# Proxying Preference: A Deep Feedforward Network for Song Categorisation

**Anna Katarina Page**
LT2326 HT21
Machine learning for statistical NLP: Advanced

## Abstract

For this project a deep feedforward network is built and trained on metadata features about tracks from two different users' Spotify likes with the aim of classifying unseen songs by probably playlist membership.

## 1 Introduction

In this project I use a deep feedforward neural network to categorise songs by playlist membership. The motivation behind this task is that if one can predict how well a particular song 'fits' in with a particular playlist, one could use this in order to determine whether one would be likely to like a song without having to listen to it.

Many pre-generated playlists, such as Spotify's discover weekly, are large and time consuming to listen through. Having a model that you could feed such a playlist into, and classify songs based on whether one is likely to add it to their own playlists would save time. That is, the question of whether or not a song is likely to 'belong' with some set of other songs (such as ones 'likes' or another playlist) can be formulated as a classification task.

As a classification task, the problem can be stated as: *Does song X belong on playlist Y?* For example, which songs present in my discover weekly playlist should I actually listen to on the basis that I might add them to my likes? Alternatively, which song on a new album would fit best with a given playlist?

However, such a task cannot be clearly evaluated without subjectivity. For that reason, in this project I have chosen to focus on a variant of this problem and aim to predict which playlist a given track came from. A successful model would essentially learn playlist characteristics, and could subsequently be applied to unseen music. Here I train a deep feedforward network on metadata available from the Spotify API on the songs in two people's Likes playlist, with the intention of predicting which of the two playlists of liked songs a given track came from. That is: which user would be most likely to 'like' a given track.

## 2 Previous efforts and literature

Plenty of work has been done - not least by Spotify - predicting e.g. genre membership or mood of songs on the basis of features. There are abundant examples of similar projects available on GitHub although they generally focus on genre prediction. The examples that I encountered also typically utilised pre-built models from e.g. Sklearn. Here I will construct a model rather than utilising one that is pre-built. Nevertheless, some of the the preprocessing in particular is similar and will has been a useful reference for this project.

(Sharma, 2020) outlines a mini machine learning project where metadata from the Spotify API about i) a list of songs the user likes, and ii) a list of songs that the user dislikes are used to train a prebuilt model to predict which set a newly encountered song comes from. As the models used are pretrained, the modelling sections and the evaluation were not adopted in this project. However, I did implement some techniques outlines there to visualise the data. Additionally, I adopted the same technique to get a baseline using a prebuilt regression model from Sklearn.

## 3 Data and Preprocessing

The Spotify API allows you to access a variety of audio features and metadata about song tracks in a JSON format. I extracted the track ids from two different user's 'Liked Songs' using an authentication token. These were then used to access the tracks' audio features. As the playlists were of drastically different lengths, the longer one was cut to be the same size as the shorter. The features extracted are danceability, energy, key, loudness, mode, speechiness, acousticness, instrumentalness, liveness, tempo, duration in milliseconds, and the time signature. These features, as well as the track id and a column denoting playlist membership were put into a dataframe which was used for the visualisation.

Danceability is a value between 0 and 1 calculated by Spotify on the basis of tempo, rhythm stability, beat strength, and overall regularity where the higher the score, the more danceable the track. The calculate energy is calculated on the basis of intensity and activity, again between 0 and 1. The key measure represents the key of the track. Loudness represents the average decibel value of the track. Mode refers to whether the track is in minor or major, represented as a binary value. Speechiness is a score between 0 and 1 representing the proportion of spoken words in a track. Acousticness is a value between 0 and 1 representing how likely it is that the track is acoustic. Tempo refers to the beats per minute that the track contains. Duration, represents the duration of the song in miliseconds. Finally, the time signature represents the number of beats per bar of each track and will be an integer between 3 and 7.

The 'playlist' column from the dataframe is used as the 'ground truth' in training and testing, while the remaining columns are used as the data for input to the training and testing. Each row of the dataframe was converted into numpy arrays of length twelve. A train, test split was created with one third being reserved for testing. Stratification was used to ensure equal proportions of the positive class and negative class in each split. At this point (following the method laid out in Sharma (2020)), the train tests splits were fed into a prebuilt regression model from Sklearn. The performance in this regression model sets the baseline to try to exceed.

Next, the data is normalized to be between 0 and 1 and is converted into an array. A custom Dataset class is created which is used to create the training and testing datasets. This converts the input and output data into vectors so that they can be easily given as input to the model. These are then fed into a Dataloader which batches and shuffles the datasets for use as input to the model for training and testing. The batch size is specified here. In this case I used a batch size of 50. The batch count is also calculated at this point which will be useful for calculating the average loss per epoch during the training stage later on.

