CPEN 355 Final project

Tom Wang

Fall, 2023

This is the final project for CPEN 355. Specific code implementation should be found in the Jupyter notebook code file attached in the same zip file.

1 Introdution and Background

Compared to images or videos, songs are much easier to analyze using computers since they carry much less information. And it is probably one of the first massive uses of AIgenerated content: Sony released a song Daddy's Car in 2016 which is a song generated by training on The Bettles.

Motivated by a personal interest in music, I have chosen to undertake this machine-learning project. My primary aim is to leverage machine learning techniques to analyze and gain insights from some of the most listened-to songs. The multifaceted objectives of this project include:

1. Classifying songs based on various criteria such as singer, genre, and other relevant attributes. (Classifications learned in class)
2. Developing a predictive model to assess whether a given song is likely to be enjoyed or not based on certain features. (Maybe I can understand how the players recommend songs)
3. Further exploring and quantifying similarities between songs, providing a nuanced understanding of musical relationships. (Plagiarism in composition is a hot topic there)
4. Also, I can identify some signatures of the songs by comparing different parts of songs.

As for the data, I decided to run the learning on my local computer since music files are large and it takes time to upload. But I do not have a good PC, so I collected 16 most listened-to signers in my player, which is about 220 songs.

2 Preparing

Below is my implementation summary, which should be in the same order as my code. Detailed implementation explanation should be found in the actual code which is in the same file.

2.1 Preprocess data

2.1.1 Thought process

Although I chose the least quality data, the sample rate of the MP3 files is 22050. So 22050 samples per second which is a lot.

At the same time, human ears can hear from 20Hz to 20000Hz approximately. So to analyze a sound wave I need to apply about 20000 times of FFT per analysis.

However, if I only apply FFT for the whole wave, then it is just a measure of the total proportion of each frequency in the wave which does not give much information.

So I need to cut the audio file into very small chunks and analyze each chunk separately. If I choose the chunk to be 0.08s and songs are on average 4s long. I will get:

$$\frac{4}{0.08} \times 22050 = 1102500$$

data cells with only two attributes: time and frequency.

To save my PC, I decided to take three snippets of 10s from start, mid and end to analyze. At the same time, I just analyze the music node frequency from A_1 to C_7 which should be the notes used in most songs.

2.1.2 Processing algorithm

1. Read the song from my drive
2. Cut up 10s chunks from the start mid and end.
3. Apply FFT to each chunk for every 0.08s snippets
4. Collect the data and write them in separate $.csv$ files so that I do not need to preprocess all the time.

2.1.3 Data spliting

Just happened to be that there were a few songs that I downloaded from the player which is not properly labeled in the MP3 file. The following code would get 'None' from the file. So naturally, these would become my test data sets and wait for the rest to train the model.

```
def get_mp3_info(mp3_filename):
# Initialize Eyed3
audiofile = eyed3.load(mp3_filename)

# Get artist and genre from the MP3 file's metadata
artist = audiofile.tag.artist
#genre = audiofile.tag.genre.name
title = audiofile.tag.title

return artist, title
```

So I have 220 songs in total, 10 of them are test data.

2.1.4 Reduce the dimension

With the current size of the snippets, I have 125×64 data points for each snippet. So I need to reduce the dimension.

To Figure out the best way to reduce the dimension, I tried the following methods:

1. Grab a sample snippet
2. try PCA with different number of components

```
#Check the cumulative variance for each number of components
for k in range(5, 60, 5):
    pca = PCA(n_components=k)
    reduced_data = pca.fit_transform(df)
    cumulative_var = np.sum(pca.explained_variance_ratio_[:k + 1])
    print(k, cumulative_var)
```

3. Compare the cumulative variance with the number of components and choose the one that gets most of the variance the decent number of components.

Here is the result:

```
5  0.7254756251686398
10  0.8747829463301018
15  0.93963397065257
20  0.9707914244773223
25  0.9860049325538303
30  0.9927169174060696
35  0.9964934674252495
40  0.9986133518375307
45  0.999622615506004
50  0.9999261757883421
55  1.0000000000000002
```

Looking at the result, I decided to use 10 components. More than 85% of the variance is explained by 10 components, which is good enough for me. So now each snippet is reduced to 64×10 data points.

