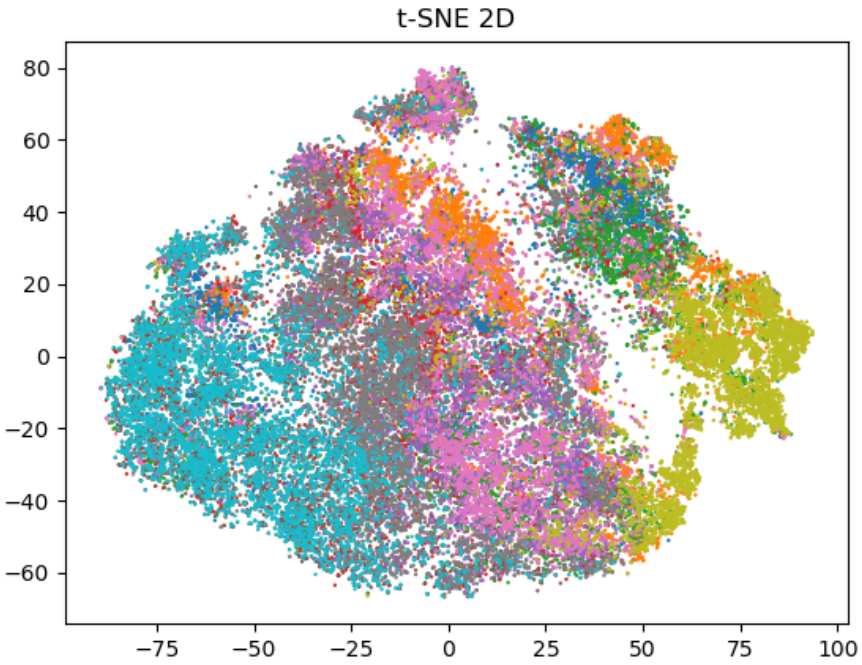
```
pca = PCA(n_components=3, random_state=seed)
pca_df = pd.DataFrame(pca.fit_transform(df_numerical))
fig = plt.figure(figsize=(15, 15))
ax = plt.axes(projection="3d")
ax.scatter3D(pca_df.loc[:, 0], pca_df.loc[:, 1], pca_df.loc[:, 2], s=1, c=y, cmap='tab10')
ax.set_title('PCA 3D')
plt.savefig('pca_3d.png')
plt.show()
```




```
Explained Variances: [3.70493226 1.34611637 1.05368136]
Explained Total Variance Ratio: 0.5549754536250777
```

