Music Genre Classification

Team 23:

Member 1: Ashish Nevan Gade
Member 2: Neha Devarapalli

Introduction

Background:

Music plays a central role in our lives, and categorizing music into predefined genres based on their audio content is a fundamental task in music information retrieval. With the exponential growth of digital music repositories and streaming platforms, the need for automated and accurate genre classification systems has become important. Genre classification allows us to organize music libraries, create personalized playlists, and even power music recommendation systems.

Motivation:

The ever-growing volume of music data necessitates scalable and efficient solutions for genre classification. While traditional machine learning approaches can achieve excellent accuracy in music genre classification, processing large music datasets can be computationally expensive. This project aims to leverage high-performance parallelization techniques to improve the efficiency of music genre classification.

Goals:

The primary goal of this project is to harness the power of parallelization to streamline the process of music genre classification. Specifically, we aim to parallelize the data preprocessing, feature extraction, model training, and evaluation stages using Python multiprocessing and Dask. By doing so, we seek to achieve a substantial reduction in processing time compared to sequential execution. Ultimately, our objective is to demonstrate the efficacy of parallelization techniques in improving the efficiency and scalability of music genre classification systems, thereby enabling faster and more robust classification of music genres on large datasets.

Methodology

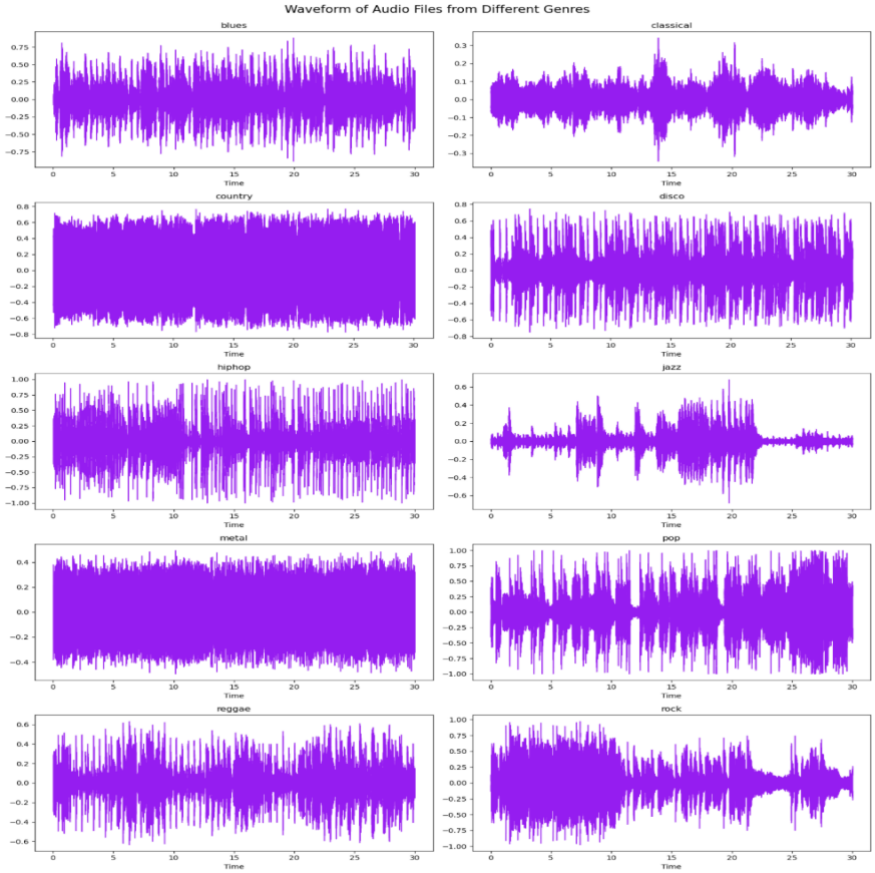
Data Exploration:

We have explored the dataset which consists of audio data (.wav) files, to get some insights about its composition and characteristics. On loading the audio signal using librosa, we found that it stores the audio as a NumPy array which is useful in our case since it is well suited for parallel computing, thanks to vectorized operations.

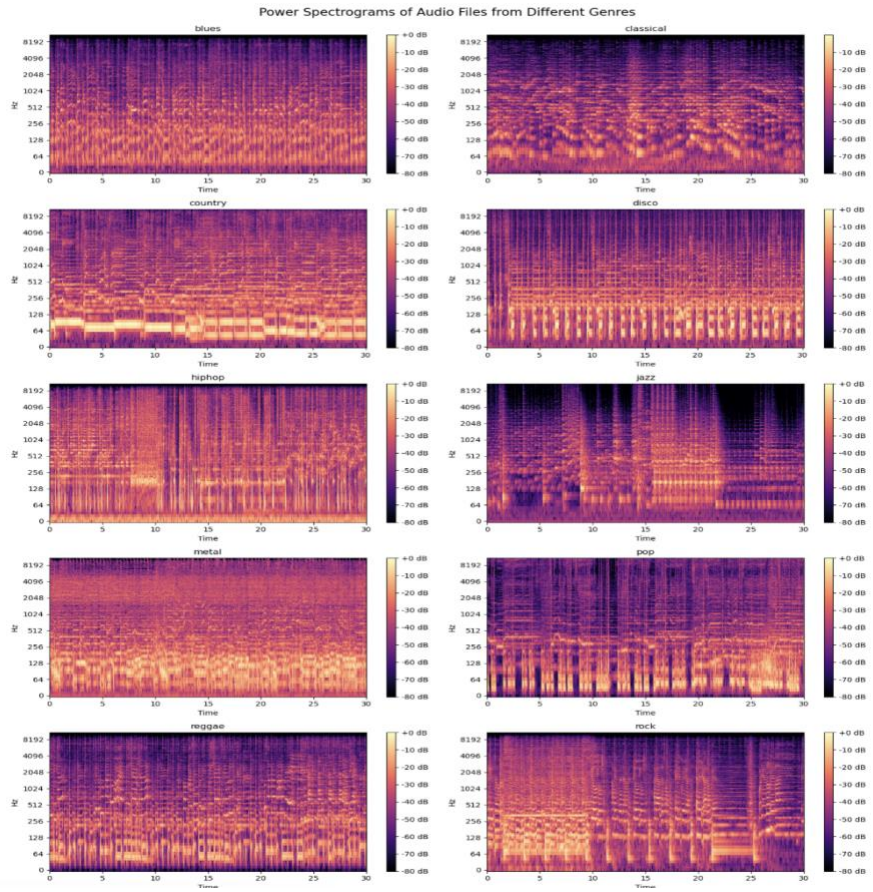
```
y: [0.02072144 0.04492188 0.05422974 ... 0.06912231 0.08303833 0.08572388]

Datatype of y: <class 'numpy.ndarray'>
```

We then performed some visualizations on the audio files to shed light on any characteristics or features which are unique to each genre so that it will then help us classify the genres more accurately. The visualizations are as follows:

