Self Organizing Maps for Visualizing Million Song Dataset

Sarah Gerard, Bhupesh Shetty, Aidan McConnell
December 11, 2016

1 INTRODUCTION

For our semester project, we were assigned the task of visualization and classification of the million song dataset. To do this we were supposed to use self-organizing maps on GPU, to reduce the dimensionality of the dataset. In order to effectively implement SOM on this dataset, it is first important to understand the data and all the features which it contains.

1.1 MILLION SONG DATASET

The million song dataset, or MSD, contains 1,000,000 contemporary popular songs. The dataset is 280 GB of data, has 44,745 unique artists, 7,643 unique terms (Echo Nest tags), 2,321 unique musicbrainz tags, 43,943 artists with at least one term, 2,201,916 asymmetric similarity relationships, and 515,576 dated tracks starting from 1922. Each of the songs within the dataset contains 220 different features. However, some of the features for songs do not have any value associated with them. This is why there are only 515,576 tracks with a specific year associated with them, because many of the tracks did not have a value for their year. Each song within the dataset is stored using h5 files. So each file contains only 1 song from the dataset, therefore, there is a total of 1,000,000 h5 files. The dataset can be found at http://labrosa.ee.columbia.edu/millionsong/. The original features from the dataset are shown int the table at the end of this report.

A dataset with 220 features is very hard to visualize and process. Therefore, it is important to understand each of the features within the dataset. This is important because if there are certain variables that are less important than others, reducing the number of variables which need to be processed is possible. For instance, there are many features in the dataset which do

not have many values associated with them. These features are more likely to be unimportant to the classification of the data, than a feature which does not have many missing values. Also, to understand which features may be more important than others, it would be useful to consult a music expert, or research the individual features on a search engine. Finding out which features will have an effect on classification, will allow us to reduce the number of dimensions of the data, making computations easier.

1.1.1 PREPROCESSING

Dealing with a dataset of this size, requires a great deal of preprocessing. The first step in preprocessing this data, is getting the files in a format which can be used. The original dataset contained the songs in A-Z files, which were nested containing h5 files. Each individual h5 file represented just one song within the dataset, so there were a total of 1 million h5 files. So in order to get the data into a workable format we had to figure out how to convert the h5 files into a file format that we could use. So we extracted the features from each of the individual h5 files, and we put the data into CSV. Since we need to compute distances between features, we need to convert the data into a format which can be used in distance computations. The original data contained features which had 1 dimensional arrays and 2 dimensional arrays. For the 1 dimensional arrays we computed the average and norms. For the 2 dimensional arrays we took the covariance, average, and norms. For example, a 2 dimensional array with 12 columns were converted to 90 features (there are 78 unique values out of 144 entries in the covariance matrix and 12 means of each column). This step helped us extracting data from otherwise difficult 2-dim arrays. The paper (see [1]) was useful in inspiring this idea. Finally, once we got the data into a usable format, we converted the h5 files to CSV.

1.2 SELF ORGANIZING MAPS

The first process in building a Self-Organizing Map, is training. Training is an iterative process that builds the map using the input data. The first step of training is figuring out the BMU, or Best Matching Unit. The BMU is the neuron which is most similar to the input data point, it can also be referred to as the winning neuron. The winning neuron, is a widely used term because the training process of building an SOM is competitive. In order to compute the BMU, some sort of similarity metric needs to be used. From the research we conducted on SOMâĂŹs, we concluded that Euclidean distance, would be a good measure of similarity with this particular dataset.

In order to compute distances, the neurons within the neural network need to have the same dimensionality of the input data. Each input data point (song) has 220 different variables, therefore, the neurons also need to have a dimensionality of 220. So each neuron within the neural network has a randomly initialized weight vector associated with it, and this weight vector has a dimensionality of 220. Next, the BMU is computed for every data point within the dataset.

