Music Genre Classification

October 7, 2024

1. Introduction

When listening to music, there are often many songs that resemble a certain genre, but you can't quite put your finger on which one it is. By analyzing distinct features for a specific genre, a computer can with ease tell you which genre the song belongs to. This process is called audio classification, and it's not only applicable for when you want to know the genre, or name of a certain song. Audio classification can be applied in virtual assistants, voice recognition or even classifying animal species based on their unique sounds. In this project we are trying to distinguish different genres from music files using machine learning methods.

Section 2 of this report briefly discusses the problem formulation, data and feature selection of music genre classification. Section 3 introduces the ML models used—KNN, Logistic Regression, and Random Forest—and explains the methods applied. Section 4 presents the results, comparing the performance of these models. Finally, Section 5 concludes with an analysis of the findings and the selection of the best model for music genre classification.

2. Problem Formulation

The data used for this project was sourced from the **GTZAN Dataset** [1], which contains 1000 audio tracks, each track split into 10 parts, across 10 different genres (= 10 000 data points). The dataset includes audio files, a visual representation for each audio file, and two CSV files containing features of the audio files. The features in the data are continuous and the labels are categorical. For this project, the CSV files containing precomputed statistical summaries of audio features will be utilized. The CSV files contain a mean and variance computed over different features, which make up the audio files. The CSV files contain loads of data but we chose 10 features based on some visual testing. In the following segment we'll explain the central features we use:

- **Spectral Bandwidth:** Measures the difference between the upper and lower frequencies in the signal
- **MFCC or Mel frequency cepstral coefficient:** Captures the timbral texture of music, which means the 'tone quality, color or voice' [a]. The timbral texture of an audio signal is very indicative of its genre. There are several different MFCC coefficients, but we'll only use the ones that are more distinguishable
- **Spectral Centroid:** Describes the center of mass of the frequency spectrum. It can be referred to as "brightness" of sound because it tells us were the majority of the spectral energy is concentrated in terms of frequency
- **Zero crossing rate:** Tells us the rate at which the signal changes from positive to negative and vice versa [c]. The zero crossing rate gives us an estimate for the "dominant frequency" in the audio clip [d]
- **RMS or Root Square Mean Measure:** Measures the average power of a sound wave over time

3. Methods

3.1 Data and feature selection

We chose to use data from the **GTZAN Dataset** [1] for this project, due its diverse data set, availability and pre-processed features. As mentioned in section 2 (Problem Formulation) the dataset includes ten thousand data points, which is enough for the purpose of this project. We are interested in all the labels of the dataset, which are: Blues, Classical, Country, Disco, Hip Hop, Jazz, Metal, Pop, Reggae and Rock.

The features were chosen based on data visualization. We made three different plots for distinguishing the most usable features. We created a Correlation Heat Map to visualize the relationships between variables, Feature Distribution histograms for each feature to visualize the distribution, and a graph of the Feature Variance to get a sense of which feature values vary the most (plots can be found at the end of the report). We selected only 10 features based on the information of these plots due to the risk of overfitting if too many features were used. The features we selected were: spectral_bandwidth_mean, spectral_centroid_mean, mfcc2_mean, spectral_centroid_var, rolloff_var, zero_crossing_rate_var, rms_mean ,mfcc13_var, and mfcc9_mean, mfcc20_var

3.2 Choice of machine learning model

Several machine learning models were considered in this project. However, we decided to use the ones that we thought were most suitable. Given the structure and nature of the dataset, models that can handle both linear and non-linear relationships between the features were selected. The models chosen for this task were K-Nearest Neighbors (KNN), Logistic Regression, and Random Forest. Majority voting is used between these models to ensure a simple yet accurate solution.

K-Nearest Neighbors (KNN)

KNN is a supervised learning algorithm that works by classifying a new data point and measuring the distance from it and its closest neighbors in the training set. All the features have to be scaled to ensure consistent distance measurement between the data points. Each future is normalized by using **StandardScaler** from `sklearn.preprocessing`. This scales the futures to have a mean of 0 and a standard deviation of 1, which will ensure that no future will dominate any distance of the calculations.

The training of the data is done by using `KNeighborsClassifier` from `sklearn.neighbors`. The model is trained on scaled training data that will store training examples that are used once the model is trained. Then we classify new data points and compare them to the nearest-neighbors in the training set.

Using this model is optimal because it does not assume any underlying distribution in the data, which is good for the complex data we have. KNN also leverages the natural of songs in space, which makes it effective for genre classification. The features used also give multidimensional information about each song that easily is extracted with KNN because of its use of multi-dimensional feature space. Apart from all that, KNN is also simple to understand and requires no complex model training. It also doesn't optimize any loss function due to the classification performance being directly evaluated through accuracy.