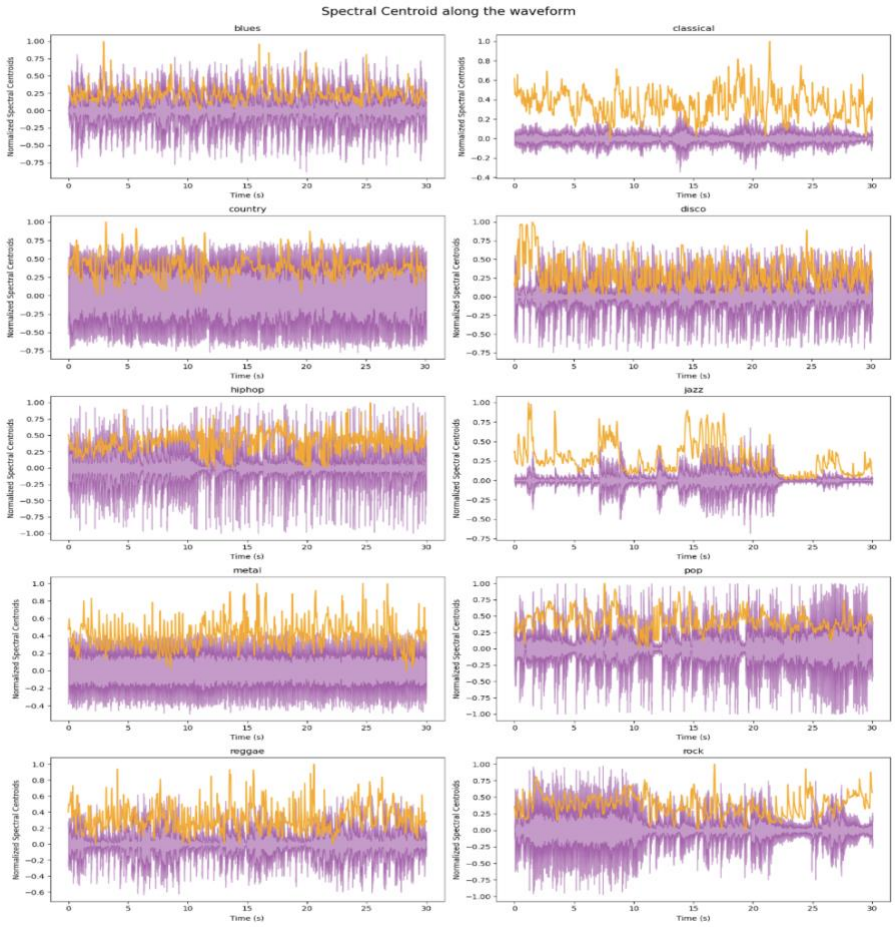
1. Waveforms: We visualized the 2D representation of audio files from each genre. They are plotted using librosa.display.waveshow() for visual inspection of raw audio data.



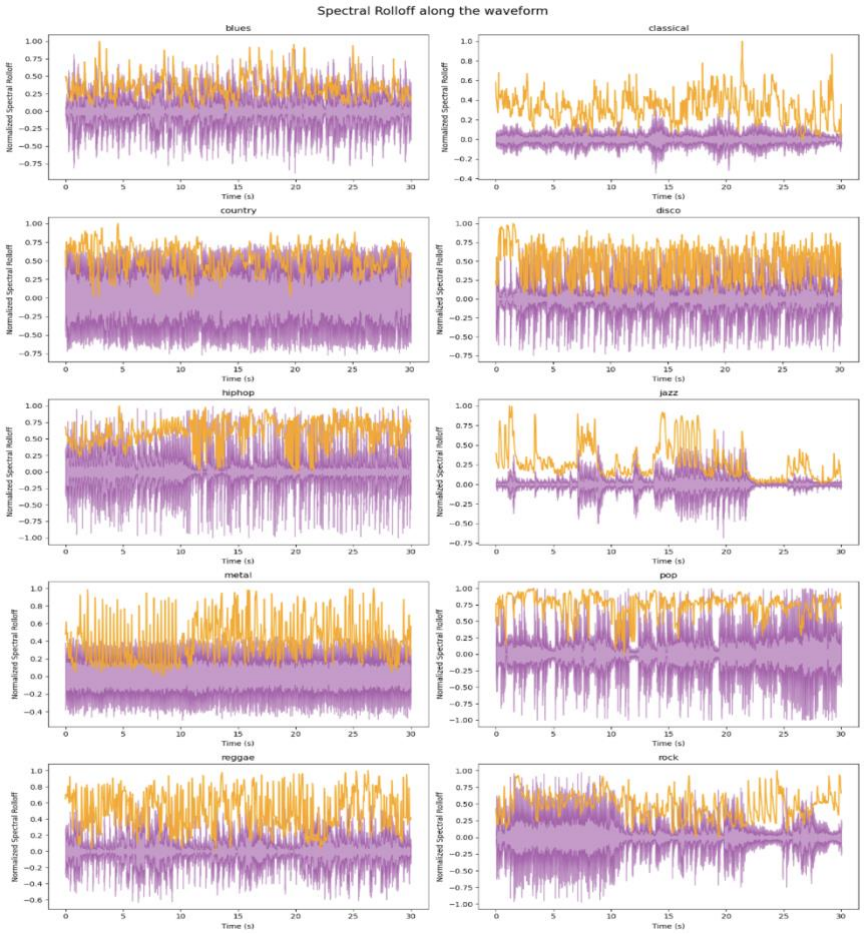
2. Power Spectrograms: It is a visual representation of the spectrum of frequencies of a signal as it varies with time.



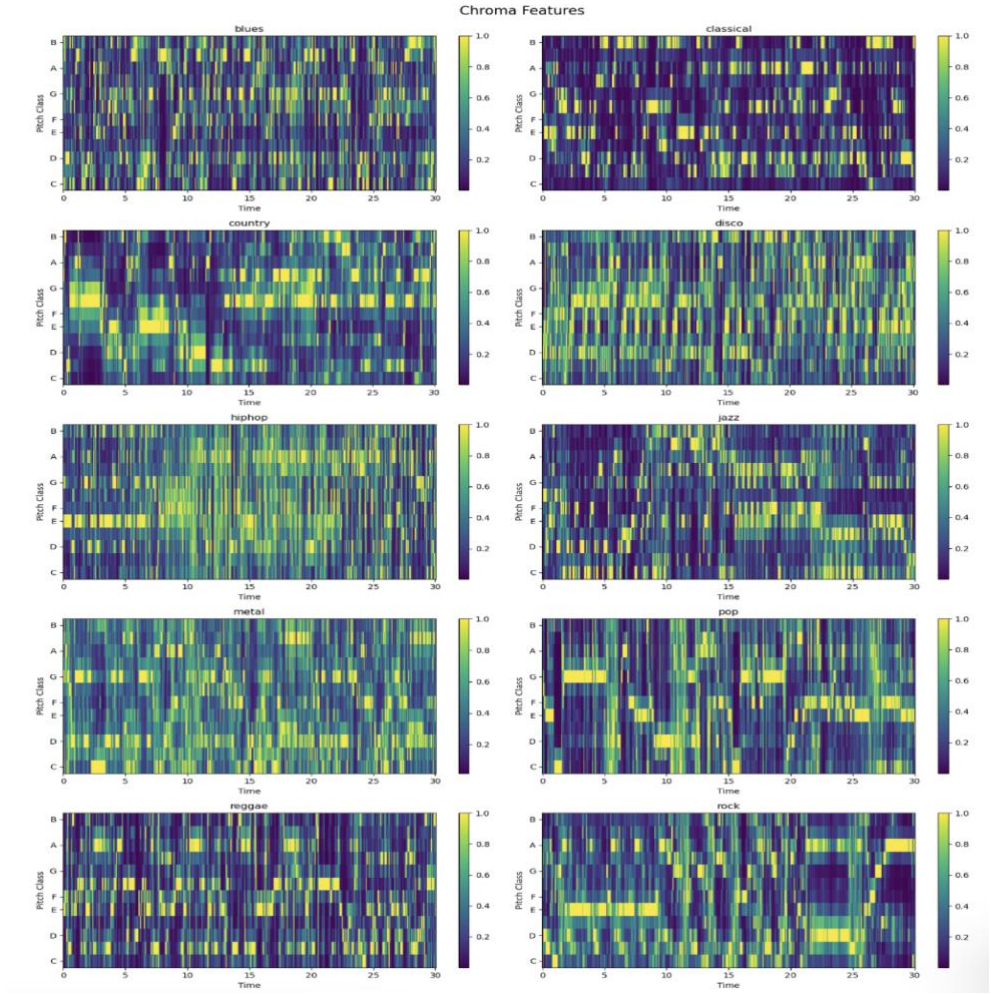
3. Spectral Centroid: It indicates where the “center of mass” for a sound is located and is calculated as the weighted mean of the frequencies present in the sound.



