Genre Classification with Lyrics

732A92 - Text Mining 2019Fall

mimte666 - Mim Kemal Tekin

3/15/2020

Abstract

In the field of Natural Language Processing, there are many ways to represent a word as numerically. In this project, we focus to compare some of state-of-the-art embedding technologies on a classification task. This paper implements a music genre classifier from lyrics by using DistilBERT embeddings as contextual embedding and GloVe, Word2Vec as uncontextual embeddings. We compare their performance in the sense of accuracy, F1-Score, computational complexity and training time on this specific task. We use a Long Short Term Memory classifier. As a result, we observe that GloVe outperforms than others. BERT is in the second place without fine-tuning, but it has the most complexity. GloVe achieves 39% after 35 minutes, BERT achieves 38% after 500 minutes and Word2Vec achieves 28% after 285 minutes of training.

Contents

Introduction

Numerical representations of a word are called word embeddings. Word embeddings are widely used for Natural Language Processing applications. As they can be generated from the statistics of a corpus, they can be trained on some loss that is calculated to find word similarities or different objectives. In this research, we aim to test different embeddings and compare their performances.

Word embeddings that are used in this project are from two different sub-categories of embeddings. One of them is BERT which is a contextual word embedding, and the other two are GloVe and Word2Vec which both are uncontextual word embeddings. All of them are trained on different objectives, this topic is explained more in the following sections. Contextual word models generate different embeddings according to the context of the sequence and the position of the word in the sequence, while uncontextual word models are trained on a corpus and return only the pretrained value of the word. Therefore, it is hard to capture different meanings of synonyms with uncontextual word models, because they always return the same embeddings for one word.

Motivation

In the last decade, recommender systems have become very important. The recent tools provide easy and fast development processes on applications and this carries the competition very high level. One of the biggest topics that play a role in this competition is recommender systems. Since the users can access the same content on different applications, they give more value for getting more accurate personalized recommendations. Therefore, users want to use the application that understands and gives suggestions on what the user is interested in to see. Such customization for the users can be done by using a recommender system that uses strong features.

In the music player applications, most of the people, besides creating their playlists they listen to personalized playlists for them and also automatically generated playlists by the provider or an algorithm. Also we can see the impact of adding these personalization features on the increasement of Spotify users from Spotify statistics[1].

One of the core features of this task can be genres of music. There are different approaches to automatic music genre classification by using an audio context. In this project, since it is in the scope of the Text Mining Course at Linkoping University, we will use lyrics to classify genres. Also in this research we aim to compare the performances of word embeddings with a deep learning method.

Related Work

There are several related works for genre classification from lyrics. Some of them use traditional machine learning approaches such as Support Vector Machines (SVMs), k-Nearest Neighbor (k-NN), Naive Bayes (NB) and Logistic Regression. Also there are some approaches with deep learning methods with recurrent neural networks and convolutional neural networks. Unfortunately, according to (McKay et al. (2010)), lyrics have less effective information on genre classification.

One of the researches covered some machine learning methods, Long Short Term Memory (LSTM) recurrent neural networks and hierarchical attention networks on a very large dataset which has close to 1 Million song lyrics (Tsaptsinos 2016). This research uses GloVe word embeddings with classification algorithms. They concluded that although hierarchical attention networks do not perform better (almost same) than LSTM, the also mention that the required training time for LSTMs is much more than HAN.

Another researcher (Dutta 2018) used the same data set that we used with different split and preprocessing. They used TF-IDF instead of pre-trained word embeddings.

[1] https://www.goodwatercap.com/thesis/understanding-spotify#winning-strategies

Theory

Long Short Term Memory (LSTM) Recurrent Neural Networks

LSTMs are based on recurrent neural network idea. Recurrent neural networks (RNN) are supported for sequential data and they can learn dependencies between observations. This makes it very suitable to use in natural language processing tasks. The problem with RNN is that they suffer from vanishing and exploding gradients and some researchers (Bengio, Simard, and Frasconi 1994) analyzed this problem in detail. After investigating this problem and (Hochreiter and Schmidhuber 1997) proposed LSTM to learn better long dependencies.

