MMD: Programming Assignment #1

Group 1: E. Guliev, J. Rass, C. Wiskott

University of Vienna — April 16, 2021

Abstract

All tasks were completed and instructions on executing the program can be found in the attached `README.txt`. Furthermore they are uploaded to the GitHub repository: `https://github.com/ezorrio/genre-classification`. The files require no input parameters and can simply be run with a standard IDE or from the command-line. The print statements produce the relevant results.

Data is stored on Github using LFS `https://git-lfs.github.com`. If you have all needed data - FMA dataset, you can clone the repository without downloading them by using

```
GIT_LFS_SKIP_SMUDGE=1 git clone
https://github.com/ezorrio/genre-classification
```

The best achieved results on the validation set are around 45% while the accuracy on the test set is around 33%.

Local Sensitive Hashing (LSH) for Item Search

The goal of this assignment was to implement an efficient nearest neighbor algorithm for genre classification of previously unclassified music tracks. For this purpose we used LSH as a framework for the similar item search while incorporating the random projection method. The sought-for result was a classification model that, given the optimal hyper-parameters obtained trough careful testing, would return a reasonable classification score for the provided test set and thus would be able to autonomously classify new music tracks.

Implementation

Subtask 1: Data Loading and Data Preparation

FMA-Class

```
class FMA:
    def __init__(self, path, subset='small',
                 feature_fields=None)
```

The data used for this experiment stems from the Free Music Archive (FMA)[1] and uses the `fma_metadata.zip` files. Out of those files `tracks.csv` and `features.csv` are considered for the training and verification process. For both data sets the `'small'` subset is considered which contains 8000 samples over 8 different genres with varying representation over the whole set. To ensure a good split of the data for training purposes, the pre defined splits `'training'`

[1] `https://github.com/mdeff/fma`

(6400 samples), 'validation' (800 samples), 'test (800 samples)' are used, which are already set up in a stratified way where each genre is represented with the same amount of samples. The data is loaded into the FMA class and processed according to the parameters given in the main program. For training a combination of feature sets of the features.csv file various combinations of the available feature sets are considered, but after a lengthy testing period the 'mfcc' feature set proved to deliver the best results.

Subtask 2: RandomHash

The random hashes are generated by using randomly generated projection matrices with the condition

$$r_{i,j} = \sqrt{3} \cdot \begin{cases} +1 & with\, probability\, \frac{1}{6} \\ 0 & with\, probability\, \frac{2}{3} \\ -1 & with\, probability\, \frac{1}{6} \end{cases}$$

using the RandomHash class.

RandomHash-Class

```
class RandomHash:
    def __init__(self, data_size, hash_length)
```

The class itself takes the number of features and the length of the hash value as inputs and generates the random projection matrix using the above displayed condition. The number of generated projection matrices is decided by the user using the LSH class, where the incoming samples get multiplied with the random projection matrices to generate hash values for the buckets. The approach of using random projections is used as a simple method of converting data in high dimensional space into a lower dimensional one. The specific method in this case is used, because it produces a uniform distribution of vectors into space with mean 0 and variance 1, which is desirable for approximated nearest neighbour algorithms, while also keeping the computational cost very low compared to generating these vectors with a Gaussian distribution methods. Furthermore this class is used to calculate the hash values, add them to their respective bucket, and retrieve them for given input data.

Subtask 3: LSH

LSH-Class

```
class LSH:
    def __init__(self, data_size, hashes_count, hash_length)
```

The *LSH* class functions as the container for the different hash tables i.e. the objects of the class *RandomHash*, each containing hash values and their corresponding track ids. The argument *hashes_count* regulates the amount of hash tables and the *hash_length* the length of the hash keys in every hash table.

The *hash_data* function takes a data set, iterates through all feature vectors and passes the vectors and their corresponding track ids to all hash tables contained in *self.hashes* i.e. builds the hash tables from the given data, which is done during the training phase of the algorithm.

The *get* function takes a feature vector and is used to obtain all track ids from each hash table that share the same hash key as the hashes feature vector.