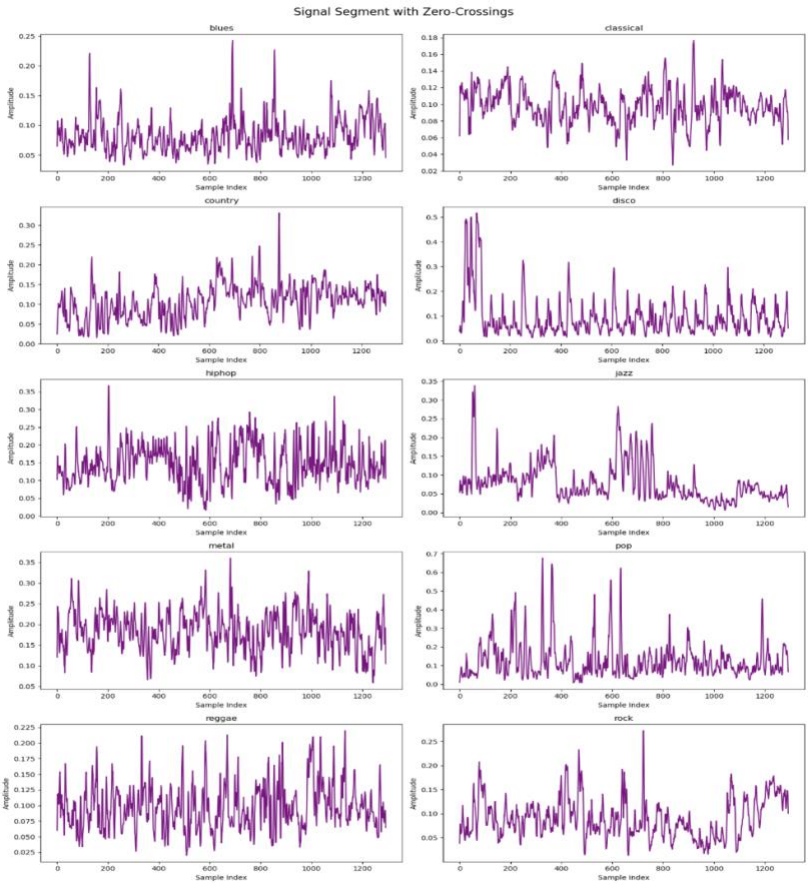
4. Spectral Rolloff: This is a measure of the shape of the signal. It represents the frequency below which a specified percentage of the total spectral energy lies.



5. Chroma Features: It is a powerful representation for music audio since in this the entire spectrum is projected onto 12 bins representing the 12 distinct semitones (or chroma) of the musical octave.



6. Zero Crossings: This is a technique used to analyze audio signals by identifying points where the waveform crosses the zero-amplitude threshold.



Feature Extraction

We have created a feature extraction pipeline which efficiently processes raw audio data using parallelization to create a comprehensive dataset for music genre classification. It extracts relevant features from audio files and compiles them into a structured CSV file, providing essential information for subsequent model training and analysis.

Our pipeline extracts a diverse set of features from each audio chunk, and some of the main features which were extracted are:

- Zero Crossing Rate
- Spectral Centroids
- Chroma Features
- Tempo
- Spectral Rolloff
- Spectral Flux
- MFCC Coefficients

These features provide insights into the temporal, spectral and timbral characteristics of the audio signals, enabling comprehensive analysis and classification.

```python
# Configuration constants
NUMBER_OF_MFCC = 20
N_FFT = 2048
HOP_LENGTH = 512
BLOCK_LENGTH = 11
FRAME_LENGTH = 6616
HOP_LENGTH_STREAM = 6615

def extract_features(trial_audio_file_path: str) -> pd.DataFrame:
    #Extracting features for each chunk of the audio file
    df0 = extract_features_per_chunk(trial_audio_file_path)

    #Concatenate dataframes
    df0 = pd.concat(df0)
    return df0

def extract_features_per_chunk(audio_file_path):
    stream = librosa.stream(audio_file_path, block_length=BLOCK_LENGTH, frame_length=FRAME_LENGTH, hop_length=HOP_LENGTH_STREAM)
    processed_chunks = []
    file_name = os.path.basename(audio_file_path)
    for i, y_block in enumerate(stream):
        #Trimming starting and trailing silences
        y_trimmed, _ = librosa.effects.trim(y_block)

        #Extracting features from trimmed audio
        df = extract_feature_means(y_block, f'{file_name}.{i}', 22050)
        df['label'] = os.path.basename(os.path.dirname(audio_file_path))
        processed_chunks.append(df)
    return processed_chunks
```

To enhance dataset granularity and improve model accuracy, we have partitioned each audio file into smaller segments or ‘chunks’. Each audio file in the dataset is 30 seconds long, we have further divided the audio into 10 chunks, with each chunk representing a 3-second segment of the original audio. We observed that this granularity allowed us to have finer feature extraction and capture nuances within the audio data more effectively which at last helped us improve the accuracy by 10%. We also checked for and trimmed any starting at trailing silences from the audio files to improve the quality of data. Part of this code was inspired by [2]. The code was extended to use dask and performing analysis on chunks of audio by us.

We have utilized parallelization using Dask to optimize the processing of raw audio data. By distributing feature extraction across multiple CPU cores, we can leverage the parallel computing capabilities to expedite the extraction process and enhance the overall efficiency.

We have also compared all the different Dask schedulers, including synchronous, threads, processes, and distributed, to determine the most efficient scheduler for parallel feature extraction. This comparison involves evaluating the execution time of feature extraction tasks under each scheduler configuration.

```python
#Parallel feature extraction using the Dask Distributed Scheduler
def extract_features_distributed(files, n_workers):
    cluster = LocalCluster(n_workers=n_workers)
    client = Client(cluster)

    res = []
    for file_path in files:
        f = delayed(extract_features)(file_path)
        res.append(f)

    extracted_features = delayed(pd.concat)(res)
    with Profiler() as prof, ResourceProfiler() as rprof, CacheProfiler() as cprof:
        df = compute(extracted_features)
    visualize([prof, rprof, cprof])

    client.close()
    cluster.close()

    return df
```

```python
#Range of CPUs to test
num_workers = [3, 7, 10, 14, 20, 28]

#List to store execution times
execution_times = []

#Extracting features for each configuration and recording execution time
for n_workers in num_workers:
    start = time()
    df = extract_features_distributed(files[:20], n_workers)
    duration = time() - start
    execution_times.append(duration)
    print(f"FEATURE EXTRACTION: Number of CPUs: {n_workers}\tTime taken: {duration:.2f} seconds")
```

After identifying the Dask distributed scheduler as the most effective option for our feature extraction pipeline, we systematically tested a range of 'num_workers' values to optimize performance. By varying the number of workers from 3 to 28, we aimed to minimize execution time while maximizing parallelization benefits. Through this iterative experimentation, we identified the optimal 'num_workers' value that achieved the fastest feature extraction times.
Additionally, we leveraged Dask's delayed functionality which allowed us to create delayed objects for each 'extract_features()' function call, which postpones their execution until explicitly triggered. This deferred execution enables the scheduler to optimize task scheduling across available workers, minimizing idle time and maximizing resource utilization which in turn enhances the processing speed.

Exploratory Data Analysis

```
data = pd.read_csv(outfile_path)
data.head()
```

	file_name	zero_crossing_rate_mean	zero_crossings_mean	spectrogram_mean	mel_spectrogram_mean	harmonics_mean	perceptual_shock_wave
0	blues.00000.wav.0	0.082332	6056	-39.814990	-54.827530	-0.000040	-0.
1	blues.00000.wav.1	0.083646	6144	-35.065994	-42.183130	-0.000019	0.
2	blues.00000.wav.2	0.073553	5397	-41.551838	-53.736260	-0.000081	-0.
3	blues.00000.wav.3	0.069974	5166	-37.317627	-46.955624	0.000001	0.
4	blues.00000.wav.4	0.082899	6094	-43.032400	-56.932766	0.000077	-0.

5 rows × 110 columns

During the dataset examination, we conducted various operations to comprehend its structure and characteristics thoroughly. Firstly, we assessed the distribution of audio files across different genres to identify any potential biases. Fortunately, we observed an even distribution across all genres, mitigating concerns regarding dataset imbalance.

```
data.dtypes
```

```
file_name                 object
zero_crossing_rate_mean  float64
zero_crossings_mean        int64
spectrogram_mean         float64
mel_spectrogram_mean     float64
                           ...
mfcc_accelerate_18_mean  float64
mfcc19_mean              float64
mfcc_delta_19_mean       float64
mfcc_accelerate_19_mean  float64
label                     object
Length: 110, dtype: object
```

```
# Display count of data rows for each genre
genre_counts = data['label'].value_counts()
print(genre_counts)
```

```
label
blues        1000
classical    1000
country      1000
disco        1000
hiphop       1000
jazz         1000
metal        1000
pop          1000
reggae       1000
rock         1000
Name: count, dtype: int64
```

Furthermore, we utilized the ‘data.describe()’ function to generate summary statistics, including mean, standard deviation, minimum, maximum, and quartile values for each feature. This analysis provided valuable insights into the range and variability of feature values, aiding in our understanding of the dataset’s composition.