LSTMs use a gated structure to overcome the problems that RNNs have. As we can see from Figure 1, in one LSTM cell we have several components: (1) Forget gate, (2) Input gate, (3) Cell state and (4) Output gate. The learned historical information is transferred with cell state and during this process it will be modified by other gates. There are some operations that provide learning and forgetting effects in the gates. Also we see that there are two inputs and two outputs in the horizontal directions of the cell. Upper input-output which is transferred by cell state is called as long term memory (LTM) (C_{t-1} and C_t where t : time step) and below input-output is called as short term memory (STM) (h_{t-1}/h_t). Also in the formulas of the following sub-sections the current input is represented as x_t. We will discuss the operations in each gate/component in the following sub-sections.

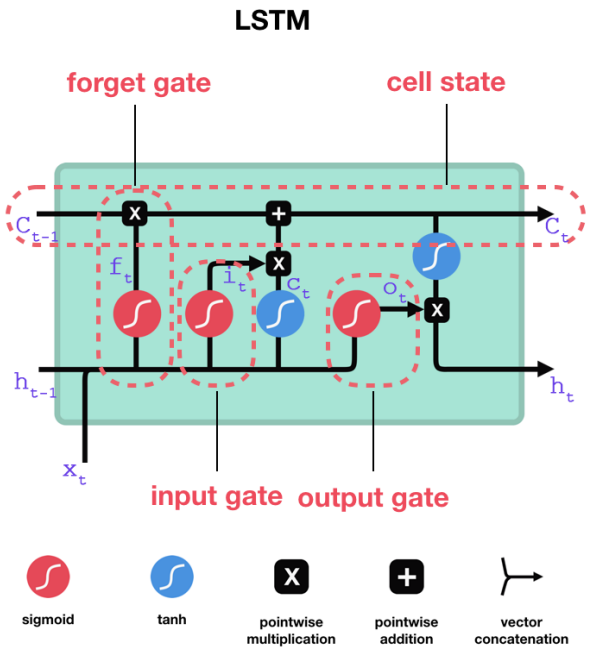

Figure 1: LSTM Cell (The source image[2] is annotated in order to be more clear about following notation.)

[2]https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

Forget gate

This gate decides which information from LTM will be kept or forgot. Input x_t and previous hidden state h_{t-1} are concatenated and passed through sigmoid function to decide which values will be updated. Since the sigmoid function maps the input value into (0, 1), it can be used to determine which values will be updated. Close values to 0 mean that forget that value and 1 mean to keep the value. In this way we forgot some values according to the information that we get from the STM and current input. In other words, we forget irrelevant history with short past and the current time. Formulation of this gate as following:

$$f_t = \sigma(W_f[x_t, h_{t-1}] + b_f)$$

Input gate

In the input gate, similarly to the forget gate, we pass a concatenated vector through sigmoid to decide which values will be kept. After this we also pass the concatenated vector through tanh function to map the values into (-1, 1) and help the network to regulate. We store this resulting tanh to use as input and map this vector with the output of the input gate in order to decide which values will be used for updating the next LTM.

$$i_t = \sigma(W_i[x_t, h_{t-1}] + b_i)$$

$$c_t = \tanh(W_c[x_t, h_{t-1}] + b_c)$$

Cell state

Now we have everything to calculate C_t (new LTM). We multiply point wised the values we have from previous gates with C_{t-1} (previous LTM). And we add input and forget gate outputs. In this way, we have the opportunity to forget some of the information from long term memory. We do this with the formula below:

$$C_t = f_t * c_{t-1} + i_t * c_t$$

Output gate

Finally we are ready to calculate the value that will be the output of cell and new short term memory. First we calculate o_t which is the output of the output gate and later we will multiply o_t and scaled LTM point wised to get the output of the gate.

$$o_t = \sigma(W_{xo}[x_t, h_{t-1}] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

Word Embeddings

Embeddings provide a numerical representation of words, therefore we can use them to train models and calculate the loss. As we can train our embeddings from scratch, we can use any of existing framework that has already proven its success in this field. We use three of them in this study. We can see more information about them in the next sub-sections.