Subtask 4: MusicSearch

Using the implementation of the blog-post as a template, we created the class *MusicSearch* in order to carry out the approximated k-nearest-neighbor (*knn*) search. It takes the parameters n (number of hash tables), l (hash length), the FMA data-subset, the subset of the features, the similarity measure, k (number of neighbors to consider in knn) as well as the metric "*magic number*", which regulates the size of the random subset of similar tracks in the course of the genre prediction. The class features a set of standard train functions that utilize the *LSH.hash_data* method to build the hash tables from the training data as well as test-functions that are used during the evaluation phase.

MusicSearch-Class

```
class MusicSearch:
    def __init__(self, data_path, n, l, subset='small',
    feature_fields=None, measure='Cosine', k=5,
    magic_number=800)
```

The rest of the functions revolve around the knn-problem, starting with computing the similar tracks via the *find_similar_tracks*-function, which takes a feature vector as an argument, hashes it to every hash table and returns the track ids with the same hash key. Because, in some cases, the amount of similar tracks are in the thousands, the problem was approximated by only taking a subset of similar tracks (see the *k_neighbors*-function). The size of this subset was jokingly called "*magic_number*", due to the enormous effect it has on the run-time as well as the classification score. With the smaller subset, the run-time of the genre classification for the "small" FMA data-set, was reduced by a factor of up to 5, while reducing the accuracy only by around 2-4%, thus constituting a sensible trade-off for this problem.

After selecting the subset, the pairwise similarities of the feature vector and the feature vectors of the similar tracks are calculated, which are then sorted according to the chosen similarity measure. Only the track ids of the k-most similar tracks are returned, which are then used to compute the most similar genre. This genre is finally taken as the predicted genre of the given feature vector. In the case, that less than k similar tracks are found for a given feature vector, the algorithm calculates the most common genre of however many similar tracks were found.

The genre classification for the entire test set culminates in the *print_classification_results*-function which prints the classification score for each genre as well as the overall i.e. average classification score over all genres.

Training

Parameters and their definition

In order to speed-up the calculation, instead of comparing given element with every single item in hashtables, we pick up maximum this amount of items in a random way

Automatic execution

In order to understand the behavior and the way each parameters affect the accuracy of the model, we have written a small script to evaluate different combinations of above-mentioned parameters.

Parameter	Description
number_of_hashtables	Defines the amount of hashtables within LSH model
hash_length	Size of hash
subset	Data subset to work on. Related to FMA. Fixed to small.
feature_fields	A list of features we want to count during model training
measure	Used to calculate similarity between features. Can be Euclidian or Cosine
k	Parameter for KNN search
magic_number	In order to speed-up the calculation, instead of comparing given element with every single item in hashtables, we pick up maximum this amount of items in a random way

Table 1: Parameters description

1. While inspecting execution logs, it was quite clear that among of all feature combinations tested, best value for *feature_field* = "mfcc".

2. Furthermore, according to those logs it was also clear that with increasing k we overall get better models

K	Amount of models with accuracy >= 40%
3	1
5	4
7	8

Table 2: Correlation of k with quality

3. In most cases increasing *magic_number* improves accuracy

4. We have 13 models with accuracy of more than 40%. 6 of them use Cosine as distance measure, 7 - Euclidean.

5. Best genre accuracy gained among those models - Folk: 70%. Worst - Experimental: 9%

6. Best overall accuracy among all the models - 41%.

Manual tuning

Experiments from previous section helped us to get a little bit of insight of how each parameter affects our model. Based on that knowledge, we can fix following parameters:

Parameter	Value
feature_fields	["mfcc"]
measure	Cosine
magic_number	As much as possible (keep in mind runtime)

Table 3: Fixed parameters and their values

Therefore, *k*, *number_of_hashtables* and *hash_length* parameters left and are subject of experiments further.

As a starting point, we took best model (accuracy-wise) from automatic executions. Magic number is set to 1500 Below steps taken for parameters optimization are provided:

Step №	k	number_of_hashtables	hash_length	Accuracy	Notes
1.	7	20	16	42.5%	
2.	10	20	16	43.125%	
3.	15	20	20	44%	
4.	15	25	20	42.375%	Failed. Restore to 3.
5.	15	20	25	45.125%	
6.	20	20	25	46%	
7.	20	25	25	45.625%	Failed. Restore to 6.