The winning neuron is computed for every data point in the million song dataset. This is done by finding the neuron with the smallest Euclidean distance from the data point.

$$j = argmin_j d(x_i, m_j)$$

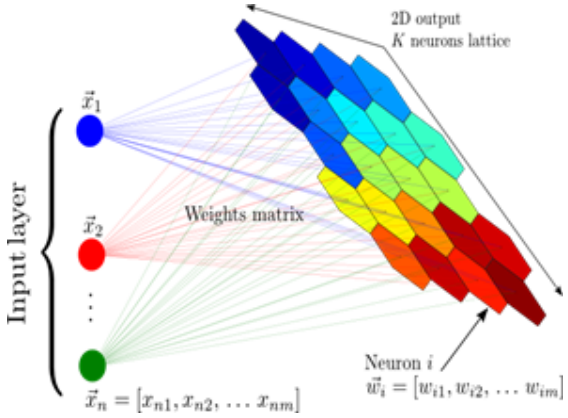


Figure 1.1: Shows a good visualization of the training of SOM. Each data point, denoted by x, is matched with the weight vector, w, of each neuron within the lattice.

When this process happens, and the BMU is found for the particular input data point, the weight vectors of the BMU and every neuron in the neighborhood are updated. This is done by using a neighborhood function, which decreases the size of the neighborhood around the BMU each iteration, until the neighborhood size is zero. The key concept of the Self-Organization of the neurons, is that every neurons weight vector is updated, not just the BMU. This is done through the use of a neighborhood function which decays exponentially with time, which reduces the size of the neighborhood around the BMU.

$$\sigma(t) = \sigma_0 exp(\frac{-t}{\lambda})$$

As the neighborhood decreases in size, the neurons within the neighborhood get pulled

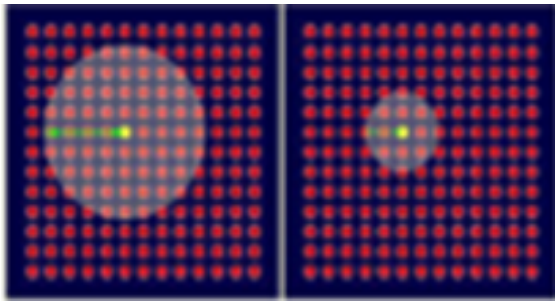


Figure 1.2: Shows the neighborhood around the BMU decreasing in size as the weights are being updated.

"towards" the input data point. This is shown in the neighborhood function above. This means the weight vectors within the neighborhood, get updated so that they become more similar to the input data point. Since the neighborhood is decreasing in size, the neurons closest to the BMU become more similar to the input data point than the neurons which are further away

$$v(t+1) = v(t) + \sigma(t)\alpha(t)(x(t) - v(t))$$

The equation above shows how the weights of the neurons within the neighborhood get updated. The weight update equation is dependent on a learning parameter, alpha. The learning parameter determines how fast or slow the weight vectors get updates. Alpha needs to be chosen wisely, because if it updates the weights too fast or too slow, the SOM will not work. Training the map in an efficient way will result in a good visualization of the SOM.

2 OUR APPROACH TO THE PROBLEM

We investigated the following approaches discussed in the class to solve big data problems.

1. Decrease the number of samples (N) or number of dimensions (d) of the Dataset.
2. Use smart algebra to speed up the computation.
3. Use super computing facilities available.

2.1 DECREASE N/D

We made a conscious decision not to decrease the number of samples and the number of dimensions. We wanted to utilize all the data that we have as this is a big data problem. We also did not want to reduce the number of dimensions as we wanted SOM to do that for us. It has been found in experiments, SOM does a good job in selecting the weights for each neurons according to the importance of features. More over, we wanted SOM to depict the higher dimensional data to lower dimensional one to make it easy for visualization. Having said the above, we faced some memory related issues with GPU and hence had to reduce features to run our experiments while using GPU for higher lattice dimensions.

2.2 SMART ALGEBRA

Somoclu (see [2]) is the SOM package that we decided to use for our project. We selected Somoclu because of the efficient implementation of smart algebra that speeds up SOM. For example, Somoclu use Gram matrix for distance calculation and this reduces the computation time for distances by 50%. Somoclu also utilizes the batch formulation of SOM that we will discuss in the next section. Batch formulation helps in parallelizing the code and hence exploit the super computing facilities available to us in a more efficient way. Somoclu was the only readily available open source software that had GPU supported code for SOM.

2.3 SUPER COMPUTING FACILITIES

We utilized the Neon Cluster High Performance Computing System at the University of Iowa with AL nodes assigned by us. We also used the Nvidia Kepler K20 Accelerator GPU cards. It features 2,496 cores and 5 GB of on-board memory.