2.2 train the model

2.2.1 Choosing models

I decided to use SVM and NN to train the model. Both of them are good at the classification of high-dimensional data for this specific problem. I did not choose decision tree because it is not good at high dimensional data.

I used SVM out of sklearn. Then I used Tensorflow to build the NN. (I tried Pytorch but I can not install on my PC for some reason. But TensorFlow looks similar in the training process and syntax. So I just used Tensorflow.)

Parameters for each model are chosen by grid search.

- For SVM, it comes with a built-in grid search function. So I just need to specify the range of parameters and it will do the rest.

```
param_grid = {
    'C': [0.1, 1, 5, 10, 100],
    'kernel': ['linear', 'rbf', 'poly'],
    #'gamma': [0.1, 1, 10, 'auto']
}
```

  Note that I commented out the gamma parameter since it caused a fixation to predict everything to be the same since there is a dominant class in the data set. (But I do not think it was that bad like only 20/220 songs are from the same singer)

  All parameters are chosen by the grid search function dynamically. And the best parameters would be printed out after each training.

- For NN, it seems that there is no built-in grid search function, so I had to do it manually. But here are the parameters that I explored:

```
param_grid = {
    'units': [16, 32, 64, 128, 256],
    'epochs': [5, 10, 20, 30],
    'batch_size': [16, 32, 64]
}
```

  And I basically tried all the combinations of the parameters, using three for loops

```
for units in param_grid['units']:
for epochs in param_grid['epochs']:
for batch_size in param_grid['batch_size']:
    # Create a model
    model = tf.keras.models.Sequential([
        tf.keras.layers.Flatten(input_shape=(X.shape[1],)),
        tf.keras.layers.Dense(units, activation='relu'),
        # line below could limit the output to be 16
        tf.keras.layers.Dense(16,
        activation=tf.keras.activations.softmax, name='output')
        # Not sure how exactly it works, but it works.
    ])
    model.compile(optimizer='adam',
        loss=tf.keras.losses
```

```
        .SparseCategoricalCrossentropy(from_logits=True),
        metrics=['accuracy'])

    # Train the model
    model.fit(X, y_tensor, epochs=epochs, batch_size=batch_size)

            ......
```

Here are a few things I noticed when training with NN:

- It does not take String as the label, so I need to convert the label to Integer. (Maybe it does, I did not find a way to do it)
- Then its predictions would be out of bounds. I only have 16 labels but sometimes it predicts 49 or something wild.
- After some researching, I put the last line there in the model instantiation. It seems like it limits the output to be 16, which is the number of singers. Not sure how exactly it works, but it works.

3 Training and result

Training is pretty straightforward. I just need to call the train function for each model. And the best parameters would be printed out after each training. Here is the iterative procedure:

1. Read the data from the $.csv$ file (start, mid, end)
2. Train the model both with SVM and NN
3. Print out the best parameters with the accuracy on the testing data set

3.1 Result

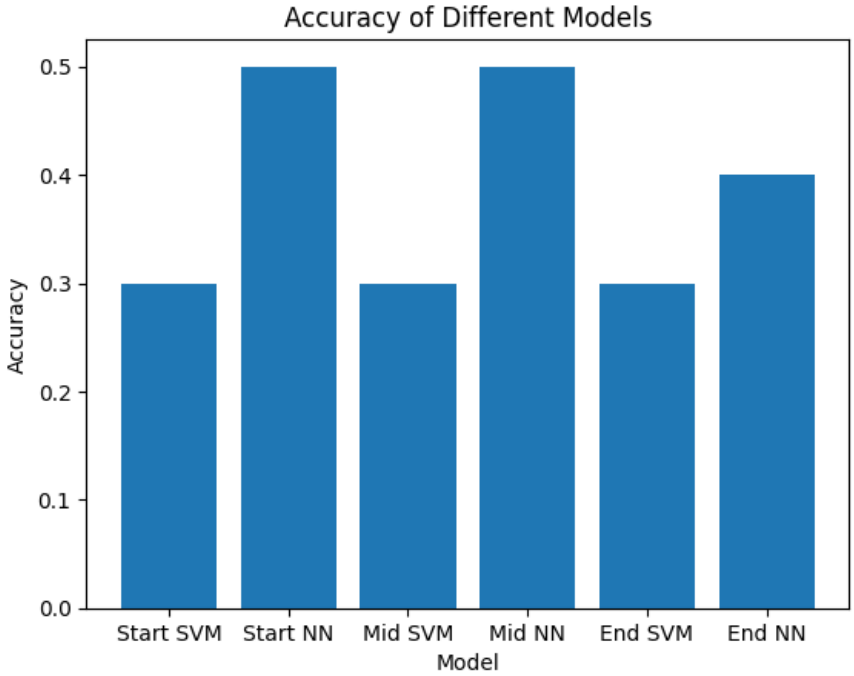
3.1.1 Fixation try with gamma in SVM

Before I decide not to use gamma, I got the following result:

Model	Parameters
Start SVM	{'C': 0.1, 'gamma': 0.1, 'kernel': 'rbf'}
Start NN	(16, 20, 16)
Mid SVM	{'C': 0.1, 'gamma': 0.1, 'kernel': 'rbf'}

Mid NN	(128, 5, 16)
End SVM	{'C': 0.1, 'gamma': 0.1, 'kernel': 'rbf'}
End NN	(16, 10, 16)

And here is the prediction results:



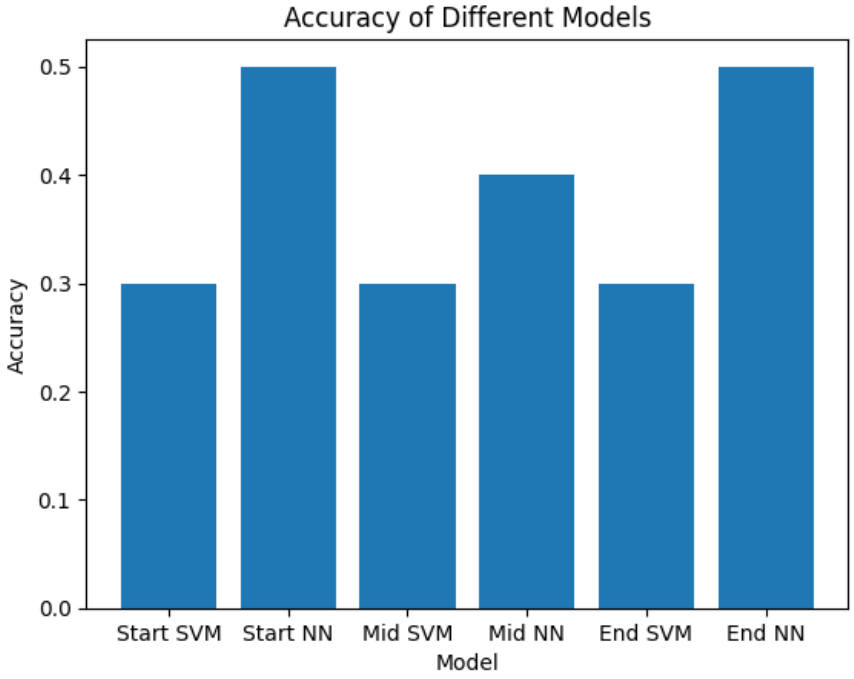
The SVM model is not very good at predicting the test data. It tends to predict everything to be the same. With this specific parameter setting, it would predict everything to be the same singer and it happens that the testing data set contains 3 songs from the same singer. So it predicts everything to be the same singer. And it performs very well on the training data set.

3.1.2 Try without gamma in SVM

Then I decided to remove that gamma and train again

Model	Parameters
Start SVM	{'C': 1, 'kernel': 'rbf'}
Start NN	(256, 10, 16)
Mid SVM	{'C': 1, 'kernel': 'rbf'}

Mid NN	(16, 5, 32)
End SVM	{'C': 1, 'kernel': 'rbf'}
End NN	(256, 5, 32)



Note that the performance on SVM did not change at all, but SVM actually predicts all kinds of singers. So it is not a fixation problem. It is just not good at predicting the test data set.

Also, the NN model performance changed a lot. But I did not do any optimization between the two trainings. So I think it is just a random thing.