Table 4: Manual optimization steps

So far best result was gained within experiment 6. Let us use that for evaluation of the model with *magic_number*=8000. Instead of picking subset of items to compare with it will use the all the items available, thus using KNN instead of aKNN.

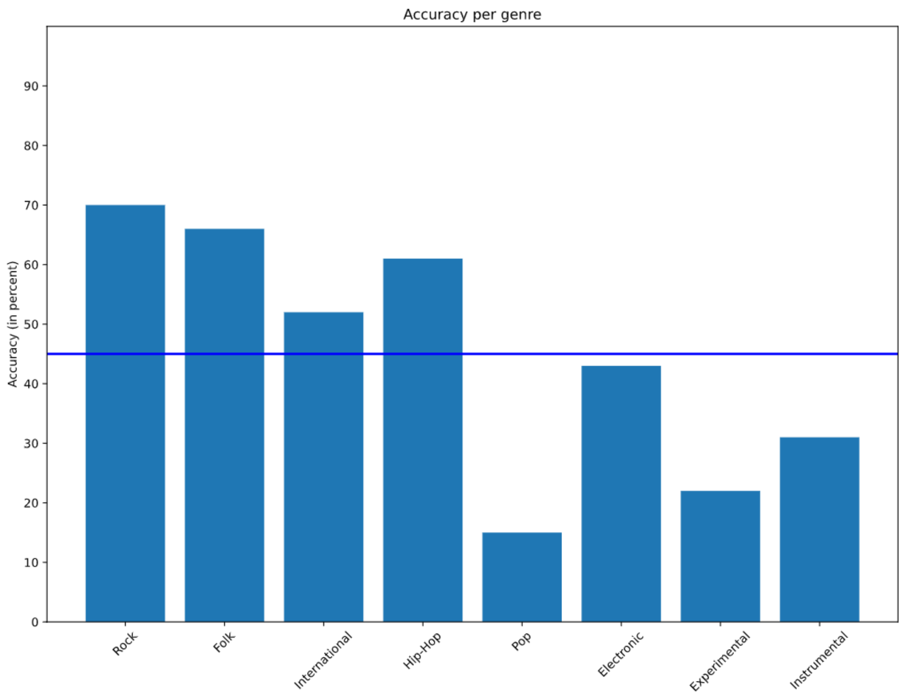

Figure 1: Training vs Validation Data

Definitely there is possibility that we can pick up better hyperparameters, but result of 45% is pretty acceptable: as we have 8 genres, trainingvalidationtest data in uniformly distributed among genres, the probability of randomly predicting the correct genre is $\frac{1}{8}$=12.5% which is much less than 45%, so model definitely works as needed.

Additionally it is worth to mention that all evaluations were done on validation set, such that we don't know yet how will it work of the test data.

Evaluating test data

For that purpose we will use the best model gained from the previous section, combine train the model on training/training and validation sets combined, and evaluate against test set.