Upon inspecting the dataset’s shape, we confirmed that it consists of 10,000 samples with 110 features, as expected from the feature extraction phase. Additionally, we performed a

comprehensive check for null values and verified the data types of each column. We identified that the label column, denoting music genres, is currently of object data type and needs to be encoded for compatibility with modeling tasks.

Data Preprocessing

We divided the dataset into features (X) and the target variable (y), where the target variable represents the music genre labels ('blues', 'classical', 'country', 'disco', 'hiphop', 'jazz', 'metal', 'pop', 'reggae', 'rock').

```
#Splitting data into features(X) and target (y)
target_column = 'label'
features = data.drop(columns=[target_column, 'file_name'])
target = data[target_column]

#Standardizing features
scaler = StandardScaler()
X = scaler.fit_transform(features.astype(float))

#Updating variables for clarity
y = target
```

We also standardized the features using the StandardScaler, which transforms the data to have a mean of 0 and a standard deviation of 1. Standardization is beneficial for algorithms that rely on distance metrics, such as K_Nearest Neighbors or Support Vector Machines, as it ensures that all features have the same scale.

At last, we encoded the target labels using the LabelEncoder, which transforms categorical labels into numerical values required for model training.

Training and Classification

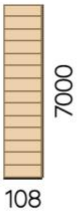
In this phase, we employed a variety of classifiers to assess their performance on the dataset. Initially, we instantiated and evaluated each model sequentially to establish a baseline for comparison. The models include K-Nearest Neighbors (KNN), Random Forest, Support Vector Machine (SVM), Neural Network, XGBoost Classifier, and XGBoost Classifier with a Random Forest objective. We adopted a 70/30 train-test split for the data.

After the above-described evaluation, we found the model which gave us the best accuracy, and XGBoost was the winner. Hence, we decided to perform parallelization on XGBoost training. We employed ‘xgboost.dask’, a distributed variant of the XGBoost library specifically designed for parallel and distributed computing environments. It leverages Dask library’s capabilities to distribute the computation of XGBoost models across multiple workers in a cluster and automatically handles data partitioning and scheduling of tasks, ensuring that each worker operates on its portion of the data without overloading the system.

We utilized Dask arrays to parallelize computation of training and testing datasets. First we transformed our NumPy arrays into Dask arrays, configuring the chunks parameter to optimize memory usage and parallelism. Then we established a Dask client with the desired number of workers to manage and distribute the computation across the cluster. Then we utilized Dask's distributed scheduler to train the XGBoost classifier in parallel.

```
# create dask array with appropriate chunk size
parallel_execution_times = []
from dask import array as da
da_X_train, da_y_train = da.from_array(X_train, chunks=(500,108)), da.from_array(y_train, chunks=(500,))
da_X_test, da_y_test = da.from_array(X_test, chunks=(500,108)), da.from_array(y_test, chunks=(500))
da_X_train
```

	Array	Chunk
Bytes	5.77 MiB	421.88 kiB
Shape	(7000, 108)	(500, 108)
Dask graph	14 chunks in 1 graph layer	
Data type	float64 numpy.ndarray	



Description of the Dataset

Dataset used for this project is the GTZAN dataset, which is considered the MNIST of audio dataset for music genre classification. The dataset consists of 10 classes of music genres, each having 100 samples of 30 second audio files in .wav format. The 10 genres are [Blues, Classical, Country, Disco, Hiphop, Jazz, Metal, Pop, Reggae, Rock]. Each genre is a folder and within each folder are .wav audio files named in the format {genre}.{00XX}.wav, where XX is from 00-99. The size of this dataset is 2.47 Gigabytes. To download this dataset, Git must be initialized with Large File Storage support.

Data Source

https://huggingface.co/datasets/marsyas/gtzan

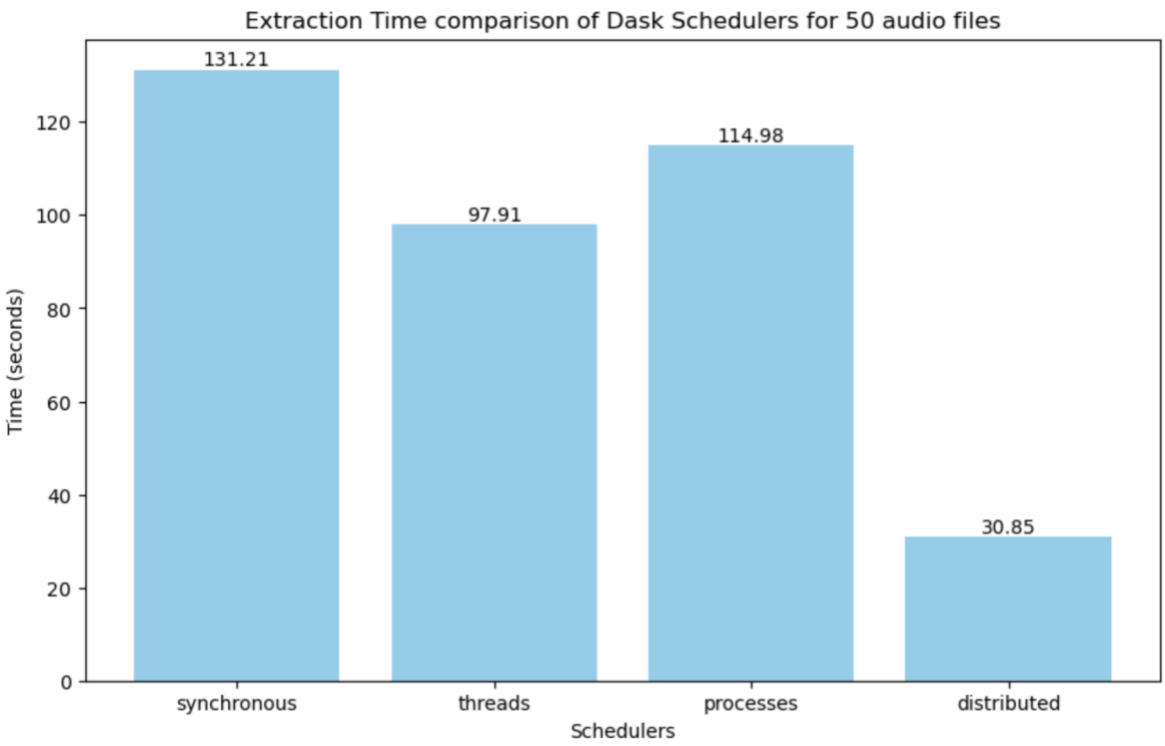
Results and Analysis

1. Parallelization techniques for feature extraction:
    a. Comparison between different Dask schedulers:

```python
#Setting the number of workers
n_workers = min(4, CPU_COUNT)

def try_scheduler(scheduler_name, files):
    print(f"Scheduler: {scheduler_name}")
    start = time()

    if scheduler_name == "distributed":
        #Using Local Cluster for distributed scheduler
        cluster = LocalCluster(n_workers=n_workers)
        client = Client(cluster)
    else:
        dask.config.set(scheduler=scheduler_name, num_workers=n_workers)  # Set the scheduler for t

    res = []
    for file_path in files:
        f = delayed(extract_features)(file_path)
        res.append(f)

    extracted_features = delayed(pd.concat)(res)
    df = compute(extracted_features)

    if scheduler_name == "distributed":
        cluster.close()  # Close the local cluster

    duration = time() - start
    print(f"Extracted features in {duration} seconds\n")
    return duration
```

```python
# Comparison between different dask schedulers to see which is best with available Dask schedulers
schedulers = ["synchronous", "threads", "processes", "distributed"]
execution_times = []
for scheduler in schedulers:
    execution_time = try_scheduler(scheduler, files[:10])
    execution_times.append(execution_time)
```

We conducted this comparison with 50 audio files. The feature extraction process utilized methods from the Librosa library, and we employed Dask's delayed and compute functions to leverage parallelization. We tested four different Dask schedulers: synchronous, threads, processes, and distributed and got the following results.