Logistic Regression

Logistic Regression, which is a supervised learning algorithm, will be used to classify music tracks into one of the 10 genres. The model works by fitting a linear decision boundary to the data and then using the logistic function to output probabilities for each class. Due to our problem having multiple classes, we use the strategy **one-vs-rest** (OvR)[e], where a separate classifier is trained for each genre. Preprocessing of the data is needed due to Logistic Regression being sensitive to future scales. We therefore apply **StandardScaler** once again from the `sklearn.preprocessing` library to normalize the features, which ensures that they have a mean of 0 and a standard derivation of 1.

To train the model, we use the **LogisticRegression** classifier from `sklearn.linear_model`. The model is trained using the scaled training data. It learns a set of weights for each feature that best separates the genres. The **one-vs-rest** strategy is automatically implemented when using `sklearn`'s Logistic Regression model.

We use the loss function Logistic Loss, which measures how well the predicted probabilities match the true labels. It penalizes predictions that are far from the true labels by assigning high loss values to wrong classifications. However in `sklearn`, the loss function is implemented by default, so we don't do any manual work for this part.

We chose to use Logistic Regression due to its simplicity and efficiency. Compared to other models, it handels multi-class classification problems effectively, which is crucial since our data set contains 10 music genres. It also provides a useful reference point to assess the performance of the other more complex models used in this project.

Random Forest

Random Forest is a supervised learning method that builds multiple decision trees and combines their predictions to improve accuracy and prevent overfitting. Each decision tree is trained with a random subset of data and features.

Unlike with KNN and Logistic Regression, we don't have to do any future scaling. This is because Random Forest operates by splitting the data at decision nodes based on the actual values of the features, not their magnitudes.

To train this model, we use RandomForestClassifier implemented in `sklearn.ensemble`. At each split within a decision tree, a random subset of features is considered, to ensure that the model doesn't rely too heavily on one feature. We tune the parameters for Randomforest so that it gives us the best optimized answer. Once trained, each tree in Randomforest votes for a class, and the class with the majority votes is chosen as the final prediction.

We use the loss function Gini Impurity to measure the probability that a random chosen element would be incorrect. Random Forest chooses to split at each nod in the decision tree that maximizes information gain, based on the Gini Impurity value. The lower the value the better the split.

We chose Random Forest due to its ability to handle non-linear relationships between futures, considering that we use linear models as well. This combination ensures that our project covers both linear and complex feature relationships. Random Forest gives a more accurate prediction of genre by averaging multiple trees, compared to using the Random Tree method. With Random Forest, we also get the ability to see which features contribute the most to the classification decision.

3.3 Model Validation Process

Model validation is crucial for ensuring that the model works for both training data, but also on completely unseen data. By splitting the data into different categories (training, validation and test sets) and using those to test the model, we can be confident that the model isn't only working with a certain dataset.

The training set is often the largest of the three. As the name implies, the training set is used for training the model and it is from the training set, from which the model learns to function properly. The validation set is used to finetune and measure the performance of the model during training and help with avoiding a situation where the model becomes dependent on the training set, and can no longer function on any other data. This is also known as overfitting. The last of the three is the test set. The test set is used to measure the final performance of the model. We decided to use a 64%-16%-20% split, 64% being the training set, 16% being the validation set and 20% being the test set. Because we have 10 000 data points , 6400 of them are being used for training, 1600 for validation and 2000 for the final testing. 64% of the total data should be enough to ensure the model learns to function correctly and with the help of the validation dataset, we are trying to prevent overfitting from occurring.

4. Results

We evaluate the performance of our chosen methods by looking at the variables accuracy, recall, precision and f1-score. Here are the observed results:

The observed results for using the selected futures and data consisting of 1000 data points

Method	**Training set**	**Validation set**	**Test set**
KNN	ACC = 0.58 PRE = 0.58 REC = 0.58 F1 = 0.56	ACC = 0.46 PRE = 0.47 REC = 0.46 F1 = 0.44	ACC = 0.44 PRE = 0.45 REC = 0.44 F1 = 0.43
Logistic Regression	ACC = 0.58 PRE = 0.56 REC = 0.58 F1 = 0.56	ACC = 0.53 PRE = 0.53 REC = 0.53 F1 = 0.52	ACC = 0.43 PRE = 0.42 REC = 0.43 F1 = 0.41
Random Forest	ACC = 1.00 PRE = 1.00 REC = 1.00 F1 = 1.00	ACC = 0.61 PRE = 0.62 REC = 0.61 F1 = 0.61	ACC = 0.49 PRE = 0.49 REC = 0.49 F1 = 0.48