Word2Vec

Word2Vec embeddings (Mikolov, Chen, et al. 2013)(Mikolov, Sutskever, et al. 2013) are non-contextual word representations that are based on word similarities. Non-contextual means that when we use these word vectors, the vector does not have information about the current sentence context. These types of vectors are trained on a large corpus by considering word similarities or different methodologies. Mikolov et al. suggest two architectures; Continous Bag-of-Words Model (CBOW) and Skip-gram Model. CBOW is based on predicting the word by using the words that are located before and after the word. The Skip-gram model tries to achieve the opposite of CBOW. Instead of predicting the word from its neighbour words, it predicts some words if they are neighbours of the input word. As a final architecture Word2Vec uses Skip-gram with Negative Sampling which means that the training samples have one input word, one potential neighbour word as input and a ground truth that says if these words are neigbours or not. The complete task suits what we explain for Skip-gram, and Negative Sampling is having this kind of training samples (if words are neigbour target is 1, else 0). We know what words are neigbours of this word and we sample some not neighbour words from the sentence by using a distance range, so these sampled values are negative samples. Each sample processed with a dot product and sigmoid function and the result will give the probability of being neighbour. As a final step loss is calculated and parameters (embeddings) are updated according to the loss. This topic is also overviewed and supported with nice visualizations by (Alammar 2019).

GloVe

GloVe (Pennington, Socher, and Manning 2014) is another successful non-contextual word representation. GloVe's model, unlike Word2Vec uses the advantages of both local context and global corpus statistics. As we discussed in the Word2Vec topic, they are trained with the Skip-gram method which is a local context window method. These kinds of local context methods can perform better in analogy tasks, but they have the disadvantage of not using global statistics to learn the words' generalized meaning. While GloVe uses global corpus settings, authors suggest a new objective that can be interpreted as a "global skip-gram" model. For named entity recognition (NER) task, GloVe outperforms other methods including Word2Vec and CBOW models.

BERT

BERT (Bidirectional Encoder Representations from Transformers) is a contextual word embedding model which means the embeddings are not static like GloVe and Word2Vec. The embedding values change by sentence context and the word's position in the sentence. Unlike other deep bidirectional representations (ELMo, GPT), BERT is a pretrained unsupervised and real bidirectional way. In training, it takes account of both the left and right contexts jointly. This is achieved by using a masked language model that randomly masks some of the tokens. This approach decreases unidirectionality because the model is trained by predicting the original word of the masked word based on its context (Devlin et al. 2018).

BERT Tokenization BERT tokenizer, after text normalization and punctuation splitting, runs the WordPiece tokenization method. This method works like splitting tokens into sub-tokens in order to capture more information about its context.
The BERT tokenizer model has 30,000 tokens. This vocabulary contains:

1) whole words
2) subwords
3) starting subwords
4) individual characters
(McCormick and Ryan 2018)

We can see the example tokenization from research Github page of (Devlin et al. 2018) below:
`Input  = John Johanson 's house`
`Output = john johan ##son ' s house`
As we can see `Johanson` tokenized as `johan ##son`. This `##son` is a subword and it has its embedding vector. This shows that a word can be split into small pieces and process it like that. These subtokens provide more contextual meaning for the whole word.

Data

Lyric Dataset

The dataset used in this project is a scraped song metadata and lyrics from a lyric provider.[3] The dataset is public and available in Kaggle.[4] The dataset has 362237 song data in total and it has `song, year, artist, genre, lyric` attributes. As the scope of this project is predicting genre from only lyric data, we will be using only these features. We will split the data into three subsets (training: 0.8, validation: 0.1 and test: 0.1).

