Project Name: Hit Song Prediction with Neural Networks

Group Members:

Oliver Mensah

Ariel Woode

Setriakor Addom

Phyllis Sitati

Bridgett Marufu

Introduction:

This project was inspired by a tweet by an Ashesi lecturer who said:

"*Dear lazy radio DJ who only plays hits instead of (doing the work involved in) finding & breaking hits. One day, your bosses will realize that an algorithm can do what you do. Consider Spotify a warning, fix up & make yourself more relevant."* – **A.Graham**

Upon the idea that this tweet inspired, the question that arose was whether a machine learning algorithm could be built that could do the work necessary for finding and breaking new hit songs and if it could suggest a playlist of hit song that would indeed render the "lazy DJ", as Graham stated it, obsolete.

The question that it rose was, could we build the machine learning algorithm that would do the work involved in finding and breaking new hits? Could it then be used to suggest a playlist of hits to be played such that a "lazy DJ" as Graham put it would be obsolete? We believe it is possible for a machine learning algorithm to learn the features of a hit song allowing it to select new hit songs and suggest playlists of hit songs if given a large enough supervised training set.

To further explore this idea, we outlined the problems and learning objectives such an algorithm would have to solve.

Problem:

- Being able to predict if a given song will be liked by an audience. i.e. predict if it will be a hit song or not.

- Suggesting songs not a part of the set of popular songs that satisfy the conditions of the best fit model.

Learning objectives:

- Analyse the meta data and audio features of a given set of songs.

- Derive a model that accurately predicts a song's hit status.

Having defined a problem and learning objective the next step was to decide what algorithm to use. The initial algorithms that were considered where a Convolutional Neural Network or a Naïve Bayes. A CNN appeared to be a good choice because Audio features of musical tracks range over a large multi-layered set of attributes and CNN's are designed to learn hierarchical features over multilayer structures. CNNs also have properties such as translation, distortion, and local invariances can be useful to learn musical features when the target musical events that are relevant to tags can appear at any time or frequency range.(Choi, Fazekas, & Sandler, 2016a). However due to the nature of the data set we used, a CNN was not a viable option. This was because, CNN's require that the data's features be concurrent and, in an order, relevant to the data. This is a direct result of the way they work. By cutting the data into smaller batches and finding the optimal solution for each batch, it is able to reduce the data set's feature space after each convolutional layer by pooling these smaller subsets of data after each convolution. This means that the features would need to be temporally relevant to be used in a CNN. Given that would could not determine the temporal relevance of the data we had, we were advised to use a different algorithm to solve our problem. As a result, we attempted two algorithms through the course of this project. One was a Naïve Bayes algorithm that was implemented and run on a smaller dataset as a proof on concept run. The second solution that was implemented was Deep Neural Network with a regression algorithm which is what was used for the final model.

Datasets

There were two datasets that were used over the course of this project. The main dataset used was the Free Music Archive (FMA). Our specific subset used 224 temporal and 8 audio features from the Echonest/Spotify library with over 13,000 examples, giving the total dataset a shape of (13129, 234). There was a secondary data set used as a proof of concept which was the Spotify Top 100 Songs of 2017. This dataset is 100 songs long with 13 significant features (Tamer, Oklap & Bilogur, 2018). This gives is a shape of (100, 13). It has a binary class label for each song which makes classification easier. This dataset is easier to use than the FMA for these reasons and was more ideal as a way to test the idea. Even though this data set works brilliantly as a proof of concept and is easier to use and read than the FMA, it still performs worse than our initial dataset the FMA for the following reasons:

- The data set has too few examples to be truly useful. With only 100 examples, even if we are able to train a very good model, we can expect to see massive overfitting and terrible generalization making it impractical to use as a solution to our problem.

- The dataset has only 13 features. Although this makes it easy to use, read and understand, they aren't nearly as expressive as the features in the FMA which number in the hundreds.

- The dataset cannot be added to and testing new, unknown songs is nearly impossible since the documentation doesn’t show how to get the features from raw audio files. This is a major feature of the FMA since it gets its data from raw audio files and describes how to do this.

As a result, we stuck to our main data set.

Implementations and results

Data: the FMA Echonest/Spotify subset.