3 SOM FOR MILLION SONG DATASET

We wanted to utilize the super computing facilities along with the GPU card for computation of SOM. We started by implementing our version of SOM. We soon found that there were better packages available online that had very efficient implementation of the code. This search led us to Somoclu package. Somoclu ([2]) provides parallelization for both CPU and GPU kernel. It is written in C++ and utilizes batch update for SOM.

3.1 THE BATCH SOM ALGORITHM

Batch SOM algorithm is an iterative algorithm, but there is a distinct difference in the way of updating weight vectors that was defined earlier. Instead of using a single data vector at time t, the whole dataset is presented to the map. Hence the name "Batch". The algorithm can be outlined as follows.

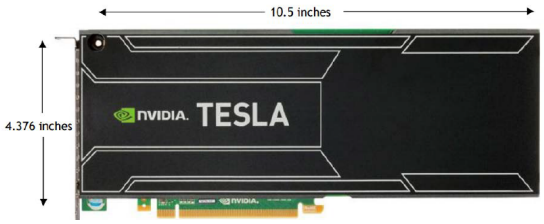
1. Selection of BMU: Consider a set of input samples X and a two-dimensional lattice where each node i a weight vector w_i is associated. The initial values of w_i can be selected at random (generally (0,1)). With the use of a distance measure, the node that is most similar to the input $x \in X$ is found and x is then copies into a sublist associated with that node.

2. Adaptation (Updating of the weight vectors): After all input samples $x \in X$ have been distributed among the nodes sublists, the reference vectors can be updated. It is updated using the following equation.
$$m_i = \frac{\sum_{x \in X} h_{ci}(t)x}{\sum_{x \in X} h_{ci}}$$
h_{ci} is the neighborhood function which generally is a gaussian function that can be defined as:
$$h_{ci}(t) = \alpha(t) e^{-\frac{\|r_c - r_i\|}{2\sigma^2(t)}}$$
The above expression enables us to exploit the parallel environment available to us. As seen from the above expression, we can distribute the load independently to each neuron and then finally aggregate all of them.

4 COMPUTATION EXPERIMENTS

We used C++ for running Somoclu ([2]) in parallel environment with CUDA acceleration for computing. We used Python for our data preprocessing and visualization tasks. Somoclu uses OpenMP for distributing the jobs in available cores in a node and MPI is used to distribute the load to all different nodes in the cluster. We utilized Neon High Performance Computing facility at the university of Iowa. Nvidia Kepler K20 Accelerator GPU card seen in Figure 4.1 was the GPU we used for our computation.

Figure 4.1: Kepler GPU Card



4.1 Parameter Estimation

The learning process involved in the computation of a feature map is stochastic in nature, which means that the accuracy of the map depends on the number of iterations of the SOM algorithm. We have seen that there are many parameters involved in defining and training a SOM, different values will have different effects on the final SOM. The values of the parameters are usually determined by a process of trial and error.

The size of the SOM determines the degree of generalization that will be produced by the SOM algorithm - the more nodes, the finer the representation of details, while the fewer nodes, the broader level of generalization. However, the same broad patterns are revealed at each level of generalization. Another issue regarding the size of the SOM, is that more nodes means longer training time. Even though there are a lot of parameters to experiment, we decided to concentrate on two important parameters - 1) Lattice dimension 2) Number of Epochs. Now the question arises how to choose which parameter works better. There are two error measures to calculate the quality of the Map - 1) Average quantization error. 2) Topographic error. Average quantization error is as a measure of how good the map can fit the input data and the best map is expected to yield the smallest average quantization error between the BMUs weight vectors and the input vectors x . The mean of $\|x_i - m_c\|$, defined via inputting the training data once again after learning, is used to calculate the error with the following formula:

$$E_q = \frac{1}{N}\sum_{i=1}^{N} \|x_i - m_c\|$$

Hence we select the parameter epoch that gave the least average quantization error. For number of epochs = 100, we got the least quantization error. Topographic error measures how well the topology is preserved by the map. Unlike the average quantization error, it considers the structure of the map. For each input vector, the distance of the BMU and the second BMU on the map is considered; if the nodes are not neighbors, then the topology is not preserved This error is computed with the following method:

$$E_t = \frac{1}{N}\sum_{k=1} Nu(x_k)$$

where N is the number of input vectors used to train the map and $u(x_k)$ is 1 if the first and second BMU of x_k are not direct neighbors of each other. Otherwise $u(x_k)$ is 0.