We can see one observation of the data like this:

```
# song:
# dream-on
#
# year:
# 2009
#
# artist:
# aerosmith
#
# genre:
# Rock
#
# lyric:
# Every time when I look in the mirror All these lines on my face getting clearer
# The past is gone It went by, like dusk to dawn Isn't that the way Everybody's
# got the dues in life to pay I know nobody knows Where it comes and where it goes
# I know it's everybody sin You got to lose to know how to win Half my life
# ...
```

[3] https://www.metrolyrics.com/
[4] https://www.kaggle.com/gyani95/380000-lyrics-from-metrolyrics

Overview and Preprocessing

```
Data Shape    : (362237, 6)
Song Count    : 250473
Year Count    : 52
Artist Count  : 18231
Genre Count   : 12
```

Figure 2: Dataset Details

```
count    266557.000000
mean        227.136504
std         157.577703
min           1.000000
25%         130.000000
50%         192.000000
75%         279.000000
max        8195.000000
```

Figure 3: Word Count

Details of rach field in the dataset are listed in Figure 2. We see that we have 12 genres and in total we have 362237 tuples. Later on we investigate word counts in these observations (Figure 3) and we see that we have extreme values in the dataset. First, we remove these extreme values in order to get more optimal dataset. Some of lyrics have only one word, they were usually instrumental songs that tagged only as instrumental, so we have to remove those. We exclude the observations that have less than 100 words and more than 400 values and we can see resulting lyric distribution in Figure 4. Also it is interesting to see work count distributions by genres. We can say from Figure 5 that Hip-Hop genre has more words in lyrics and other genres look like balanced.

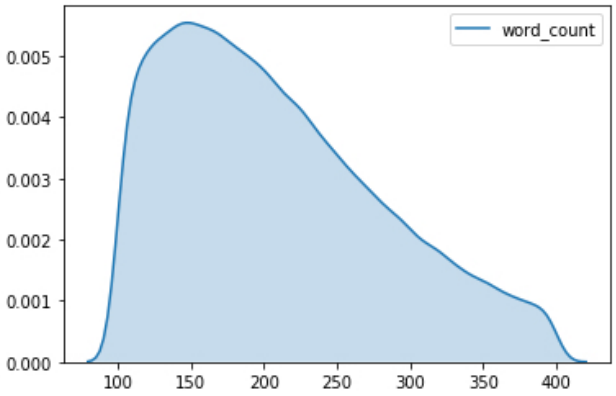

Figure 4: Word count distribution after excluding extreme values.

Another important thing is proportions of genres. We want to use a balanced data as much as possible, so that we can learn a better class differentiation. In order to achieve this we plot observation counts and proportions for all data in Figure 6 (left) and we decide to use only 7 most frequent genres as we can see its distribbution in Figure 6 (right). As a last thing we create another field for genres, since Pytorch expects it as a class id. So, we assign one unique class id for each genres.

Next step is doing some more cleaning after splitting the training, validation and test subsets. We split the data with proportion of `{training: 0.8, validation: 0.1, test: 0.1}`. As a result, we have `{training:`