In the full implementation of our project, we used the Echonest/Spotify subset of the FMA. This subset was chosen for the following reasons:

- The data consists of over 13,000 examples. This is far smaller than the full FMA dataset (over 100,000 examples) therefore allowing it to be trained on a laptop with relative ease. It is also large enough to allow for good generalization and to prevent over fitting.

- This subset contains 233 audio and temporal features of songs. This large feature space is ideal for a neural network since its performance is greatly improved by large feature spaces. This is also good because the features are natural and not hand coded or mathematically generated to achieve model complexity hence reducing the likelihood of massive over fitting.

- This subset had its own rating or "goodness" score in the form of Song Hotness. This is a value between 0 and 0.5 calculated by the Echonest/Spotify API. This made it easier to train on than a hand coded variable that may yield poor results if done poorly. Seeing as how a songs hotness could ideally be retrieved from the API along with a songs features, it made it ideal to use.

Algorithm: Deep Neural Network Regression (tf.estimator.DNNRegressor)

How DNNRegressor Works

The DNNRegressor depends on the optimization function which can be Gradient Descent, Adam, Adam Gradient Descent to train a neural network by taking into consideration what kind of loss function it should implement to reduce the error rate. At its core, DNNRegressor uses rectified linear unit as activation function but it can be modified with the different activation functions.

The core of our approach used a Deep Neural Network Regression algorithm supplied by the tensor flow estimator API. This algorithm allows for regression analysis to be performed on data given an estimator, an architecture and a dataset, amongst other things.

Although this formed the heart of our approach, given that it was the main learning algorithm we used, our performance was greatly influenced by the other implementation descicoins we took, most of which were informed by the google machine learning crash course introduction to Neural networks. This gave us insight into methods and approaches that would be beneficial to our project.

Our major implementation decisions are as follows:

- Selecting a large epoch number (5000 in this case). We did this to ensure the data had enough iterations to learn a good regression model while keeping it within reasonable time (20 minutes on average) and acceptable error margins. In our testing, we found that epochs less than 5000 did not guarantee convergence even though multiple runs showed that the data began to converge around the period 8 mark. More epochs than this also showed that the error had plateaued and hence would not benefit our models learning.

- Selecting a learning rate of 0.0007. testing showed us that the best performance for this algorithm came from a learning between 0.0005 and 0.0009. as such, we settled on 0.0007 given that it is in the middle of this range and gave us good results.

- Selecting a loss function: Tensor flow's DNNRegressor uses the Mean Squared Error for as its loss function. This is good for calculating an optimal minimal solution. However, given that the MSE is the square of the data's error, we decided to calculate the RMSE to show the error in the order of the data.

- Selecting activation function: By default, DNNRegressor uses rectified linear units (ReLU) as the internal activation function to take in the weighted sum of all inputs form previous layer and then generate and pass an output to the next layer. The higher performance of the ReLU over sigmoid influence our decision to still use the default activation function. Our algorithm however does not have an activation function for the final layer. Given that the model learns to predict a single continuous value, the final layer outputs the raw predicted value which it uses to calculate the RSME.

- Selecting the optimizer. The DNNRegressor optimizes the loss function with any of the said functions. Our implementation made use of Gradient Descent optimizer after trying different functions that works well with our loss function.

- Shuffling the data at the start of every period: This decision came with both positives and negatives, the latter of which we will discuss in a later section. On the positive however, this proved beneficial to our training as it led to quicker convergence and more accurate validation and prediction scores. Similar to the reason for randomising the data at the beginning, doing this prevented the model from learning the pattern of the data as opposed to learning how the data maps to the target values. This forces the model to rely more on learned features, making it better at predictions.

Approach:

Our approach has two main stages. Data handling through the data_preprocessor function and training through the train_nn_regression_model.

In previous itterations of our project, data handling was the hardest part to handle. to address that, we built the function data_preprocessor which makes use of functions taken from the google machine learning crash course. This allows us to split and handle the data in the correct formats for processing.

Our implementation uses these functions to split each data set into its features and targets, and to input them into the DNNRegressor properly. The data sets are divided into 3 parts for training, validation and testing. Our unique function, data_cutter, splits a pandas data frame object created from our CSV data file into these three sets using proportions. Each data set is then split up into its features and targets and passed into the tensor flow graph for training.