## 4 The deep feedforward network

The feedforward network used here is fairly simple. It consists of three linear layers, each followed by an activation function. The first activation function used is ReLU, the latter two are the sigmoid function. The input size to the first linear layer is 12, the size of the input tensors and the output size of the first linear layer is set to 36. For the second linear layer, 36 is the size of the input and maintained as the size of the output of the second layer.

The model was also tested with other sizes as the output of this second linear layer, including 64 and 128, but in both cases the model's performance dropped significantly and nothing appeared to be learned during training with the larger output sizes. Finally, the last linear layer takes 36 as the input size and 1 as the output size, given that the task is a classification task. The model was also tested with additional layers (up to 5), but this did not improve the models performance and in fact worsened it, so these layers were removed.

A more complicated model was also trialed. This model consisted of more nuanced layers, including batch normalisation after each linear layer. This more complicated model was also tested with and without dropout (which was deemed to be unnecessary as the model did not appear to be at risk of overfitting the data). Ultimately, as with the trialled modications to the above model, none of this complexity improved performance, so this model was dropped from the project. Though the paramaters can still be loaded for comparison.

## 5 Training

The training loop for the model takes the following arguments: the model name, the device, the input size (i.e. the size of the input tensor), the output size, the name of the dataloader, the model itself (which has the relevent model, DffNetwork, as a default), the optimizer to be used, the loss function, the learning rate, and the number of epochs. The default optimizer is set to Adam, but the model was also tested with SGD, which did not improve perforance. The default loss function is set to BCEloss as the task is a binary classification task. MSEless was also tested, but did not improve the model's performance. The default learning rate is set to 0.001. Finally, the default number of epochs is set to 30, although in the actual training a significantly higher number is used.

Training runs on the GPU device specified. For each epoch, the average loss is calculated and printed, and, when training is completed, the parameters and history are saved so that they can be re-accessed easily and utilised in the evaluation of the model's performance.

## 6 Testing and evaluation

Four functions are defined for use in testing and evaluation. The first, `get_evaluation_variables` is used to access the following values: the predications made by the model for each item in the test batch, the true values of each item in the test batch, the average test loss, and the raw predications. The raw predictions are useful to be able to understand how or why the model makes the predictions that it does and to be able to fine-tune how those predictions are made. I will discuss this in more detail below.

The above function takes the saved path to the model's parameters as well as the GPU device used and runs the testing data from the dataloader forward through the model in evaluation mode and calculates the loss. To get a measure of the model's performance, in this function, the raw predictions output by the model (which all fall somewhere between 0 and 1) are converted into either a 0 or a 1 on the basis of a cutoff threshold. To do this, the helper function `get_predictions` is called on the raw predictions for each batch which converts them into either a 0 or 1. A list of the test predictions is then extended on the basis of the output of the helper function.

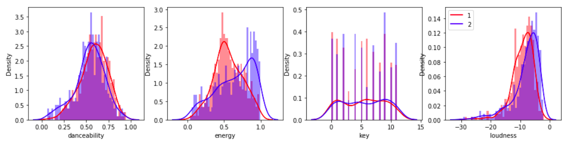
The threshold used is set manually at the time that the `get_evaluation_variables` is run. To determine the ideal threshold, for the first pass, the threshold is arbitrarily set to 0.5, but once the `get_evaluation_variables` has been run once and the raw predictions are accessible, these are then converted into a flattened list and the minimum and maximum values on that list are extracted. These are used to calculate the mean, which is then manually set as the threshold and the `get_evaluation_variables` function is rerun. I chose to use the mean as the threshold as for this task, there are an equal number of items in each of the positive and negative classes. A different threshold would be needed if the model was to be applied to other tasks where the classes are not the same size.

Next, a third function is called (`get_evaluation_metrics`) which takes the output of the previous function (the test predictions, the test truth and the average loss) as well as a device (in this case the CPU) and uses these to compute the accuracy, the precision, the recall and the F1 score. These are displayed in a Pandas Dataframe.

Finally, to visualise the model's learning across the epochs, a final function called `display_training_validation_loss` is fun on the history file saved during the training loop. This creates a matplotlib plot of how the loss changed across the epochs during the training of the model.

## 7 Results and Discussion

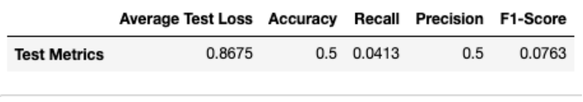
The performance of the model in predicting which playlist a song in the test data originated from after training is equivalent to chance. In developing the model, I had been comparing two users playlists who happened to listen to very similar music and, as such, had very similar lists of ‘liked tracks’. For the final run through however, I tested the model on two people’s playlists whose music tastes appeared to be very different from each other. The data visualisation in the preprocessing sections confirms that there are visualisable difference in the nature of the playlists.



As can be seen, the playlists differ visable in terms of daceability scores, energy scores and loudness scores. Other visualisations can be found in the Jupyter notebook, some of which also demonstrate clear differences between the features of the tracks in the respective playlists.