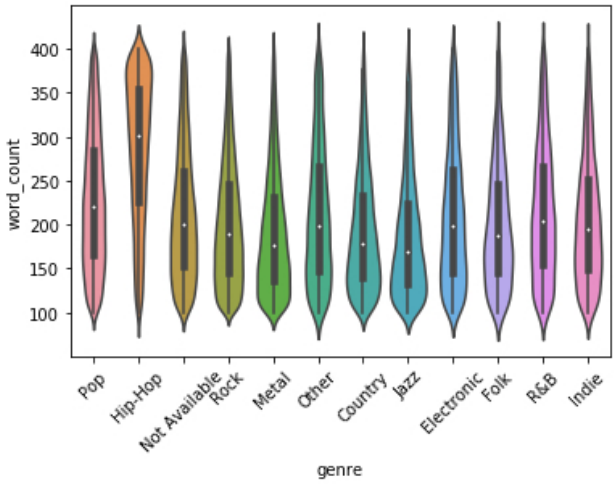

Figure 5: Word count distributions by genres

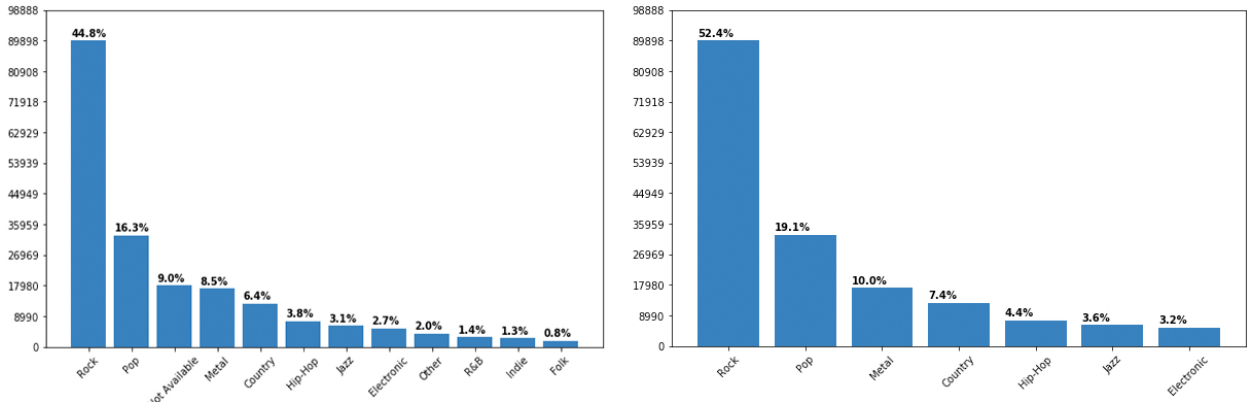

Figure 6: Genre Proportions. Left: Before cleaning, Right: After Cleaning

`137498, validation: 16995, test: 17166}` observations for each subsets. But we want to train on a balanced data, not a data like in Figure 6. So we balance the training data by sampling each genre with the size of smallest genre we have in the training split. This will give us such a data `{training: 30142, validation: 16995, test: 17166}` obbservations for each.

Now we are done with data refactoring. We have our data splits and our next step will be preparing the data for training. We will get embedding indexes from our embedding models and save the data like this. This will make training faster. As we can do this as a preprocessing part, we implemented a way to do this during training too, so that we will not have to do long preprocessing and we can do it dynamically just by passing the original sentences/lyrics to the model. It will automatically apply a general tokenization and get embedding indexes for the target embedding model (word2vec, GloVe or BERT). We will explain how these embeddings are used and which tools are used to get these vectors in Methods topic with more technical details.

Methods

Embeddings

We used three pretrained embedding models which are listed below. For accessing pretrained embeddings we use `gensim`[5] package for word2vec and GloVe; and `transformers`[6] package for BERT model. We applied tokenization before each embedding models. We use `spacy`[7] package for tokenization of word2vec and GloVe embedding inputs and BERT has its own tokenizer in `transformers` package. Pretrained models that are used:

- word2vec: `word2vec-google-news-300`; embedding size: 300
- GloVe: `glove-wiki-gigaword-300`; embedding size: 300
- BERT: `distilbert-base-uncased`; embedding size: 768

Models

As a baseline we used random classifier which is already implemented as `DummyClassifier` in `sklearn`[8] package. This model classifies without using features, it generates predictions randomly and we used uniform strategy which means each class has equal probability to be chosen.