Ideally we should have considered both the above error measures to fine tune our parameters, however because of lack of time we consider just the quantization error. After holding

epochs = 100, we ran multiple experiments changing the lattice dimensions to 25 X 25, 30 X 30, 40 X 40, 50 X 50, 75 X 75 and 100 X 100. We then measured quantization error to pick the best map for our problem.

In the following section, we will discuss the different visualizations that we developed to present results from the Self Organizing Map of Million Songs Dataset.

5 RESULTS

5.1 CPU VS GPU

To demonstrate how parallelizing SOM helps in speeding up the computation, we ran experiments with 1) Naive or On-line SOM implementation. 2) Batch SOM without GPU but using MPI. 3) SOM with GPU. The results are as shown in Table 7.1. We ran the experiment with number of epochs =100 , 30 X 30 for lattice dimension and a gaussian neighborhood function. The learning factor changed from 0.1 to 0.01 in an exponentially decreasing manner. As seen from the results, the GPU card increased the speed of the computation by a huge margin. However as we increased the lattice dimension, we experienced memory issues and hence settled with parallel implementation without GPU support.

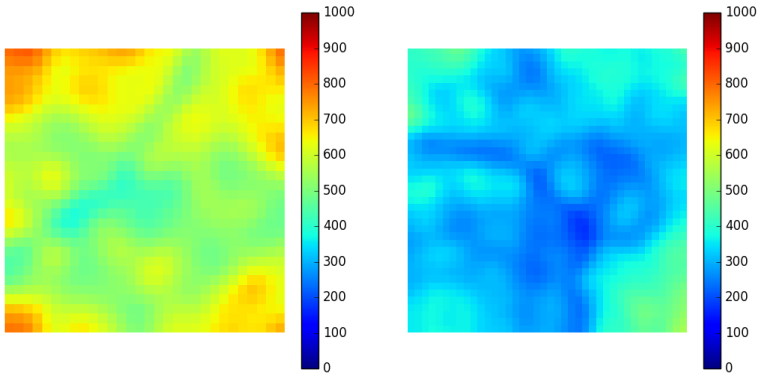
Methods	Runtime(Seconds)
Naive (on-line)	53,800
Parallel without GPU	1,056
Parallel with GPU	402

Table 5.1: Runtimes for different SOM methods

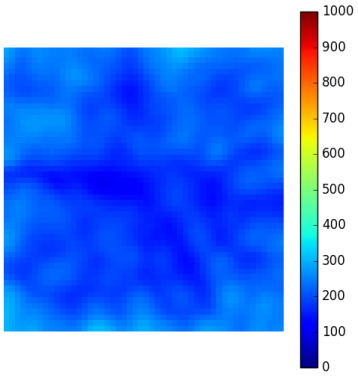
5.2 VISUALIZATION

SOM is a useful means for visualiztion of data. It allows us to take our dataset that exists in a very high dimensional space, and project it to a low dimensional space that is possible for humans to percieve and draw information from. There are several ways to visualize a SOM, for our analysis we chose three popular methods: hitmap, unified distance matrix (U-matrix), and component planes. We will discuss each of these in detail.

The hitmap displays the number of songs that are assigned to each neuron according to the best matching unit of the song. At the beginning of training, most of the songs are at a select few neurons, and there are many neuronrs that have little or no songs. As training goes on, the SOM starts to better fit the data and the distubution of songs over the map becomes more uniform. Figure 5.1 show the hitmaps for a SOM of sizes $30x30$. $40x40$, and $50x50$. For all map sizes, the songs are evenly distributed over the map for the most part, although as the map size increases there is less songs at any given neuron. There is clearly a tradeoff when choosing the best map size, smaller maps result in much lower training time however can be harder to analyze when there are so many songs at each neuron.



(a) $30x30$ (b) $40x40$



(c) $50x50$

Figure 5.1: Hitmap with different dimensions of SOMs.

The unified distance matrix, or U-matrix, displays the average distance of each neuron to each of it's neighbors. A U-matrix is a valuable too used to visualise the different clusters in a dataset. Regions of low value in the U-matrix are close together and can be considered a cluster. The high values in a U-matrix can be seen as ridges that separate different clusters. In Figure 5.2 we show a U-matrix for the $50x50$ SOM. The distinct red barrier separates the SOM into two distinct clusters, which turns out to be the binary mode variable which takes on a value or either 1 or 0. Additonally, the yellow curves further divide these two clusters into subclusters.