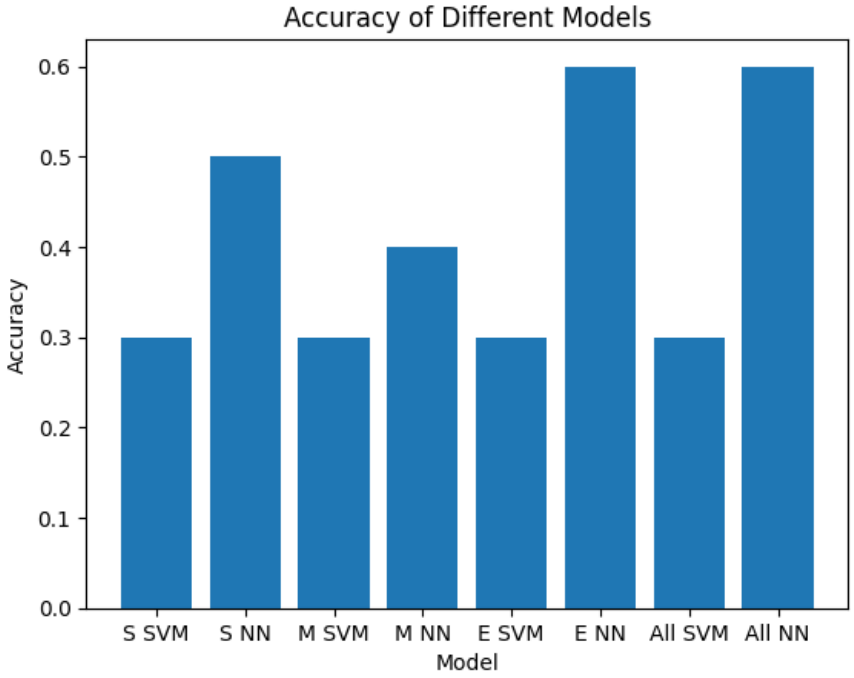
3.1.3 Try with all snippets

The performance did not match my expectations at all, so I suspect that the snippets are too short. So I decided to try with all the snippets. So for one time, I train with all start, mid and end snippets. I twisted a bit with my PCA algorithm to make it work with all snippets.

Now here is the reuslt:

Model	Parameters
Start SVM	{'C': 1, 'kernel': 'rbf'}
Start NN	(32, 30, 32)

Mid SVM	{'C': 1, 'kernel': 'rbf'}
Mid NN	(256, 30, 32)
End SVM	{'C': 1, 'kernel': 'rbf'}
End NN	(128, 20, 64)
All SVM	{'C': 1, 'kernel': 'rbf'}
All NN	(16, 30, 32)



I used abbreviations here so that they can fit in the same graph.

So SVM performance is still very constant. But NN is still very random. I rerun the all model training multiple times and the result was very random. So I think it is just a random thing.

4 Discussion and conclusion

- My primary guess of the training is that NN performs better than SVM given that it is 20 times the training time, which is the case.
- However, I was also expecting that the mid-snippets should have a better performance than the start and end snippets since it is the most representative part of the song (most likely

that the singer is actually singing in that part). However, the result did not match my expectations.

- Most importantly, I found that NN has a very unstable performance. I rerun the training multiple times and the result is very random. I think it is because of the random initialization of the weights. But I am not sure how to fix it.
- In contrast, SVM is very stable. It always predicts the same thing with the same model (the same parameters). But, it is not good at predicting the test data set.

4.1 Future improvement

- I think the most important thing is to find a way to make NN more stable. I think it is because of the random initialization of the weights. But I am not sure how to fix it.
- Also, I think I should try to use the whole song as the snippet. I think it would be more representative. But it would take a lot of time to train. Maybe when I get a better PC.
- Having a cross-validation would be nice.
- Testing data size could increase as well. This way, it might avoid the fixation problem in SVM so that fixation would not reach the best accuracy.
  - I think I can use the whole song as the testing data set. This way, it would be more representative.
  - Then I should probably try to use all the frequency ranges that could be heard by human ears. The FFT work I have done now only covers the music node frequency. But I think the whole frequency range would be more representative. Tone color and overtones are also important features of a song (Both to the singer and the instruments). But these frequencies do not live in the music node frequency range.
  - Number of songs could increase as well. SVM complains about the dominant class in the data set. Also, it complains that some classes have few samples so it can not split the data set into validation and training sets. So I think I should increase the number of songs.
  - All these probably would take a lot of time to train. So I should probably only try if I have a better PC (I increased from 64*10 to 64*100 and it took much longer to train. Now SVM took only 10 seconds, but after increasing, it took 9 minutes without completion, so I stopped it. SVM is bad already, NN takes 20 times longer so...)