For the main models that we want to investigate, we used a simple LSTM architecture that is 2 layers for each training. The input of each model is embedding indexes which are already preprocessed and these observations may have different lengths and this is not suitable for LSTM. We use a padding factor that is 200. This means that longer sequences than 200 tokens will be shorten to 200 and shorter sequences will be filled with zero paddings. By this way we have each observation in input shaped as (200). For Word2Vec and GloVe we have one Embedding Layer which takes embedding indexes as input and returns vectors shaped as (200, 300). For BERT we have directly DistilBert model which is can be enabled for training, but due to our hardware limitations we could not tune BERT for this task specifically. BERT model returns (200, 768). After this layer everything is same for both models. We use LSTM model for this task and after a dropout layer we classify outputs with a linear fully connected layer that maps the LSTM output to the label space and LogSoftMax activation function to get output probability distribution. We can see the architecture illustration in Figure 7.

For loss criterion we use a NLLLoss[9] function which is negative log likelihood loss function. This loss function is often used in classification tasks when we have many classes. Also when it is used with LogSoftMax

[5] https://pypi.org/project/gensim/
[6] https://huggingface.co/
[7] https://spacy.io/
[8] https://scikit-learn.org/stable/
[9] https://pytorch.org/docs/stable/nn.html#nllloss

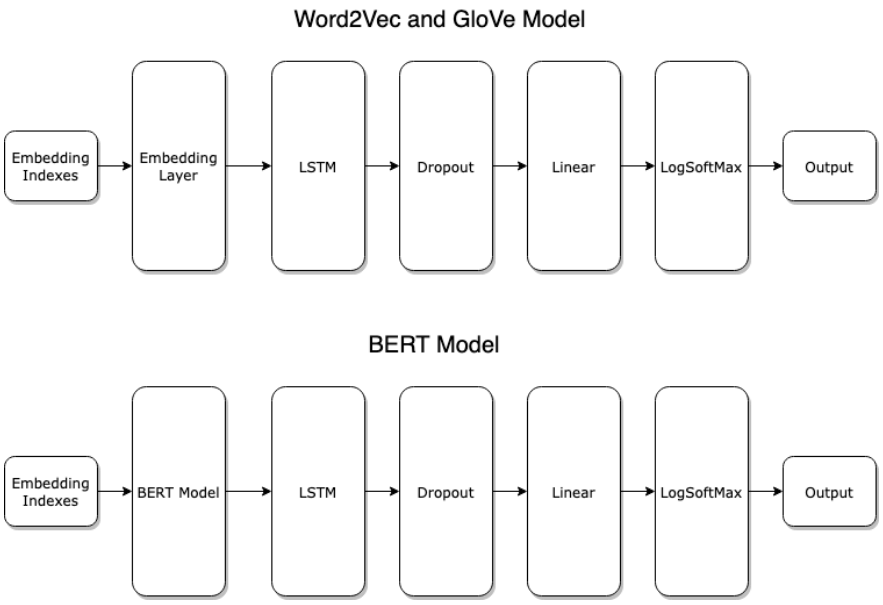

Figure 7: Model Architectures. Upper: Model for Word2Vec and Glove, Lower: Model for BERT

activation it is same as cross entropy loss. As an optimiser, we use Adam Optimizer[10].

Results

Table 1: Accuracies and Other details for models.

	Accuracies (%)			F1-Score (%)	Details	
Models	**Train**	**Validation**	**Test**	**Test.**	**Epoch.No**	**Time.mins**
Baseline	14	14	14	17	0	0
DistilBERT	50	38	38	39	88	500
GloVe	52	39	39	41	91	35
Word2Vec	39	29	28	28	88	285

The training has been done for 100 epochs for each model without early stopping and the epoch that has the lowest loss is used for the evaluation. We can see from the table above all embeddings outperformed than random baseline. The only difference between running parameters is the learning rate. For BERT and Word2Vec models we have a learning rate $5e-6$, but for GloVe we have $1e-5$. With bigger learning rates no convergence is observed. As we can see that the longest training is BERT. It is significantly slower to train than other non-contextual embeddings.