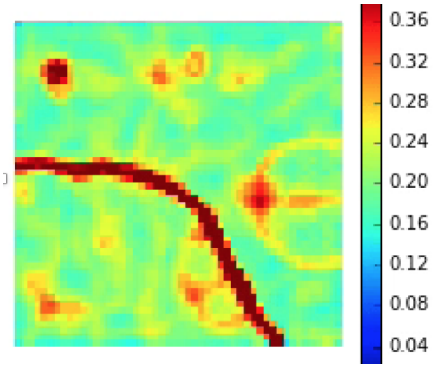


Figure 5.2: U-matrix for 50x50 SOM.

The component plane visualization provides a means to visualise the distribution of each individual feature separately. For any given feature we can see the value of that feature at every neuron of the map by taking a slice through all the feature vectors. This allows us to see different trends and correlations between the features. Figure 5.3 show some of the interesting features in our dataset. Additionally, we show the covariance matrix of the segmentats timbre feature in Figure 5.4. Many of these features are highly correlated and redundant.

Additionally, we plotted the map in the dataspace of three features to help us visualize the map unfolding during training. Figure 5.5 shows the map before the training occurs, where the map is very tangled up and not organized. Figure 5.6 shows the map at two angles after training. You can see the map does a nice job of becoming untangled and is nicely distributed over the dataspace. The z-axis of these plots is the mode features, which unlike the other features, takes on a binary value of 0 or 1. This is why the map organizes into two regions along the z-axis.

Lastly, we attempted to cluster the neurons of the trained SOM. To do this we projected all of the neurons into the original data space according to their final weight vector. In the data space, we performed a k-means clustering to group together the closest neurons. This clustering does not inforce any neighborhood constraint of the neuron neighborbood in the SOM definition space.After the clustering was performed, we projected the neurons along with their cluster label back to the SOM definition space. The resulting clusters are shown in Figure 5.7. From this we conclude that we have a faithful projection in terms of recal: points that were close together in the dataspace were projected to be close together in the SOM defintion space. Additionally, we chose some select songs to display at their BMU of the SOM

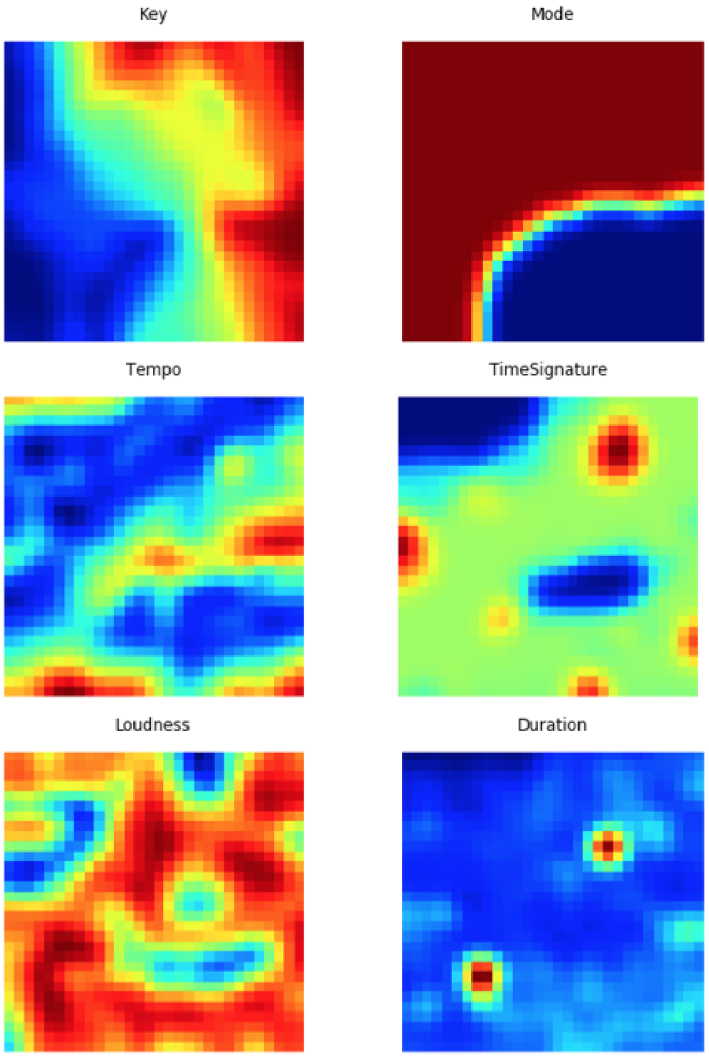


Figure 5.3: Component planes for several selected features.