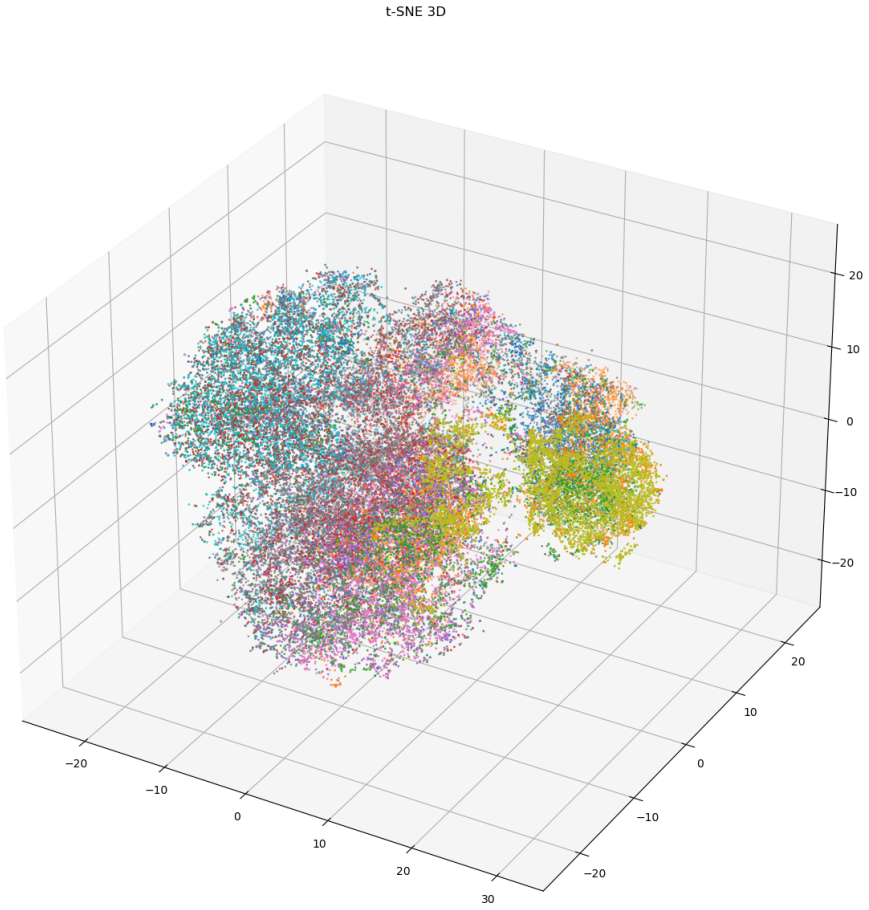
t-SNE with 2 embeddings

```
tsne = TSNE(n_components=2, perplexity=100, random_state=seed)
tsne_df = pd.DataFrame(tsne.fit_transform(df_numerical))
plt.scatter(tsne_df.loc[:, 0], tsne_df.loc[:, 1], s=.5, c=y, cmap='tab10')
plt.title('t-SNE 2D')
plt.savefig('t-SNE_2D.png')
plt.show()
```



t-SNE with 3 embeddings

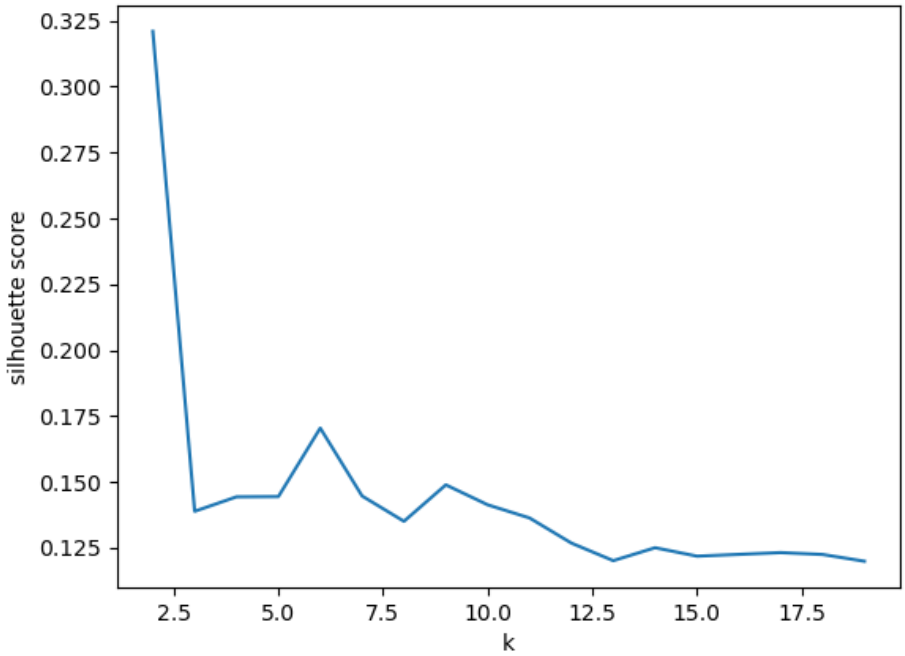
```
tsne = TSNE(n_components=3, perplexity=100, random_state=seed)
tsne_df = pd.DataFrame(tsne.fit_transform(df_numerical))
fig = plt.figure(figsize=(15, 15))
ax = plt.axes(projection="3d")
ax.scatter3D(tsne_df.loc[:, 0], tsne_df.loc[:, 1], tsne_df.loc[:, 2], s=1, c=y, cmap='tab10')
ax.set_title('t-SNE 3D')
plt.savefig('t-SNE_3d.png')
plt.show()
```