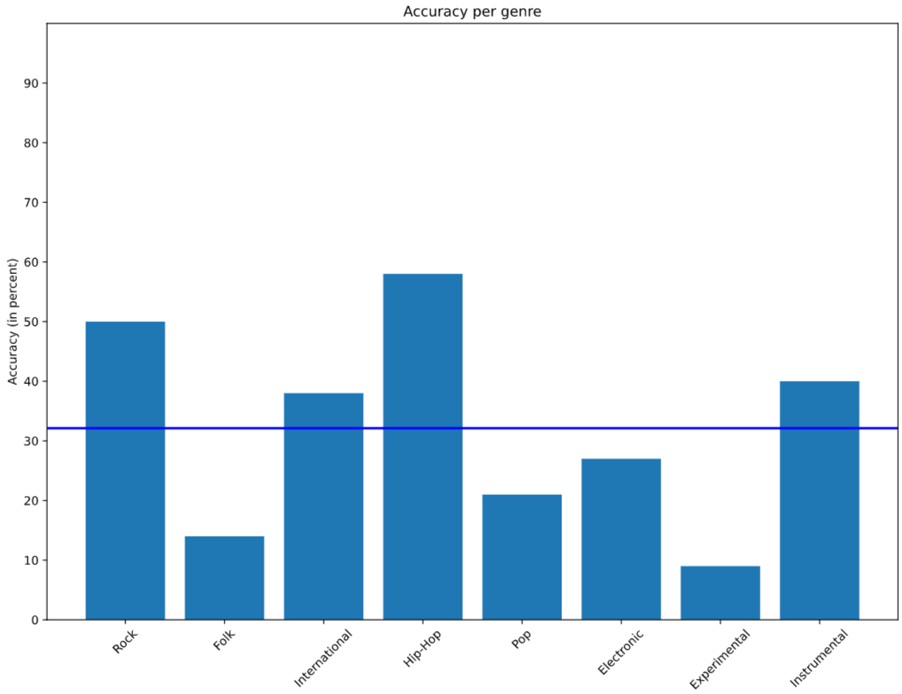

Figure 2: Training vs Test Data

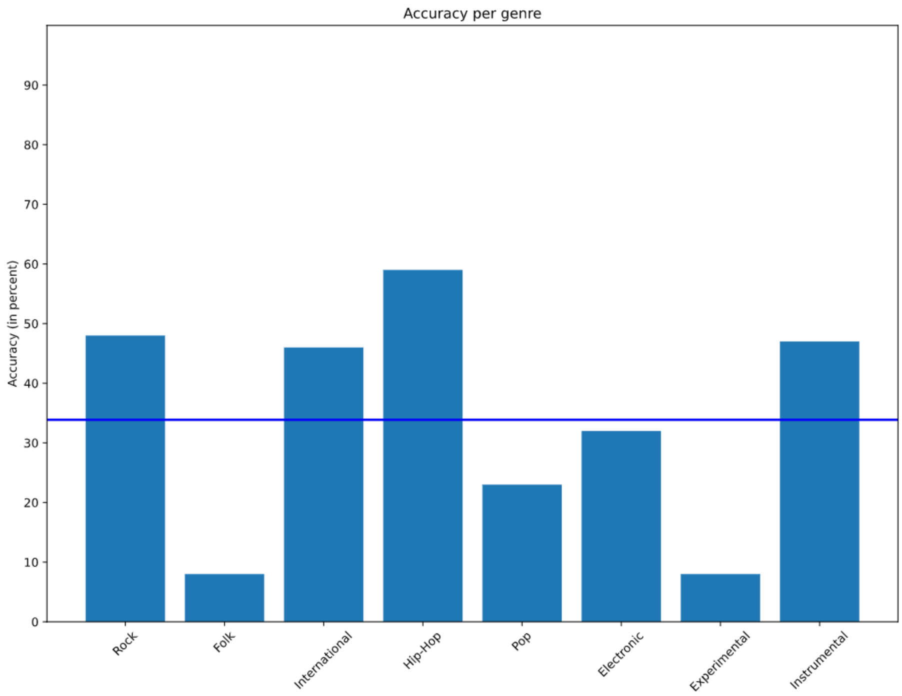

Figure 3: Training and Validation vs Test Data

When evaluating against the final testing data the model trained on the only the training data and accuracy of 32.125% is achieved, while the model trained on the training and validation data achieves an accuracy of 33.875% with the following hyperparameters.

Parameter	**Value**
number_of_hashtables	20
hash_length	25
subset	small
feature_fields	mfcc
measure	Cosine
k	20
magic_number	8000

Table 5: Parameters used for runs against test data

Contributions

The tasks and corresponding contributions from each member are listed below.

Task	E. Guliev	J. Rass	C. Wiskott
Initial Project Setup	✓	✓	✓
Task distribution planning	✓	✓	✓
Data exploration	✓	✓	✓
Initial Draft Version	✓	✓	✓
Merging the solutions	✓	✓	✓
Global refractor, splitting algorithm into classes	✓	✓	✓
Execution, Optimizing parameters	✓	✓	✓
Testing, Evaluating accuracy	✓	✓	✓
Report	✓	✓	✓

Every team member tried to develop their own version as an initial draft, which got merged together to form the final algorithm. Testing the hyperparameters was also split up between the team members to split up the run time for validating the parameters.