The train_nn_regression function houses the DNNRegression and passes the data into the model using the input_fn function. This allows the model to change the state of the data after every period to improve accuracy and reduce computational and memory load. It does this

by taking the data in batches to be worked on so as not to overload the memory of the computer.

This function also runs the DNNRegressor in a loop for two main reasons:

- To get periodic performance updates to see how well the model is doing at every stage.
- To allow the model to shuffle the data as often as possible to allow for good genralisation.

On every run of the loop, the model predicts an output using the training and validation data sets. This allows us to see how well it is performing using the RMSE function as a test. This shows us the error in the order of the data which we plot on a graph to show the training and validation errors.

Results:

Our testing showed some interesting results. After multiple runs, the model showed an optimal RMSE of ~0.055 on the training set and a validation error of +/- 0.004 in our best-case scenarios.

However, our model displayed some peculiar behavior during our testing. Coming to a definitive optimal solution was hard because we realised that on different runs, our model predicted different optimal solutions even when it was given the same model parameters.

Model training finished.

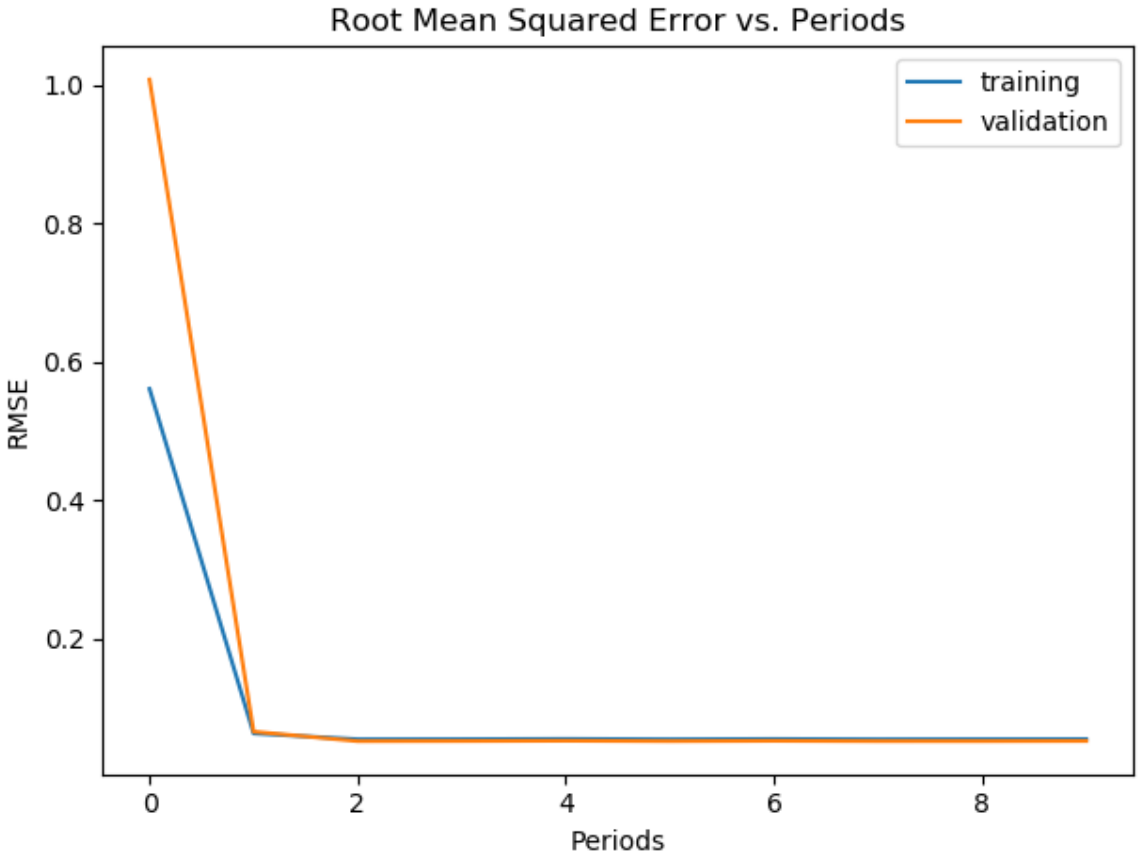
Model description: Steps: 5000 || Architecture: [10, 5] || Buffer size: 10503

Final RMSE (on training data): 0.0556

Final RMSE (on validation data): 0.0529

Duration: 0:34:36.718520

Final RMSE (on training data): 0.1725



Model training finished.