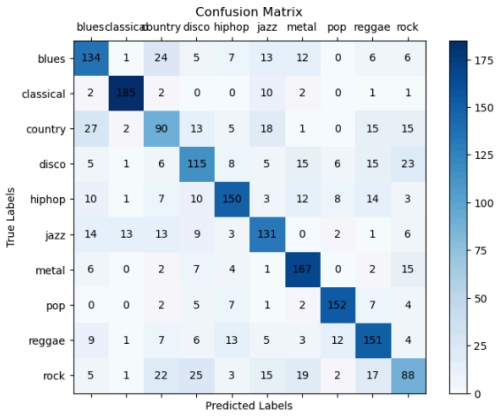
The observed results for using the selected futures and data consisting of 10 000 data points

Method	**Training set**	**Validation set**	**Test set**

KNN	ACC= 1.00 PRE = 1.00 REC = 1.00 F1 = 1.00	ACC = 0.62 PRE = 0.62 REC = 0.62 F1 = 0.62	ACC = 0.60 PRE = 0.61 REC = 0.60 F1 = 0.61
Logistic Regression	ACC = 0.49 PRE = 0.46 REC = 0.49 F1 = 0.47	ACC = 0.49 PRE = 0.47 REC = 0.49 F1 = 0.47	ACC = 0.50 PRE = 0.47 REC = 0.50 F1 = 0.47
Random Forest	ACC = 1.00 PRE = 1.00 REC = 1.00 F1 = 1.00	ACC = 0.70 PRE = 0.70 REC = 0.70 F1 = 0.70	ACC = 0.68 PRE = 0.68 REC = 0.68 F1 = 0.68

The accuracy of all of the methods are relatively low. The KNN method improves its performance with the increase in the amount of data, as do all of the methods. Logistic Regression has the smallest increase in accuracy when the amount of data is multiplied by 10, which indicates it being the worst of these three methods for music genre classification. The winner for the best performance and therefore the chosen method is the Random Forest model with both 1000 and 10 000 data points. In KNN and Random Forest it is noticeable that the training accuracy drops for both the methods by quite a bit, but the fall off is less drastic for random Forest. Though Linear Regression gives a quite solid score that doesn't fluctuate a lot from the training, to validation to testing, the accuracy of the model is almost ⅕ less than Random Forest. The Random Forest method predicted the correct genre for the song with a test accuracy of 68% (test error of 32%). 68% is a decent accuracy, considering the amount of possibilities for the genre is 10, and by tossing a 10-sided die the probability of getting the right genre would be 10%.

Confusion Matrix of Random Forest method



5. Conclusion

Although the Random Forest method achieves an respectable accuracy of 68% we can't say that the problem is solved satisfactorily, but rather that there is room for improvement. There is a presence of overfitting which can be seen in the form of the training accuracy being 100%. It indicates that the model memorizes the data, which is then noticed in the model not performing nearly as well with unseen data.

We can see the effect the increase in data has on the problem, but an implementation of more advanced methods, majority voting or even cross-validation would most likely increase the accuracy and minimize the error of the model. We predicted that a training set of 64% of the total data would be enough to train the model, and the 16% validation set was supposed to help with avoiding overfitting. This split was evidently not ideal, and the methods could have benefitted from a smaller training set and therefore larger validation and test set It is observable from the confusion matrix presented in the results, that the computer has a harder time with genres, which the human ear could also classify similarly (e.g disco and pop, metal and rock).

6. Bibliography/References

For data:

- kaggle.com [1]

For information

- https://www.hoffmanacademy.com/blog/what-is-timbre-in-music-description-and-examples/ [a]
- https://www.researchgate.net/publication/330796993_Chroma_Feature_Extraction [b]
- https://www.ibm.com/topics/overfitting
- https://speechprocessingbook.aalto.fi/Representations/Zero-crossing_rate.html [c]
- https://www.sciencedirect.com/topics/engineering/zero-crossing-rate [d]
- https://machinelearningmastery.com/one-vs-rest-and-one-vs-one-for-multi-class-classification / [e]
- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html [f]
- https://h2o.ai/wiki/logistic-regression/#:~:text=Where%20is%20logistic%20regression%20used,or%20false%2C%200%20or%201. [g]

7. Appendix

Music_Genre_Classification (2) (3) (1) (2)