Analysis:

- Synchronous scheduler: This scheduler executes tasks sequentially without parallelism, resulting in the longest execution time of 131.21 seconds. Each task had to wait for the previous one to complete, leading to suboptimal performance.
- Threads scheduler: The threads scheduler allows concurrent execution within a single Python process, leveraging multiple CPU cores if available. While this concurrency improved performance compared to the synchronous scheduler, the presence of Python's Global Interpreter Lock (GIL) possibly limited the efficiency.
- Processes scheduler: With this, the tasks were executed concurrently using separate Python processes. However, the overhead of process creation and management increased the execution time slightly as compared to threads scheduler.
- Distributed scheduler: This scheduler is much faster than single machine schedulers. It completed the task the fastest in 30.85 seconds. This scheduler is optimized for distributed computing across multiple CPU cores or machines. It efficiently leveraged parallelism, distributing the workload across workers, and optimizing task scheduling and resource allocation.

b. Comparison between Python multiprocessing methods:

```
start = time()
with mp.Pool(4) as p:
    for i in files[:50]:
        res = p.apply(extract_features, (i,))
print(f'took {time()-start} seconds')
```

```
took 151.48904848098755 seconds
```

```
start = time()
with mp.Pool(4) as p:
    for i in files[:50]:
        res = p.apply_async(extract_features, (i,))
        res.get()
print(f'took {time()-start} seconds')
```

```
took 152.5634536743164 seconds
```

```
start = time()
with mp.Pool(4) as p:
    res = p.map(extract_features, files[:50])
print(f'took {time()-start} seconds')
```

```
took 107.38028383255005 seconds
```

```
start = time()
with mp.Pool(4) as p:
    res = p.map_async(extract_features, files[:50])
    res.get()
print(f'took {time()-start} seconds')
```

```
took 109.47061848640442 seconds
```

```
SUBSET_SIZE = 50
```

```
# Serial code
start = time()
for file in files[:SUBSET_SIZE]:
    df = extract_features(file)
end = time()
serial_extraction_time = end-start
print(f'serial execution took {serial_extraction_time} seconds')
```

```
serial execution took 131.56800079345703 seconds
```

We used multiprocessing techniques with Python's multiprocessing module to parallelize the feature extraction process. This was also performed on 50 audio files. Four different multiprocessing methods were evaluated: apply(), apply_async(), map(), and map_async().

Analysis:

- apply() method: This method was used to sequentially apply the extract_features() function to each audio file in the dataset. It took 151.489 seconds to complete the task. This method executes tasks sequentially, leading to suboptimal performance due to lack of parallelism.
- apply_async() method: With this method, tasks were submitted asynchronously to the multiprocessing pool, allowing for parallel execution. However, the implementation still resulted in a lengthy execution time of 152.56 seconds due to the overhead of task submission and retrieval.
- map() method: This method applied the extract_features() function to each audio file in parallel, leading to improved performance compared to the previous methods. It completed the task in 107.38 seconds, demonstrating better efficiency by leveraging parallelism.
- map_async() method: This method has parallelized task execution like map() but is allowed for asynchronous submission and retrieval of tasks. Despite this, it resulted in a slightly longer execution time of 109.47 seconds compared to the map() method.

```
start = time()
jobs = []
for i in files[:50]:
    p = mp.Process(target=extract_features, args=(i,))
    p.start()
    jobs.append(p)
for job in jobs:
    job.join()
p.close()
print(f'took {time()-start} seconds')
```

```
took 80.06569623947144 seconds
```

Among all multiprocessing methods evaluated, Process class was best in terms of performance.

Analysis: Since feature extraction task is a mixture of non-vectorized librosa module and vectorized numpy functions, just using threads is not enough. Therefore, Process class outperforms Pool class here. However, a much significant speedup can be observed when using Dask Distributed Cluster with Distributed Scheduler which is demonstrated below.

While multiprocessing approaches can improve performance by leveraging multiple CPU cores, it still did not perform that much better than serial computation which took only 131.57 seconds. Whereas Dask distributed scheduler combined

with delayed operation took only a fraction of its time. Overall, the speedup and efficiency achieved for the best configuration is as follows:

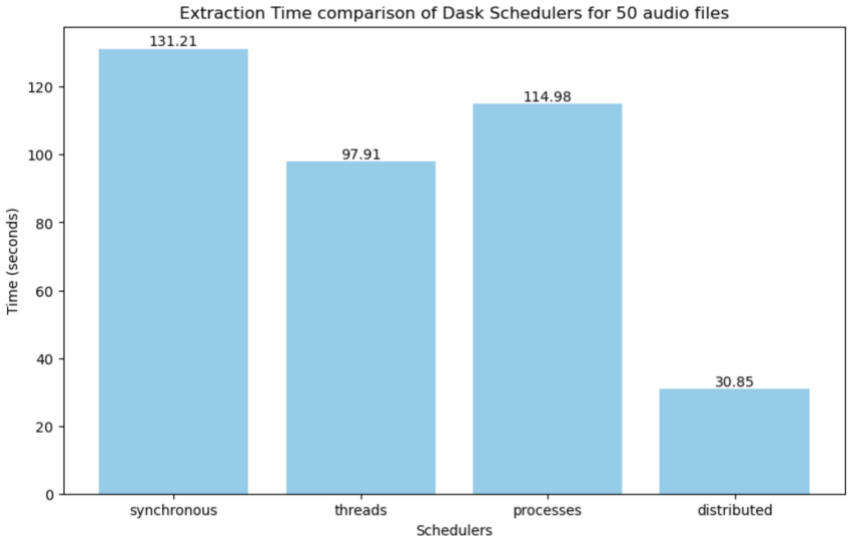
```
#Calculating the speedup achieved by using parallel on feature extraction
parallel_extraction_time = min(execution_times)
speedup_feature_extraction = serial_extraction_time / parallel_extraction_time

print("Speedup achieved for feature extraction: ", round(speedup_feature_extraction))

#Calculating the efficiency achieved by using parallel on feature extraction
efficiency_feature_extraction = speedup_feature_extraction / n_workers

print("Efficiency achieved for feature extraction: ", round(efficiency_feature_extraction))
```

```
Speedup achieved for feature extraction:  4
Efficiency achieved for feature extraction:  1
```

Out of the 4 schedulers in Dask, we found the fastest and best configuration to be Dask distributed scheduler with Dask delayed feature extraction operation with 14 workers as you can see below:

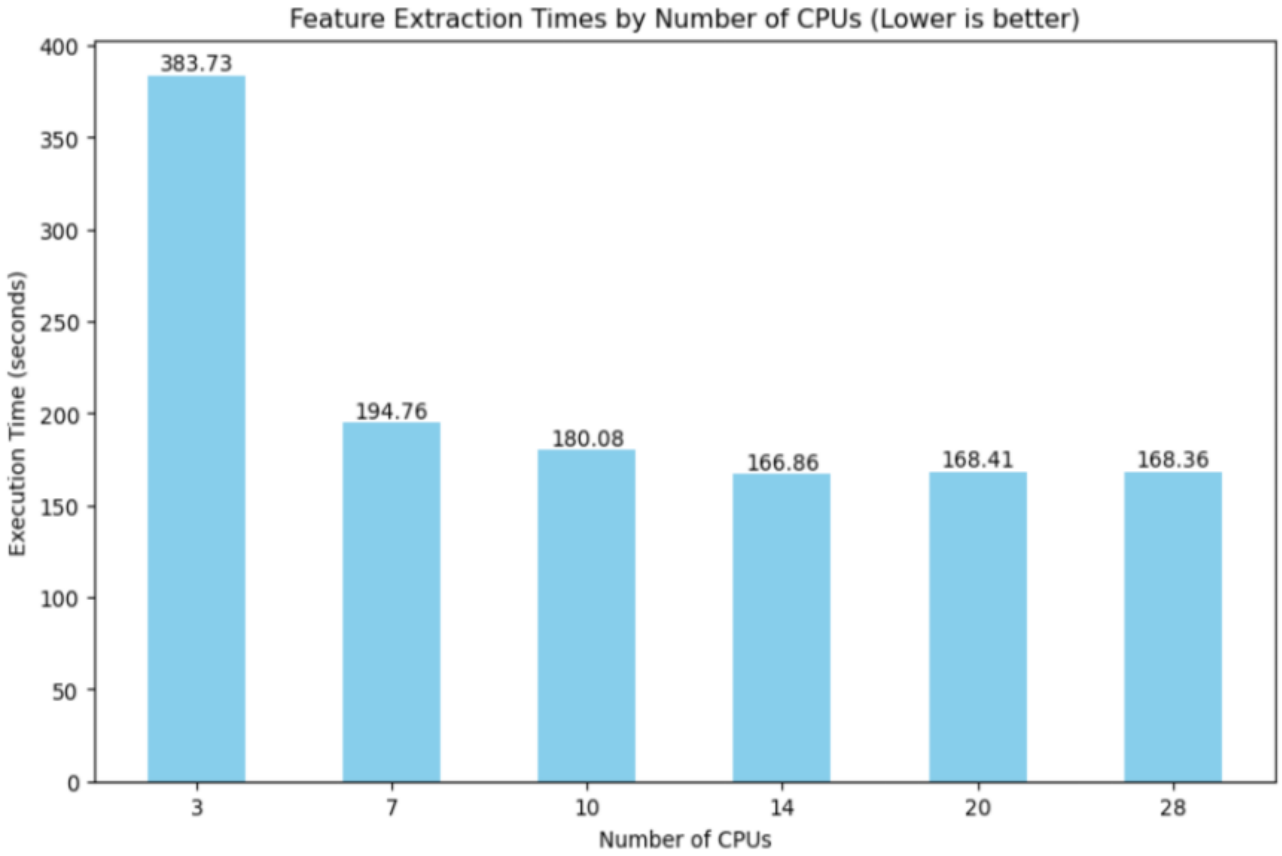
```
#Setting the number of workers
n_workers = min(4, CPU_COUNT)

def try_scheduler(scheduler_name, files):
    print(f"Scheduler: {scheduler_name}")
    start = time()

    if scheduler_name == "distributed":
        #Using Local Cluster for distributed scheduler
        cluster = LocalCluster(n_workers=n_workers)
        client = Client(cluster)
    else:
        dask.config.set(scheduler=scheduler_name, num_workers=n_workers)  # Set the scheduler for threads or processes or synchronous

    res = []
    for file_path in files:
        f = delayed(extract_features)(file_path)
        res.append(f)

    extracted_features = delayed(pd.concat)(res)
    df = compute(extracted_features)

    if scheduler_name == "distributed":
        cluster.close()  # Close the local cluster

    duration = time() - start
    print(f"Extracted features in {duration} seconds\n")
    return duration
```

Using the code above, we tried all the 4 schedulers available in Dask and recorded the execution times. The observed values are visualized below:

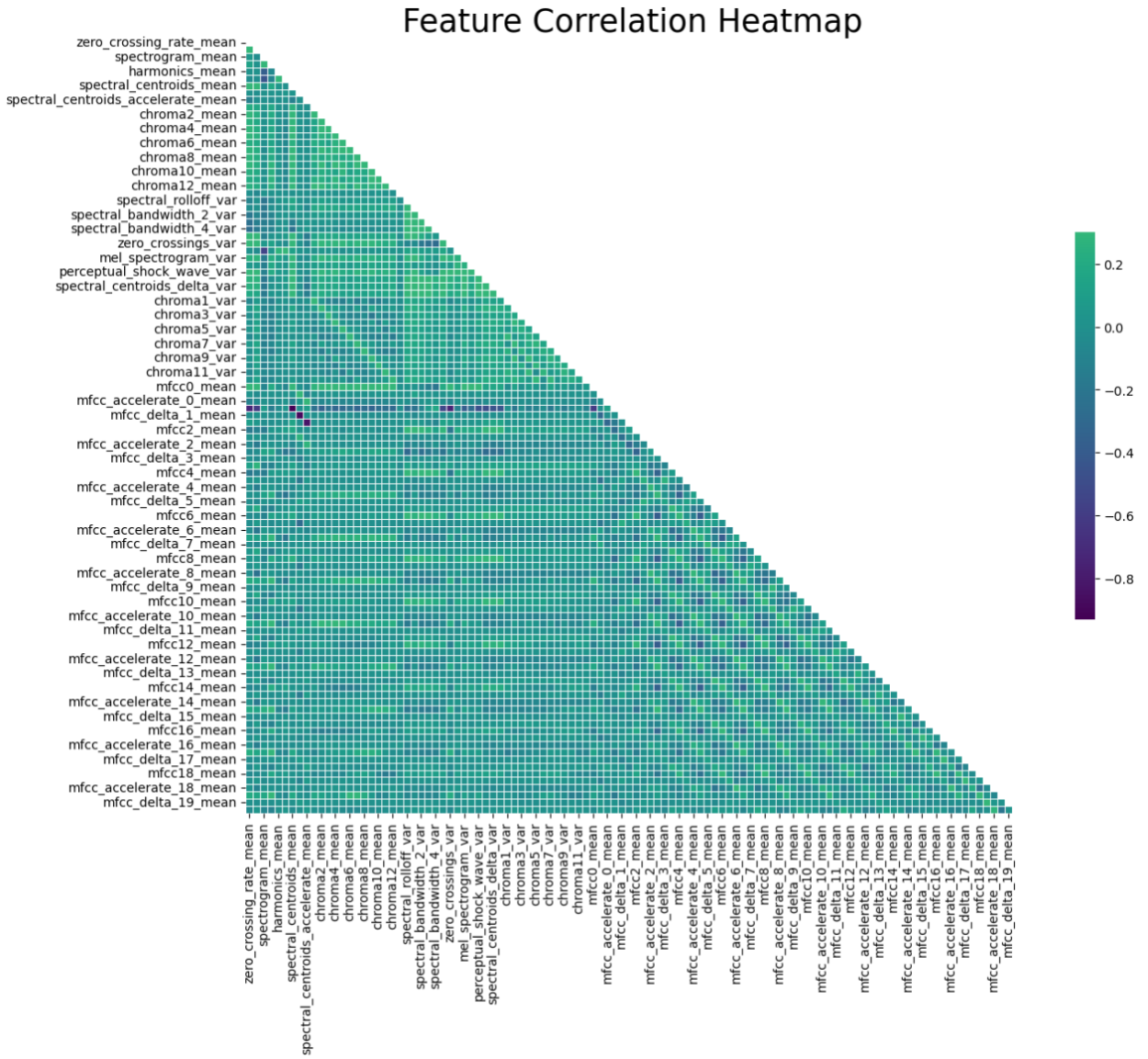


Note that these values are for extracting features for only 50 files.

Furthermore, we have used various values for n_workers on distributed scheduler to extract features of all the 10K samples. The observed values are visualized below:



2. Feature Correlation Heatmap:



Plotted the above feature correlation heatmap to analyze any existing highly correlated features. But as you can see, most of them are moderate or weak correlations which is beneficial due to reduced redundancy.