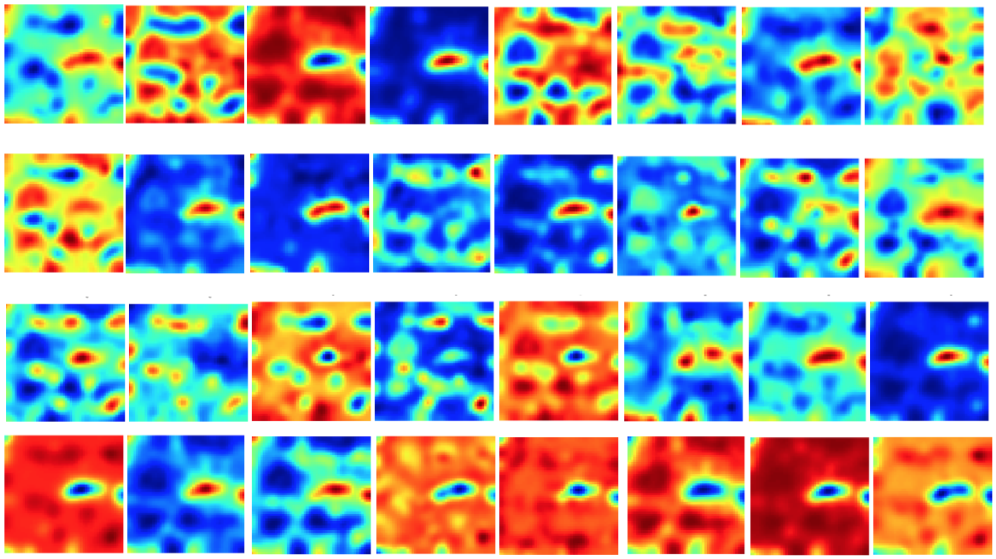


Figure 5.4: Component planes for features extracted from covariance of segments timbre feature.

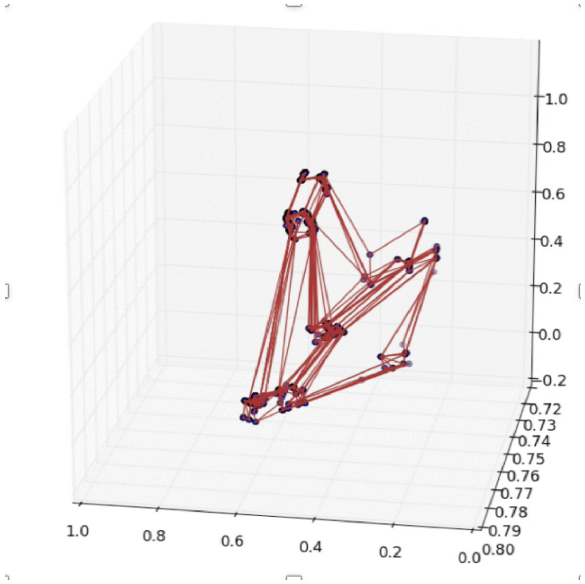
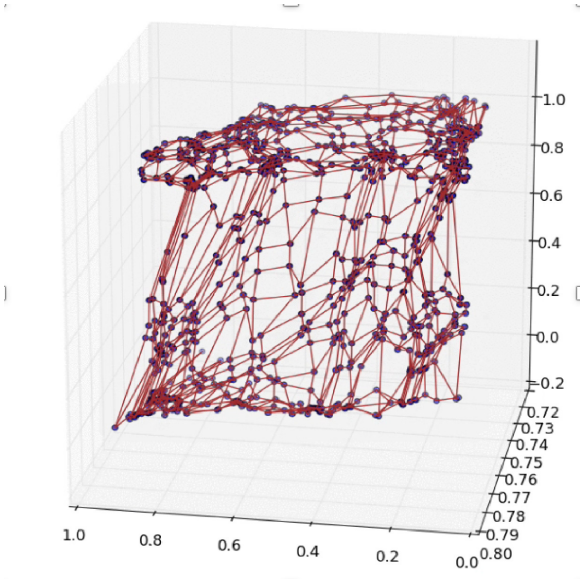
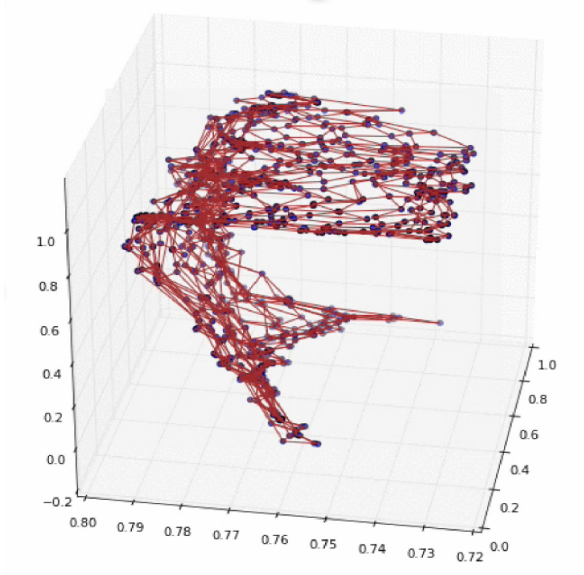