We also measured the performance by calculating F1-Score[11] which is the harmonic mean of the precision and recall. This gives a better understanding of how precision and recall metrics are balanced for the classifier. So with F1-Score we can evaluate the classifier independent from the test class (negative/positive observations) distributions. It is calculated as the following formula:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

[10]https://pytorch.org/docs/stable/optim.html#torch.optim.Adam
[11]https://en.wikipedia.org/wiki/F1_score

Discussion

From Table 1 we can see that all of the models performed better than the baseline. The best models on test data are GloVe and DistilBERT. They are very close to each other but GloVe is better in both Test accuracy and F1-Score.
We see that all of the models that we tried converged the best loss very close epoch counts, but another factor that we can see is training time. We see that DistilBERT has the longest training time and the shortest is GloVe, although it is the best performance that we could get on this project. We see that Word2Vec underperforms a lot when we compare it with the other results we have and it takes more time to train than GloVe.

When we compare our results with other researchers' (Dutta 2018) who reported their best results for Multi-Layer Perceptron (MLP), our results underperform. They reported 63.5% accuracy with MLP and using only TF-IDF Vectors. Although they use the same dataset with us, this is still not a fair comparison, since they used 5 genres with 5000 lyrics for each on training. Additionally, they do not have details about their validation set.

For trainings in this project, we used Google Colab[12] and Google Cloud[13] which we had many limitations. We could not run enough experiments to improve models better. So just to notice you these results should not be considered as final results in this field. The project source code[14] is available on GitHub and the source can be used as a basis. The implementation is in Python and Pytorch[15] is used as a framework.

Conclusion

In this project, we see that BERT which is a contextual embeddings performs slightly lower than an uncontextual embedding GloVe. But one of the disadvantages of using contextual embedding is that they are computationally expensive than static embeddings, especially compared to GloVe. BERT's training time takes 14 times of GloVe's training for our architecture.
One of thing that we did not consider while training BERT is finetuning. This process would add another training process for BERT, but as the original paper of BERT (Devlin et al. 2018) says that with finetuning it is possible to get state-of-art results. This will of course add more computational cost that we could not deal easily.
As a final comment we can say that there may be a possibility that this method can apply on a larger dataset that has more lyrics and more balanced in the sense of labels. This can give some improvements.

[12]https://colab.research.google.com/
[13]https://cloud.google.com/
[14]https://github.com/rubicco/music-genre-classification
[15]https://pytorch.org/

References

Alammar, Jay. 2019. "The Illustrated Word2vec." http://jalammar.github.io/illustrated-word2vec/.

Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. 1994. "Learning Long-Term Dependencies with Gradient Descent is Difficult." *IEEE Transactions on Neural Networks* 5 (2): 157–66. https://doi.org/10.1109/72.279181.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." *CoRR* abs/1810.04805. http://arxiv.org/abs/1810.04805.

Dutta, Dpayan. 2018. "Lyric Based Music Genre Classification." https://github.com/dipayandutta93/Music-Genre-Classification-using-lyrics.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. "Long Short-Term Memory." *Neural Computation* 9 (8): 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735.

McCormick, Chris, and Nick Ryan. 2018. "BERT Word Embeddings Tutorial." https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial.

McKay, Cory, John Ashley Burgoyne, Jason Hockman, Jordan B. L. Smith, Gabriel Vigliensoni, and Ichiro Fujinaga. 2010. "Evaluating the genre classification performance of lyrical features relative to audio, symbolic and cultural features." *Proceedings of the 11th International Society for Music Information Retrieval Conference, ISMIR 2010*, no. January: 213–18.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Efficient estimation of word representations in vector space." *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, 1–12. http://arxiv.org/abs/1301.3781.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Distributed representations ofwords and phrases and their compositionality." *Advances in Neural Information Processing Systems*, 1–9. http://arxiv.org/abs/1310.4546.

Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. 2014. "GloVe: Global vectors for word representation." *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, 1532–43. https://doi.org/10.3115/v1/d14-1162.

Tsaptsinos, Alexandros. 2016. "Music Genre Classification by Lyrics using a Hierarchical Attention Network."