For this reason, I was expecting the model to be able to learn some of these differences. However, the model neither performed better than chance, nor outperformed the regression model from Sklearn.

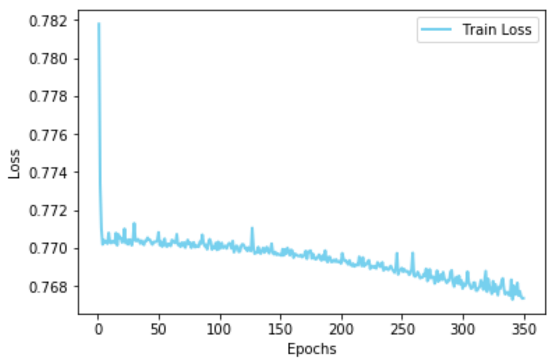
The evaluation metrics for the model run on the test data are as below:



| | Average Test Loss | Accuracy | Recall | Precision | F1-Score |
|---|---|---|---|---|---|
| **Test Metrics** | 0.8675 | 0.5 | 0.0413 | 0.5 | 0.0763 |

As can be seen, the overall accuraccy of the model is 0.5, meaning that the model performs exactly as well as it would if guessing randomly. The recall score (that is, the amount of items from the positive class which the model corectly categorises as being the positive class) is extremely low at 0.004.

The poor performance of the model is less surprising when we examine the loss across the epochs:



Although an initial drop can be seen, and although there appears to be a gradual downward trend in the loss, after the initial drop all of the loss values are within a very small range, only changing about 0.03 of an integer between the second epoch and the 350th epoch. In tuning the , I tried training the model for 100, 200 and 500 epochs. With the first two, the trend remained downwards. When the model was trained for 500 epochs however, the downwards trend stagnated and the loss both got worse, and became much more volatile after a certain point. For this reason, I settled on using 350 epochs, which is just before the model’s performance gets worse.

I also experimented with adjusting the learning rate. Neither higher learning rates nor lower ones improved the performance, and in fact were both detrimental.

## 8 Further attempts to improve performance

Aside from adjusting the number of epochs and the learning rate, I also tried implementing other, more significant modifications to the model. These were also discussed above. Various different model architectures were tested including utilising more linear layers, using different activation functions, using batch normalisation, and using MSE loss instead of BCE loss, and using the SGD optimizer over the Adam optimizer. None of these alternations improved the perforamce of the model, each independanly (i.e. when I changed only one of these at a time) tended to bring the overall performance down from 50% accuracy (equivilant to chance) to approximately 40% accuracy. That is, they lead the model to perform at even below chance. For this reason, I did not include these adaptations in the final version or run-throughs of the model.

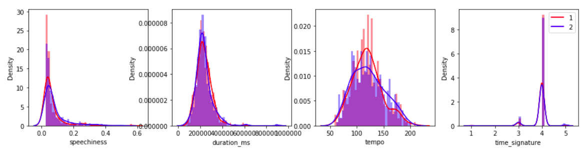
Also as discussed above in an earlier section, I attempted an entirely different model architecture, which used linear layers, batch normalisation and dropout. It also did not improve performance in any way and this model was ultimately scrapped, though the parameters can be loaded for comparison if desired.

## 9 Speculations

I am not sure how to account for the poor performance of this model and for why no alterations to the model that I made were able to improve the performance above chance. It may be that although there are certain visualise differences between the features of the tracks on the respective playlists, that overall the tracks are simply not dissimilar enough from each other for the model to be able to usefully learn to categorise new songs on the basis of the training. It may also be that with the subtle differences, the model would need more data for training than was given as input to the model in this case.

## 10 Future directions

In order to come to a clearer understanding of the model's poor performance, it would be interesting to try changing some elements of the way that the data itself was processed. One example that I believe would be particularly interesting would be to drop features from the input tensors where the values for that feature are too similar across the playlists, and generally, to try including and excluding various features from the input data. The image below shows visualisations of two features that may be interesting to experiment with excluding.



The first example of a feature to drop may be the time signature, almost every track in both playlists has a time signature of 4, so it is unlikely that the model learns anything of use from this feature and it may be just 'muddying the waters'. Another example of a feature where dropping it may improve performance is the length of the tracks. In including it originally, I speculated that some users may be drawn to or avoid particularly long or short songs, but this did not appear to be the case. As this number was represented in miliseconds, it was also very large, which forces the normalisations of the data into more compact ranges, though this should not really be overly problematic. The lack of variation across the playlists can be clearly seen in the image above for both of the two features discussed.

## References

Palash Sharma. 2020. [mini ml project] predicting song likeness from spotify playlist. https://machinelearningknowledge.ai/mini-ml-project-predicting-song-\likeness-from-spotify-playlist/.