Figure 5.5: SOM map displayed in dataspace of 3 features before training.



(a) After Training



(b) After Trainging (rotated 90 degrees)

Figure 5.6: SOM map displayed in dataspace of 3 features (key, tempo, and mode) after training, two different angles are displayed to illustrate 3D structure.

map. In many instances the SOM did a nice job of clustering songs of similar genre together, for example the rap songs are in the same cluster and the country songs are in the same cluster. However, some classic rock songs are very close to the rap songs which we would not think of as being similar songs.

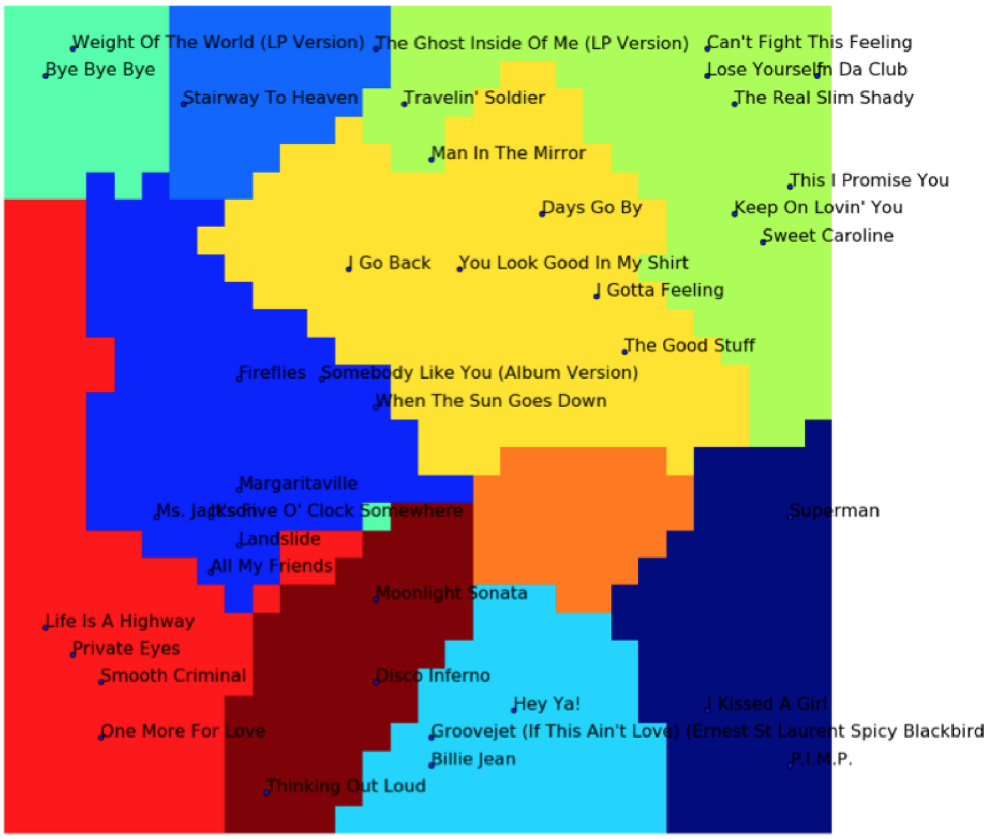


Figure 5.7: K-means clustering performed on neurons in data space. Some selected songs are also displayed on their associated BMU.