October 9, 2024

1 Music Genre Classification

First we will download the data set

[ ]:
```
import numpy as np
import pandas as pd
from sklearn.svm import SVC
```

[ ]:
```
data = pd.read_csv('features3.csv', sep=';')
X= data.drop(['label', 'filename', 'length'], axis=1)
y= data['label']
```

We create a Correlation Heatmap to visualise the relationships between variables in a dataset

[ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
```

[ ]:
```
le = LabelEncoder()
y_encoded = le.fit_transform(y)

plt.figure(figsize=(20, 16))
corr_matrix = X.corrwith(pd.Series(y_encoded)).abs().
 ↪sort_values(ascending=False)
sns.heatmap(corr_matrix.to_frame(), annot=True, cmap='coolwarm')
plt.title('Correlation of Features with Target')
plt.show()
```

We create a Feature Distribution histograms to get a sense of the distribution between features

[ ]:
```
X.hist(figsize=(15, 12), bins=20)
plt.suptitle('Feature Distributions', fontsize=16)
plt.show()
```

Then we plot a graph of Feature Variance

[ ]:
```
plt.figure(figsize=(10, 6))
X.var().plot(kind='bar')
```

```
plt.title('Variance of Features')
plt.ylabel('Variance')
plt.xlabel('Features')
plt.xticks(rotation=90)
plt.show()
```

Based on these visual representations we chose 10 features to use

We scale our feutures and split the data into training sets

[ ]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

[ ]:
```
# Assuming 'X' is your features and 'y' is your target labels
X = data[['spectral_bandwidth_mean', 'spectral_centroid_mean', 'mfcc2_mean',␣
 ↪'spectral_centroid_var', 'rolloff_var', 'zero_crossing_rate_var',␣
 ↪'rms_mean','mfcc13_var', 'mfcc9_mean','mfcc20_var']]
y= data['label']

# Standardize the features (important for KNN)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)


# Split the data into training and test sets (80% train/validation, 20% test)
X_train_val, X_test, y_train_val, y_test = train_test_split(X_scaled, y,␣
 ↪test_size=0.2, random_state=42)

# We further pplit the data into training, validation and test sets (64% train,␣
 ↪16% validation)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,␣
 ↪test_size=0.2, random_state=42)
```

We intialize our first machine learning method: KNN We find the optimal amount of neighbors to use

[ ]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix
```

[ ]:
```
# We define the parameter grid (range of k values we try)
param_grid = {'n_neighbors': range(1, 50), 'metric':['euclidean']}

# We initialize GridSearchCV with KNN and the validation set
grid_search = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

```
# Best value of k
best_k = grid_search.best_params_['n_neighbors']
print(f"Best k value found using validation set: {best_k}")

# We train the final KNN model with the best k on the full training set
knn_model = KNeighborsClassifier(n_neighbors=best_k)
knn_model.fit(X_train, y_train)
```

We evaluate the KNN model's performance using accuracy, classification report, and confusion matrix.

[ ]:
```
y_train_pred = knn_model.predict(X_train)
train_accuracy = accuracy_score(y_train, y_train_pred)
print(f"Train accuracy: {train_accuracy:.4f}")

print("Train Classification Report:")
print(classification_report(y_train, y_train_pred))

# We evaluate on validation set for tuning
y_val_pred = knn_model.predict(X_val)
val_accuracy = accuracy_score(y_val, y_val_pred)
print(f"Validation accuracy: {val_accuracy:.4f}")

# Display classification report for validation set
print("Validation Classification Report:")
print(classification_report(y_val, y_val_pred))

# Evaluate the final model on the test set (after tuning)
y_test_pred = knn_model.predict(X_test)
test_accuracy = accuracy_score(y_test, y_test_pred)
print(f"Test accuracy: {test_accuracy:.4f}")

# Display classification report for test set
print("Test Classification Report:")
print(classification_report(y_test, y_test_pred))
```

We plot the confusion matrix

[ ]:
```
import seaborn as sns
from matplotlib import pyplot as plt
from sklearn.manifold import TSNE
```

[ ]:
```
# Compute the confusion matrix
cm = confusion_matrix(y_test, y_test_pred)

# Plot the confusion matrix
fig, ax = plt.subplots(figsize=(8, 6))
```

```
cax = ax.matshow(cm, cmap=plt.cm.Blues)

# Add colorbar
plt.colorbar(cax)

# Add labels and titles
ax.set_xlabel('Predicted Labels')
ax.set_ylabel('True Labels')
ax.set_title('Confusion Matrix')

# Set the tick marks and labels
classes = ['blues', 'classical', 'country', 'disco', 'hiphop', 'jazz', 'metal',␣
 ↪'pop', 'reggae', 'rock']  # Adjust to match your labels
ax.set_xticks(np.arange(len(classes)))
ax.set_yticks(np.arange(len(classes)))
ax.set_xticklabels(classes)
ax.set_yticklabels(classes)