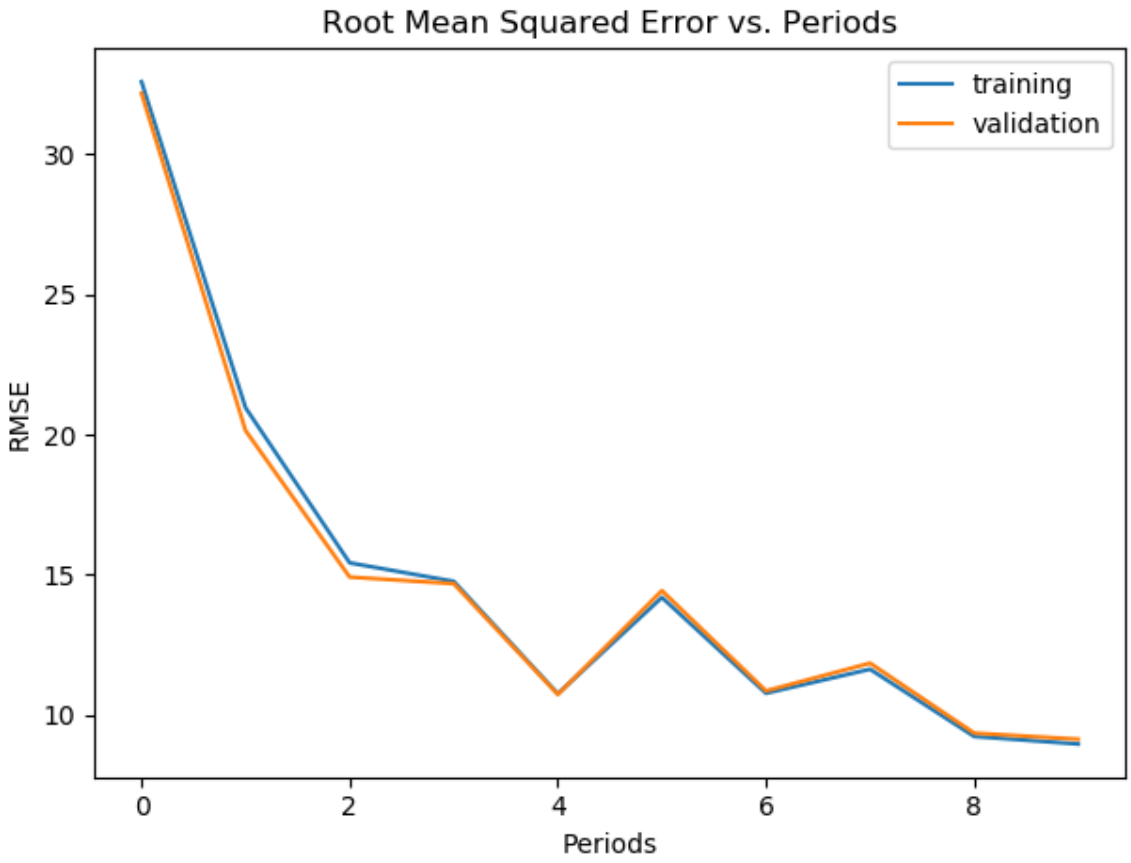
Model description: Steps: 5000 || Architecture: [10, 5] || Buffer size: 10503

Final RMSE (on training data): 8.9720

Final RMSE (on validation data): 9.1432

Duration: 0:34:47.286400

Final RMSE (on training data): 8.1660



Model training finished.