3. Training and Classification:
   a. Evaluating different ML models for the one with best performance

```
def evaluate_model(model, model_name = "", cr = False):
    model.fit(X_train, y_train)
    y_hat = model.predict(X_test)
    print('Accuracy', model_name, ':', round(accuracy_score(y_test, y_hat), 5), '\n')
    if cr:
        target_names = sorted(set(label_encoder.inverse_transform(y_encoded)))
        print(classification_report(y_test, y_hat, target_names=target_names))
```

```
# KNN
knn = KNeighborsClassifier(n_neighbors=19)
evaluate_model(knn, "KNN")

# Random Forest
rforest = RandomForestClassifier(n_estimators=1000, max_depth=10, random_state=0)
evaluate_model(rforest, "Random Forest")

# Support Vector Machine
svm = SVC(decision_function_shape="ovo")
evaluate_model(svm, "SVM")

# Neural Nets
nn = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(5000, 10), random_state=1)
evaluate_model(nn, "Neural Network")

# Cross Gradient Booster
xgb = XGBClassifier(n_estimators=1000, learning_rate=0.05)
evaluate_model(xgb, "XGBoost Classifier")

# Cross Gradient Booster (Random Forest)
xgbrf = XGBRFClassifier(objective= 'multi:softmax', n_rounds=100)
evaluate_model(xgbrf, "XGBoost Classifier (Random Forest)")
```

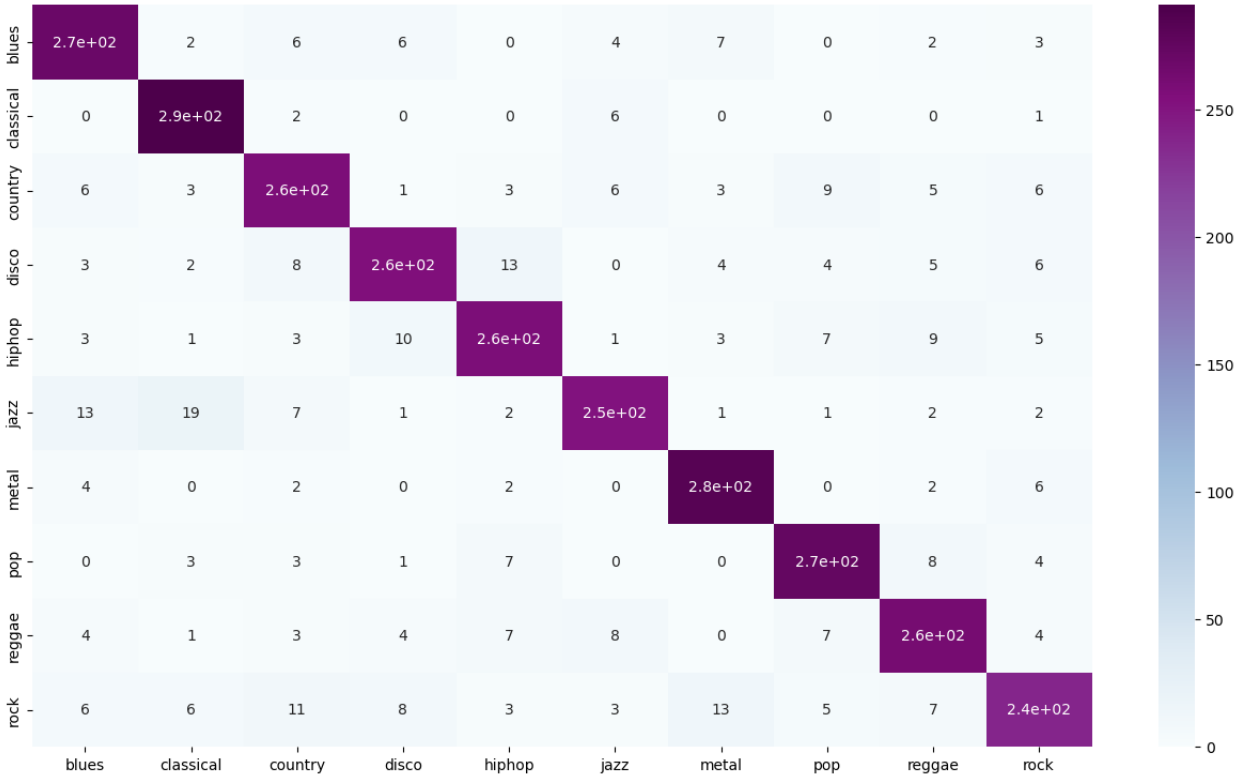
```
Accuracy KNN : 0.76533

Accuracy Random Forest : 0.796

Accuracy SVM : 0.79767

Accuracy Neural Network : 0.83567

Accuracy XGBoost Classifier : 0.887

Accuracy XGBoost Classifier (Random Forest) : 0.74867
```

The above displayed results reveal that XGBoost Classifier stands out as the top-performing model. Its ability to leverage ensemble learning and handle complex relationships within the data leads to superior accuracy compared to other models. Part of this code was taken from [1].
Classification report of the XGBoost model is as follows:

```
Training set predictions: [8 2 0 ... 3 0 7]
Testing set predictions: [1 2 6 ... 6 3 4]
Training accuracy: 0.9992857142857143
Training Classification Report:
              precision    recall  f1-score   support

       blues       1.00      1.00      1.00       700
   classical       1.00      1.00      1.00       700
     country       1.00      1.00      1.00       700
       disco       1.00      1.00      1.00       700
      hiphop       1.00      1.00      1.00       700
        jazz       1.00      1.00      1.00       700
       metal       1.00      1.00      1.00       700
         pop       1.00      1.00      1.00       700
      reggae       1.00      1.00      1.00       700
        rock       1.00      1.00      1.00       700

    accuracy                           1.00      7000
   macro avg       1.00      1.00      1.00      7000
weighted avg       1.00      1.00      1.00      7000

Testing accuracy: 0.8806666666666667
Testing Classification Report:
              precision    recall  f1-score   support
...
    accuracy                           0.88      3000
   macro avg       0.88      0.88      0.88      3000
weighted avg       0.88      0.88      0.88      3000
```

The model predicts the genres of the training set with extremely high accuracy, achieving a training accuracy of about 99.93%. This suggests that the model is overfitting but effectively learned the patterns and features present in the training data. Because it has good testing accuracy, we can ignore the overfitting for now and resolve that during hyperparameter tuning using learning_rate parameter.
Precision, recall and F1-score metrics for each genre class are all at 1.00 indicating perfect classification performance for each genre on the training set predictions.

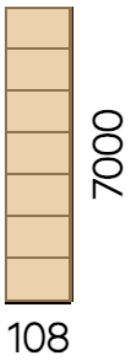
The model also performs well on the testing set, achieving an accuracy of 88.07%. While being slightly lower than the training accuracy, this is still good as it indicates strong generalization performance of the model.
Precision, recall and F1-score metrics for each genre class in the testing set are also high, averaging around 88% which indicates the model's performance on unseen data remains consistent and reliable.



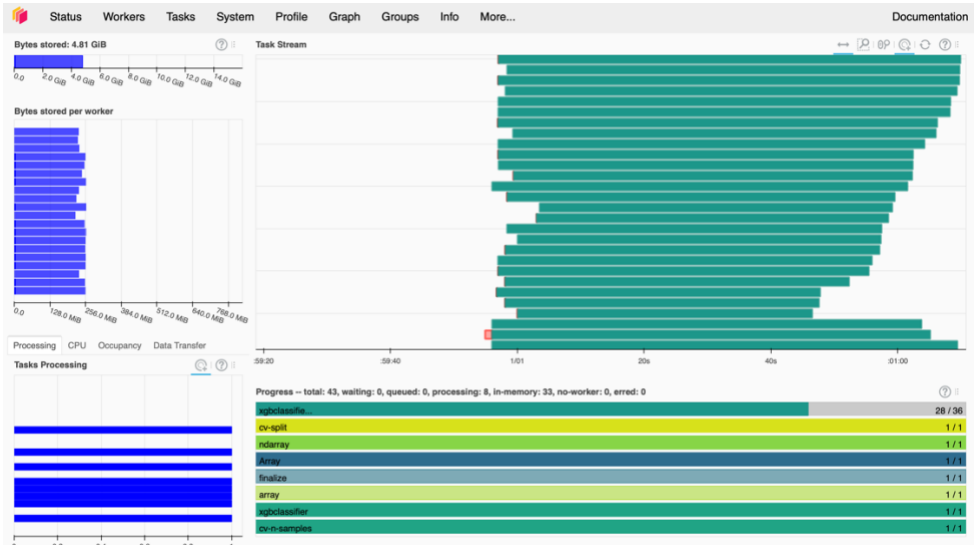
b. Parallelization using xgboost.dask

- The following steps are taken to speedup xgboost classifier training process:
- XGBoost.dask is used as a drop-in replacement of XGBoost Classifier.
- Attached Dask distributed client to xgboost.client attribute.
- After observing the dask dashboard with different chunk sizes, we found out that chunk size 1000 works best.
- Dask distributed client is used to train XGBoost Classifier in parallel.
- We experimented with n_workers set to 2, 4, 7, 14, 21, 28.
- This approach did not provide any speedup to the serial execution, so we moved on to try parallization using joblib module.