# Loop over data dimensions and create text annotations
for i in range(len(cm)):
    for j in range(len(cm[i])):
        ax.text(j, i, cm[i, j], ha="center", va="center", color="black")

plt.show()
```

We visulize the Cluster

[ ]:
```
# Assuming X_train_scaled, y_train, and knn_model have already been defined

# Apply t-SNE to the scaled training data
tsne = TSNE(n_components=2, random_state=42)
X_train_tsne = tsne.fit_transform(X_train)

le = LabelEncoder()
y_train_numeric = le.fit_transform(y_train)

# Visualize the clusters
plt.figure(figsize=(10, 8))
plt.scatter(X_train_tsne[:, 0], X_train_tsne[:, 1], c=y_train_numeric,␣
 ↪cmap='viridis', s=50)
plt.colorbar(label='Genre')
plt.title('t-SNE Visualization of KNN Classification')
plt.xlabel('t-SNE Feature 1')
plt.ylabel('t-SNE Feature 2')
plt.show()
```

We initialize Logistic Regression model

[ ]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report
from sklearn.metrics import hinge_loss, accuracy_score, log_loss
from sklearn.linear_model import LogisticRegressionCV
```

[ ]:
```
param_grid = {'C': [0.01, 0.1, 1, 10, 100]}
log_reg = LogisticRegression(multi_class='ovr', solver='lbfgs', max_iter=1000)

grid_search = GridSearchCV(log_reg, param_grid, cv=5)
grid_search.fit(X_train, y_train)


# Evaluate on the training set
best_log_reg = grid_search.best_estimator_
train_predictions = best_log_reg.predict(X_train)
train_accuracy = accuracy_score(y_train, train_predictions)

print(f'Training Accuracy: {train_accuracy:.4f}')
print("Training Classification Report:")
print(classification_report(y_train, train_predictions))


#Evaluate on the validation set
best_log_reg = grid_search.best_estimator_
val_predictions = best_log_reg.predict(X_val)
val_accuracy = accuracy_score(y_val, val_predictions)

print(f'Validation Accuracy: {val_accuracy:.4f}')
print("Validation Classification Report:")
print(classification_report(y_val, val_predictions))


#Train on the full training set and test on the test set
best_log_reg.fit(X_train_val, y_train_val)
test_predictions = best_log_reg.predict(X_test)
test_accuracy = accuracy_score(y_test, test_predictions)


print(f'Test Accuracy: {test_accuracy:.4f}')
print("Test Classification Report:")
print(classification_report(y_test, test_predictions))
```

[ ]:
```
# Compute the confusion matrix
cm = confusion_matrix(y_test, test_predictions)
```

```
# Plot the confusion matrix
fig, ax = plt.subplots(figsize=(8, 6))
cax = ax.matshow(cm, cmap=plt.cm.Blues)

# Add colorbar
plt.colorbar(cax)

# Add labels and titles
ax.set_xlabel('Predicted Labels')
ax.set_ylabel('True Labels')
ax.set_title('Confusion Matrix')

# Set the tick marks and labels
classes = ['blues', 'classical', 'country', 'disco', 'hiphop', 'jazz', 'metal',␣
 ↪'pop', 'reggae', 'rock']  # Adjust to match your labels
ax.set_xticks(np.arange(len(classes)))
ax.set_yticks(np.arange(len(classes)))
ax.set_xticklabels(classes)
ax.set_yticklabels(classes)

# Loop over data dimensions and create text annotations
for i in range(len(cm)):
    for j in range(len(cm[i])):
        ax.text(j, i, cm[i, j], ha="center", va="center", color="black")

plt.show()
```

We initialize the Random Forest model:

We use unscaled features in this method:

[ ]:
```
from sklearn.ensemble import RandomForestClassifier
```

[ ]:
```
# Split the data into training and test sets (80% train/validation, 20% test)
X_rand_train_val, X_rand_test, y_rand_train_val, y_rand_test =␣
 ↪train_test_split(X, y, test_size=0.2, random_state=42)

# We further pplit the data into training, validation and test sets (64% train,␣
 ↪16% validation)
X_rand_train, X_rand_val, y_rand_train, y_rand_val =␣
 ↪train_test_split(X_rand_train_val, y_rand_train_val, test_size=0.2,␣
 ↪random_state=42)
```