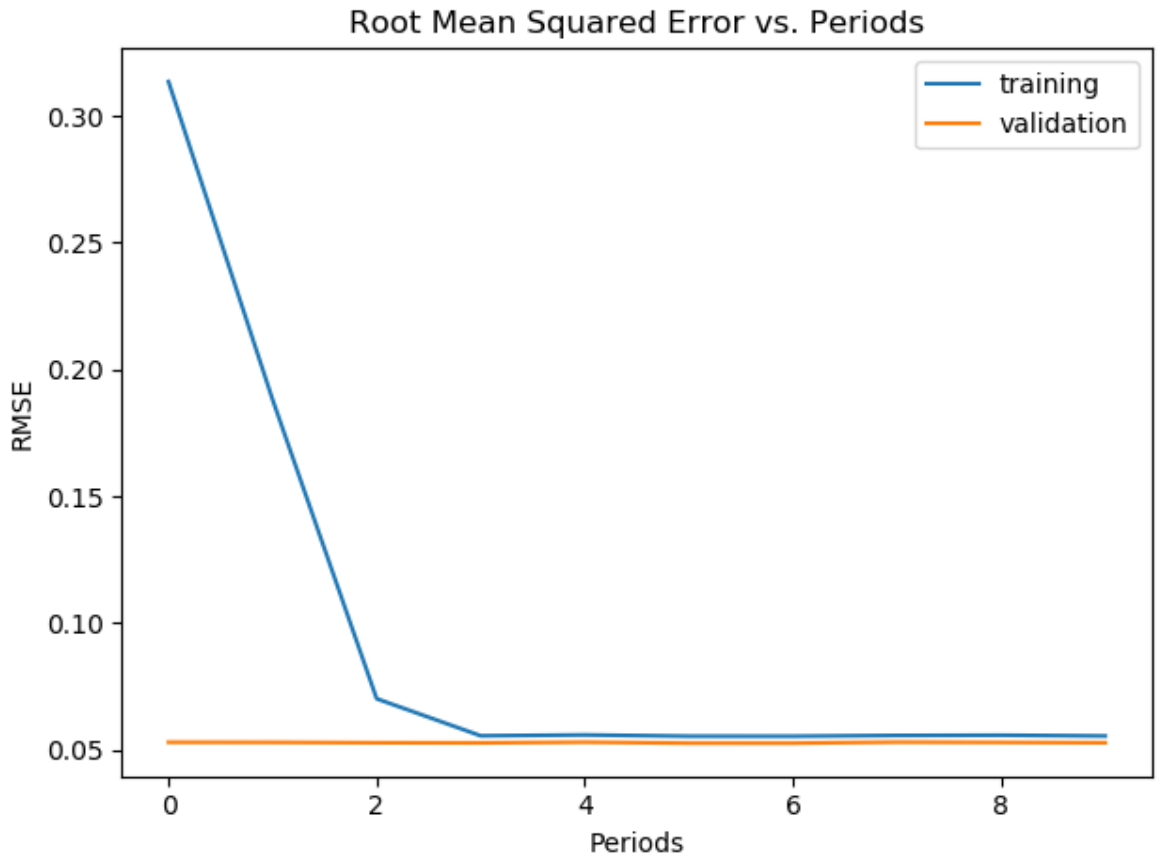
Model description: Steps: 5000 || Architecture: [10, 5] || Buffer size: 10503

Final RMSE (on training data): 0.0555

Final RMSE (on validation data): 0.0528

Duration: 0:22:52.043698

Final RMSE (on training data): 0.0850



We came to the conclusion that the data shuffling was responsible for this. When running without shuffled data, the model stayed more consistent even though it performed worse than the model with shuffle on.

Issues:

Our model varies too much between retrains to be considered reliable. As seen in the above example, the variations in model predictions are too great between runs for this model to trusted to predict good results consistently.

Another issue we had was with the architecture. From research, we learnt that neural networks generally perform better when given more neurons to work with. However, our testing seemed to show that the number of neurons or even layers seemed to affect the

performance very little, leaving us to settle on a 10X5 neural network model which seemed to have consistently decent results even despite it variations between runs.