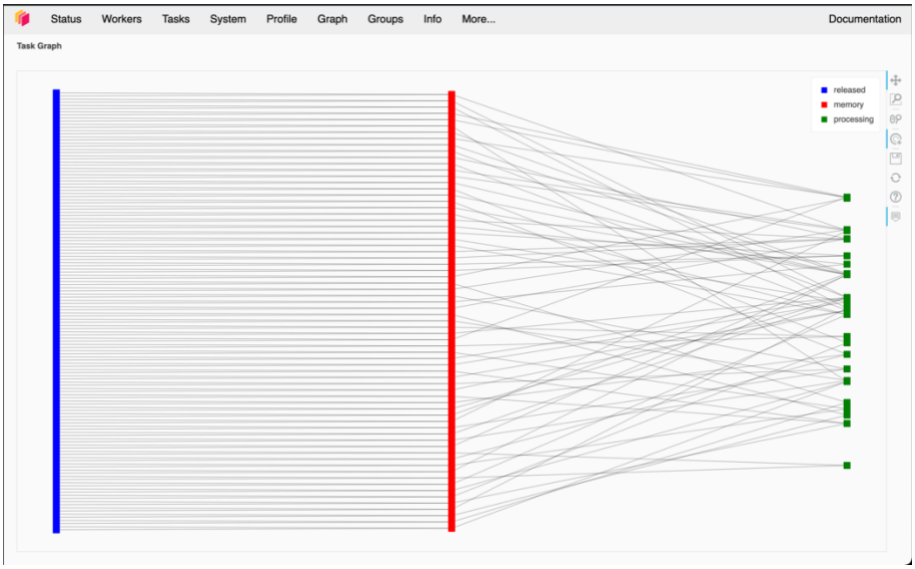
	Array	Chunk
Bytes	5.77 MiB	843.75 kiB
Shape	(7000, 108)	(1000, 108)
Dask graph	7 chunks in 1 graph layer	
Data type	float64 numpy.ndarray	



Below is a picture of dask dashboard visualizing the task stream with good chunk size.



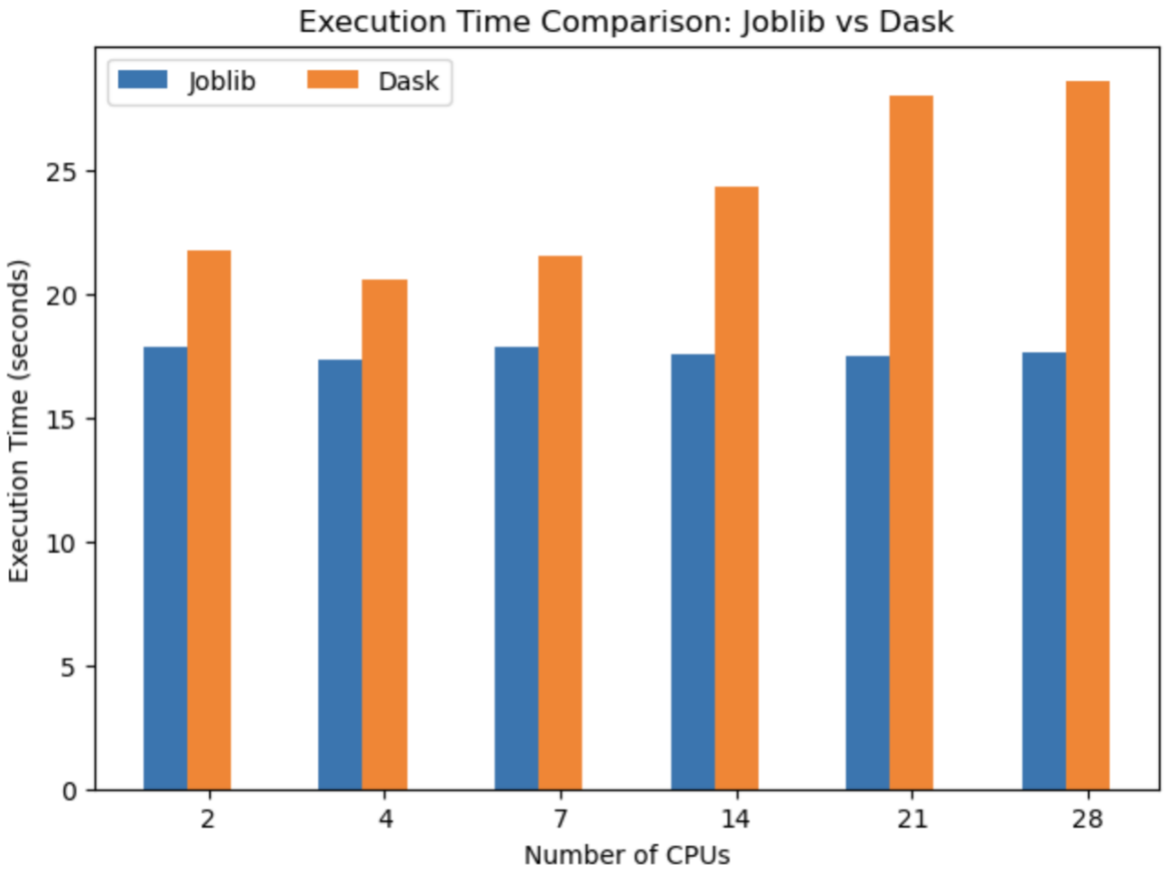
Below is a picture of the corresponding task graph from the dask dashboard.



c. Parallelization using joblib

- We made use of parallel_backend class from joblib module to parallelize the XGBClassifier because the results from earlier method were not good.
- The backend was set to “threading” and n_jobs parameter was variable.
- We experimented with n_jobs values as 2,4,7,14,28 and recorded the execution times.

- This time we observed the execution time at n_jobs was lower than serial execution time but very slightly.
- We have tried to parallelize the model with 2 different methods and still have not seen good results for parallelization.
- Analysis: Because the serial task not large enough, parallelization with multiple CPUs is not giving good speedup due to the overhead incurred.



4. Hyperparameter tuning:
   GridSearchCV is performed to tune the hyperparameters for the XGBClassifier model. We imported the GridSearchCV class from dask_ml module which uses Dask distributed client to search for best parameters in parallel. The n_workers was set according to the combination of parameters supplied to the parameter grid (4*3 = 12, n_workers = 12). Upon initializing the GridSearchCV class and giving it a grid of parameters, we got the best parameters which improved the accuracy of classifier model slightly. This part of the code was taken from [3].

Conclusion

In this project, we took a journey to explore the potential of parallelization techniques in improving the efficiency and scalability of music genre classification systems. Leveraging Python multiprocessing and Dask, we endeavored to streamline the various stages of the classification pipeline, from data preprocessing to model evaluation.

Through our experimentation, we observed significant reductions in processing time when employing parallelization compared to serial execution. Parallelization allowed us to distribute the computational workload across multiple cores or nodes, enabling faster feature extraction, model training, and evaluation. This enhanced efficiency holds immense promise for real-world applications, where processing large volumes of music data is a common challenge.

Furthermore, our analyses of classification models revealed promising results in terms of accuracy and performance.

There are several avenues for future research and development in this domain. Firstly, exploring more sophisticated feature extraction techniques, such as deep learning-based methods, could further enhance the discriminative power of the classification models. Additionally, investigating alternative parallelization frameworks and optimizing parallelization strategies could lead to even greater improvements in efficiency and scalability.

In conclusion, this project serves as a testament to the potential of parallelization techniques in advancing the field of music genre classification. By harnessing the power of parallel computing, we have made significant strides towards building more efficient and scalable classification systems, paving the way for exciting advancements in music information retrieval and beyond.

References

[1] Work w/ Audio Data: Visualize, Classify, Recommend [URL]
[2] Cornell Birdcall Identification [URL]
[3] Dask Documentation [URL]