K-Means without labels

Here I ignored the labels and fitted K Means clustering on the dataset. I varied the value of k to see which k yields the highest silhouette score.

```
silhouettes = pd.Series(index=np.arange(2, 20), dtype='float64')
for k in silhouettes.index:
    kmeans = KMeans(n_clusters=k, n_init='auto', random_state=seed)
    labels = kmeans.fit_predict(df_numerical)
    silhouettes[k] = silhouette_score(df_numerical, labels)
plt.plot(silhouettes)
plt.xlabel('k')
plt.ylabel('silhouette score')
plt.savefig('silhouette_vs_k.png')
plt.show()
```



2 clusters has the highest silhouette score, which is the minimum number of clusters possible. This means the points are too close together, and there's no meaningful clustering possible. The silhouette score for 10 clusters, which is the true cluster count, is not much higher than neighboring k values, and follows the normal downwards trend.

Summary on Clustering

As we can see from the graphs, no meaningful clustering can be done with this dataset. The points don't form clearly visible clusters for all methods I tried. The K-Means clustering with silhouette score showed that 2 clusters is the best, which is consistent with the conclusion that the points are too close together.

One genre stands out from the graphs: Classical. The points for this genre (yellowish green) are far away from other genres. This suggests classical music is really different from the rest, and is the most unique type of music in the dataset. This insight is consistent with the classification results earlier: the AUC for Classical is the highest amongst all genres for all models.

Overall Summary and Discussion

For this project, **gradient boosting models (XGBoost specifically) performs the best with an AUC of 0.935**, with other tree-based models not far away in terms of performance. Since this is a classification project with labels provided, clustering isn't that necessary, and doesn't work well anyways. Classical music consistently gets the highest AUC out of all genres, and all models work the best at classifying this type of music.

Despite **the clustering graphs not showing clear clusters**, we can still infer some conclusions from them. Knowing that classical music is at the far right of the PCA graphs, I looked at the dots on the far left: they are Hip-Hop and Rock. **I would hypothesize that the horizontal principal component from the PCA describes how "elegant", "formal", or "timeless" the music is.** This factor would have a high feature importance value for the models, and contributes to the good performance of them.

Extra Credit

I’m not a professional in music, nor have I ever extensively learned to play any instrument. I do know that there’re 12 possible keys in music, and each key can match with either the mode Major or Minor. I printed out all the unique combinations of key and mode in this dataset, as well as their count:

	0	1	2	3	4	5	6	7	8	9	10	11
key	A	A	A#	A#	B	B	C	C	C#	C#	D	D
mode	Major	Minor	Major	Minor	Major	Minor	Major	Minor	Major	Minor	Major	Minor
count	2950	1875	1593	1763	1662	2127	4307	1215	3910	1495	4109	1156

	12	13	14	15	16	17	18	19	20	21	22	23
key	D#	D#	E	E	F	F	F#	F#	G	G	G#	G#
mode	Major	Minor	Major	Minor	Major	Minor	Major	Minor	Major	Minor	Major	Minor
count	1030	560	1721	2039	2338	2003	1619	1482	4332	1395	2528	791

Guess what, there’re indeed 24 combinations! G Major and C Major are the most common ones. That is actually consistent with my prior knowledge. On a piano, keys C, D, F, G, and A have black keys to the right of them, denoting an half-key increase in pitch (called sharp, the # sign). Keys E and B does not have black keys to the right. This knowledge is also consistent with the keys in the dataset, as C, D, F, G, and A have their respective sharp keys, but E and B does not.