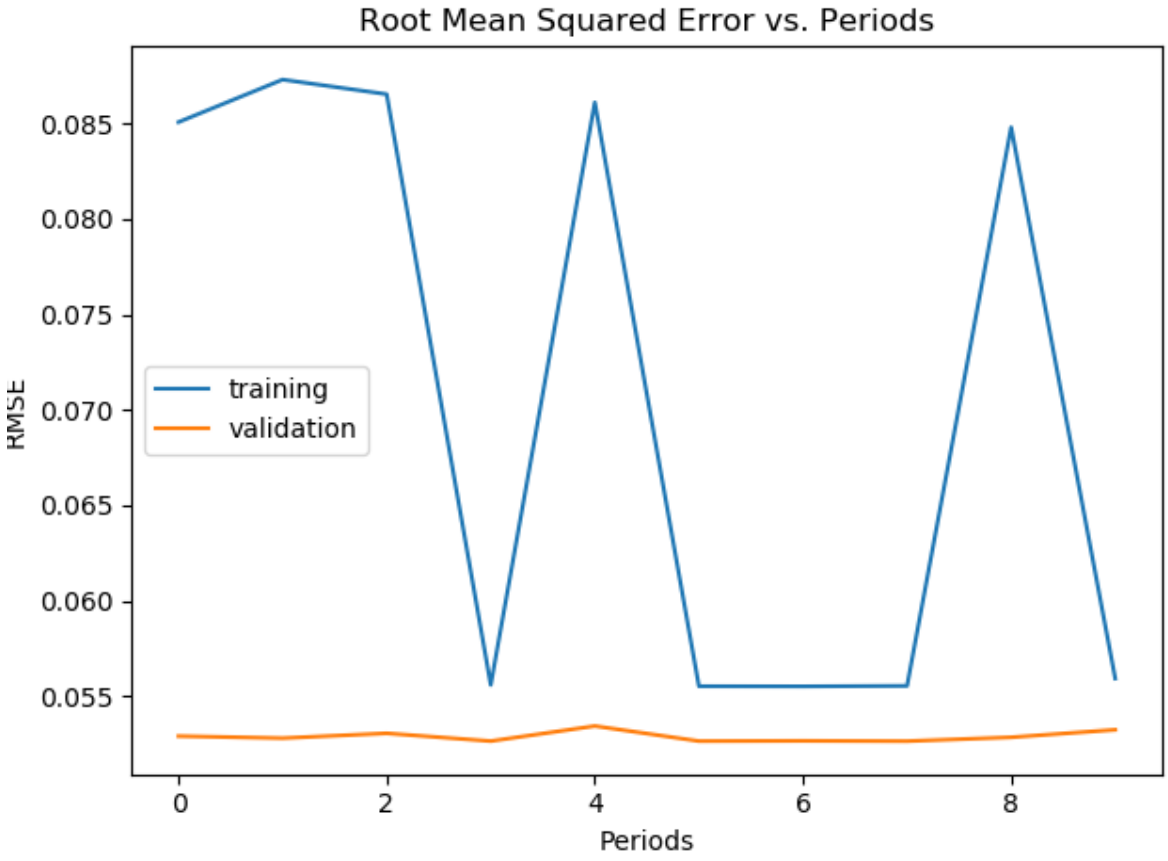
Model description: Steps: 5000 || Architecture: [1000, 500, 250, 100, 10, 5] || Buffer size: 10503

Final RMSE (on training data): 0.0559

Final RMSE (on validation data): 0.0533

Duration: 0:21:39.643164

Final RMSE (on test data): 0.0875



The biggest issue we had with our model however was with determining if the results we were getting were good or not. In its current form, we cannot seem to break the best case RSME score of ~0.055. On multiple occasions, our model seemed to plateau at this region leading us to believe that it is possible that we may have reached an optimal solution. This makes some intuitive sense given that our data targets are decimals between 0 and 0.5, some with up to 8 significant figures. As such, an error of +/- 0.05 for predictions seemed plausible. However, this is still a significant error within that range and so we cannot conclude that this is a good score even though it might look like it.

Given that the model is a regression model, we decided to try using the R^2 score to see how well it is performing. This yielded poor results. However, neural networks with multiple layers may learn a non-linear regression model in which case, the R^2 score wouldn't be able to tell us much about the models' performance.

Bibliography

1. Choi, K., Fazekas, G., & Sandler, M. (2016a). Automatic tagging using deep convolutional neural networks. ArXiv:1606.00298 [Cs]. Retrieved from http://arxiv.org/abs/1606.00298

2. Tamer, N., Oklap, C., & Bilogur, A. (2018). Top Spotify Tracks of 2017 | Kaggle. Kaggle.com. Retrieved 21 March 2018, from https://www.kaggle.com/nadintamer/top-tracks-of-2017

3. Google. (n.d.). Introduction to Neural Networks: Programming Exercise | Machine Learning Crash Course | Google Developers. Retrieved from https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/programming-exercise