[ ]:
```
param_grid = {
    'n_estimators': [100, 200, 300],  # Number of trees in the forest
    'max_depth': [10, 20, 30],        # Maximum depth of the trees
    'min_samples_split': [2, 5],      # Minimum number of samples required to␣
 ↪split a node
```

```
    'min_samples_leaf': [1, 2]        # Minimum number of samples required at␣
 ↪each leaf node
}

rf = RandomForestClassifier(random_state=42)
grid_search = GridSearchCV(rf, param_grid, cv=5, n_jobs=-1, verbose=2)
grid_search.fit(X_rand_train, y_rand_train)


#Evaluate the model on the training set
best_rf = grid_search.best_estimator_
train_predictions = best_rf.predict(X_rand_train)
train_probabilities = best_rf.predict_proba(X_rand_train)
train_accuracy = accuracy_score(y_rand_train, train_predictions)
train_log_loss = log_loss(y_rand_train, train_probabilities)

print(f'Training Accuracy: {train_accuracy:.4f}')
print(f'Training Log Loss: {train_log_loss:.4f}')
print("Training Classification Report:")
print(classification_report(y_rand_train, train_predictions))



# Evaluate the model on the validation set
best_rf = grid_search.best_estimator_
val_predictions = best_rf.predict(X_rand_val)
val_probabilities = best_rf.predict_proba(X_rand_val)
val_accuracy = accuracy_score(y_rand_val, val_predictions)
val_log_loss = log_loss(y_rand_val, val_probabilities)

print(f'Validation Accuracy: {val_accuracy:.4f}')
print(f'Validation Log Loss: {val_log_loss:.4f}')
print("Validation Classification Report:")
print(classification_report(y_rand_val, val_predictions))


#Train on the full training set and test on the test set
best_rf.fit(X_rand_train_val, y_rand_train_val)
test_predictions = best_rf.predict(X_rand_test)
test_probabilities = best_rf.predict_proba(X_rand_test)
test_accuracy = accuracy_score(y_rand_test, test_predictions)
test_log_loss = log_loss(y_rand_test, test_probabilities)

print(f'Test Accuracy: {test_accuracy:.4f}')
print(f'Test Log Loss: {test_log_loss:.4f}')
print("Test Classification Report:")
print(classification_report(y_rand_test, test_predictions))
```

[ ]:
```
# Compute the confusion matrix
y_rand_pred= best_rf.predict(X_rand_test)
cm = confusion_matrix(y_rand_test, y_rand_pred)

# Plot the confusion matrix
fig, ax = plt.subplots(figsize=(8, 6))
cax = ax.matshow(cm, cmap=plt.cm.Blues)

# Add colorbar
plt.colorbar(cax)

# Add labels and titles
ax.set_xlabel('Predicted Labels')
ax.set_ylabel('True Labels')
ax.set_title('Confusion Matrix')

# Set the tick marks and labels
classes = ['blues', 'classical', 'country', 'disco', 'hiphop', 'jazz', 'metal',␣
 ↪'pop', 'reggae', 'rock']  # Adjust to match your labels
ax.set_xticks(np.arange(len(classes)))
ax.set_yticks(np.arange(len(classes)))
ax.set_xticklabels(classes)
ax.set_yticklabels(classes)

# Loop over data dimensions and create text annotations
for i in range(len(cm)):
    for j in range(len(cm[i])):
        ax.text(j, i, cm[i, j], ha="center", va="center", color="black")

plt.show()
```

[ ]:

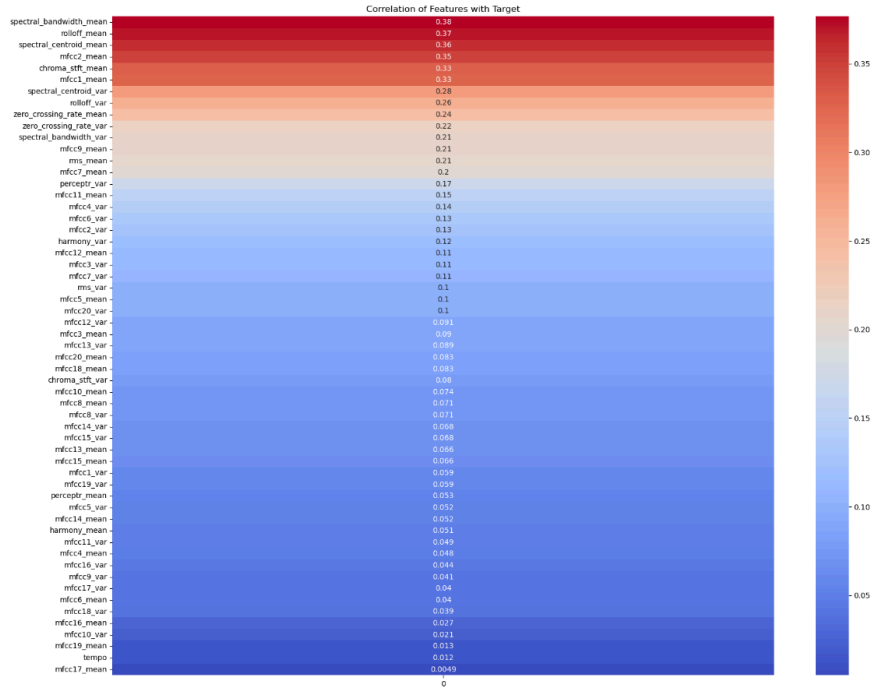
[ ]:

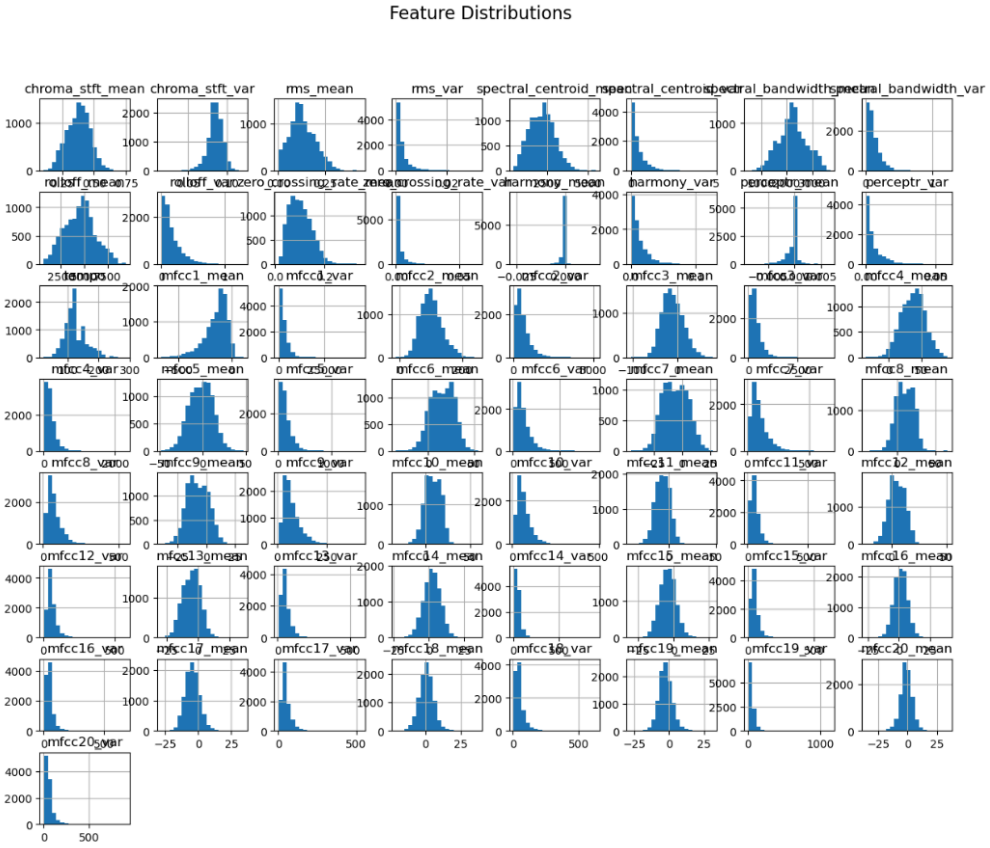
8. More interesting facts

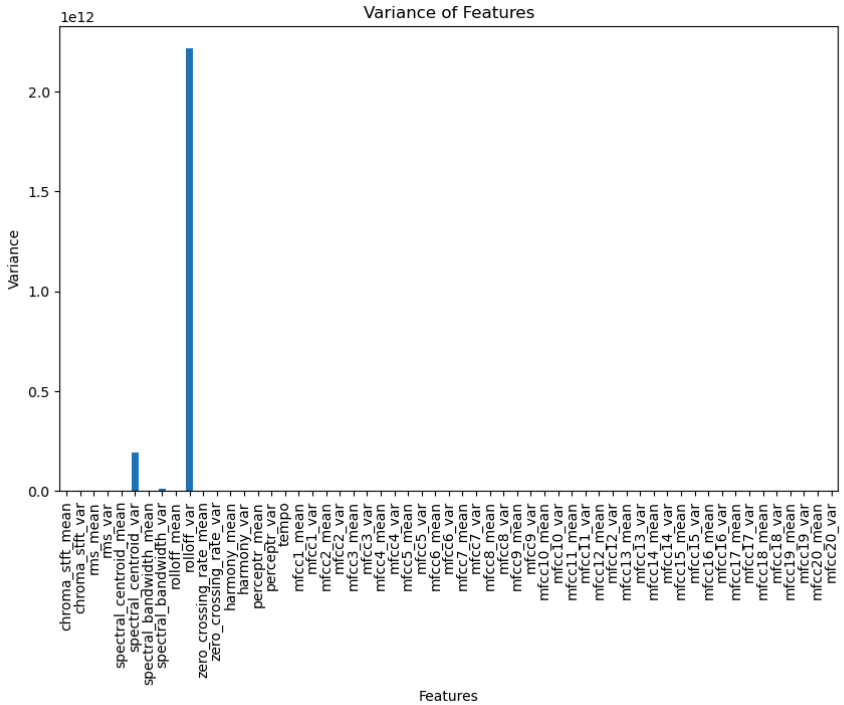
Test results for using a data set of 1000 and all the features

Method	Training set	Validation set	Test set
KNN	ACC= 1.00 PRE = 1.00 REC = 1.00 F1 = 1.00	ACC = 0.68 PRE = 0.71 REC = 0.69 F1 = 0.69	ACC = 0.665 PRE = 0.68 REC = 0.67 F1 = 0.67
Logistic Regression	ACC = 0.88 PRE = 0.88 REC = 0.88 F1 = 0.88	ACC = 0.67 PRE = 0.68 REC = 0.67 F1 = 0.67	ACC = 0.69 PRE = 0.69 REC = 0.69 F1 = 0.69
Random Forest	ACC = 1.00 PRE = 1.00 REC = 1.00 F1 = 1.00	ACC = 0.73 PRE = 0.77 REC = 0.73 F1 = 0.73	ACC = 0.66 PRE = 0.66 REC = 0.66 F1 = 0.65

Plot visualizations for choosing the features:




Feature Distributions
chroma_stft_mean
chroma_stft_var
rms_mean
rms_var
spectral_centroid_mean
spectral_centroid_var
spectral_bandwidth_mean
spectral_bandwidth_var
rolloff_mean
rolloff_var
zero_crossing_rate_mean
zero_crossing_rate_var
harmony_mean
harmony_var
perceptr_mean
perceptr_var
tempo
mfcc1_mean
mfcc1_var
mfcc2_mean
mfcc2_var
mfcc3_mean
mfcc3_var
mfcc4_mean
mfcc4_var
mfcc5_mean
mfcc5_var