6 Limitations and Opportunities

One area that we could have improved our model is by carefully treating different features. A better understanding of the musical terms such as segments, timbres etc would have served us well while selecting the features. Parameter estimation for SOM is a complex topic since the combination of different parameters is extremely large. We feel there is a lot of opportunities to improve the parameters we used for mapping. For example, we used square dimensions for all lattices we visualized. It is worth experimenting with different parameters to see whether it improves our results. We selected some features of the dataset as labels to use it for visualization. One such feature was Year. One interesting experiment we could have

tried was to include Years feature in our dataset. This potentially could have given us some interesting clusters that could have led to some interesting interpretations. The Somoclu ([2]) package that we used to implement Self Organizing Maps had issues with GPU memory when we increased the lattice dimension. Some experience in programming in GPU platform could have helped us in solving the memory issues and use GPU for all our experiments. One of the biggest constraints in this project was time. The amount of time it took us to understand the dataset and pre-process it was a lot. We believe that if we had more time to work on this we could have made inroads to many of the limitations and opportunities discussed above.

7 Conclusion

In this semester project, we implemented both on-line and parallel versions of Self Organizing Map on Million Songs Dataset. We utilized the super computing facility in the university to implement the code on GPU card. We observed that the GPU card greatly enhanced the speed of the computation. We also developed visualization using SOM for the dataset and observed that our model performs well to classify different songs in a sensible manner.

In conclusion, we found Self Organizing Maps to be a very effective method to visualize and classify a huge dataset (both number of samples and dimensions) such as Million Song Dataset.

References

[1] Thierry Bertin-Mahieux, Daniel PW Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *ISMIR*, volume 2, page 10, 2011.

[2] Peter Wittek. Somoclu: An efficient distributed library for self-organizing maps. *arXiv. org*, 2013.

Field name	Type	Description
analysis sample rate	float	sample rate of the audio used
artist 7digitalid	int	ID from 7digital.com or -1
artist familiarity	float	algorithmic estimation
artist hotttnesss	float	algorithmic estimation
artist id	string	Echo Nest ID
artist latitude	float	latitude
artist location	string	location name
artist longitude	float	longitude
artist mbid	string	ID from musicbrainz.org
artist mbtags	array string	tags from musicbrainz.org
artist mbtags count	array int	tag counts for musicbrainz tags
artist name	string	artist name
artist playmeid	int	ID from playme.com, or -1
artist terms	array string	Echo Nest tags
artist terms freq	array float	Echo Nest tags freqs
artist terms weight	array float	Echo Nest tags weight
audio md5	string	audio hash code
bars confidence	array float	confidence measure
bars start	array float	beginning of bars, usually on a beat
beats confidence	array float	confidence measure
beats start	array float	result of beat tracking
danceability	float	algorithmic estimation
duration	float	in seconds
end of fade in	float	seconds at the beginning of the song
energy	float	energy from listener point of view
key	int	key the song is in
key confidence	float	confidence measure
loudness	float	overall loudness in dB
mode	int	major or minor
mode confidence	float	confidence measure
release	string	album name
release 7digitalid	int	ID from 7digital.com or -1
sections confidence	array float	confidence measure
sections start	array float	largest grouping in a song, e.g. verse
segments confidence	array float	confidence measure
segments loudness max	array float	max dB value
segments loudness max time	array float	time of max dB value, i.e. end of attack
segments loudness max start	array float	dB value at onset
segments pitches	2D array float	chroma feature, one value per note
segments start	array float	musical events, ~ note onsets
segments timbre	2D array float	texture features (MFCC+PCA-like)
similar artists	array string	Echo Nest artist IDs (sim. algo. unpublished)
song hotttnesss	float	algorithmic estimation
song id	string	Echo Nest song ID
start of fade out	float	time in sec
tatums confidence	array float	confidence measure
tatums start	array float	smallest rythmic element
tempo	float	estimated tempo in BPM
time signature	int	estimate of number of beats per bar, e.g. 4
time signature confidence	float	confidence measure
title	string	song title
track id	string	Echo Nest track ID
track 7digitalid	int	ID from 7digital.com or -1
year	int	song release year from MusicBrainz or 0