mfcc6_mean
mfcc6_var
mfcc7_mean
mfcc7_var
mfcc8_mean
mfcc8_var
mfcc9_mean
mfcc9_var
mfcc10_mean
mfcc10_var
mfcc11_mean
mfcc11_var
mfcc12_mean
mfcc12_var
mfcc13_mean
mfcc13_var
mfcc14_mean
mfcc14_var
mfcc15_mean
mfcc15_var
mfcc16_mean
mfcc16_var
mfcc17_mean
mfcc17_var
mfcc18_mean
mfcc18_var
mfcc19_mean
mfcc19_var
mfcc20_mean
mfcc20_var


Variance of Features
1e12
Variance
Features
chroma_stft_mean
chroma_stft_var
rms_mean
rms_var
spectral_centroid_mean
spectral_centroid_var
spectral_bandwidth_mean
spectral_bandwidth_var
rolloff_mean
rolloff_var
zero_crossing_rate_mean
zero_crossing_rate_var
harmony_mean
harmony_var
perceptr_mean
perceptr_var
tempo
mfcc1_mean
mfcc1_var
mfcc2_mean
mfcc2_var
mfcc3_mean
mfcc3_var
mfcc4_mean
mfcc4_var
mfcc5_mean
mfcc5_var
mfcc6_mean
mfcc6_var
mfcc7_mean
mfcc7_var
mfcc8_mean
mfcc8_var
mfcc9_mean
mfcc9_var
mfcc10_mean
mfcc10_var
mfcc11_mean
mfcc11_var
mfcc12_mean
mfcc12_var
mfcc13_mean
mfcc13_var
mfcc14_mean
mfcc14_var
mfcc15_mean
mfcc15_var
mfcc16_mean
mfcc16_var
mfcc17_mean
mfcc17_var
mfcc18_mean
mfcc18_var
mfcc19_mean
mfcc19_var
mfcc20_mean
mfcc20_var