English Lyrics Origin Classification: *native* VS *non-native* speakers

Adela-Nicoleta Corbeanu, NLP
January 2025

Task & Dataset Summary

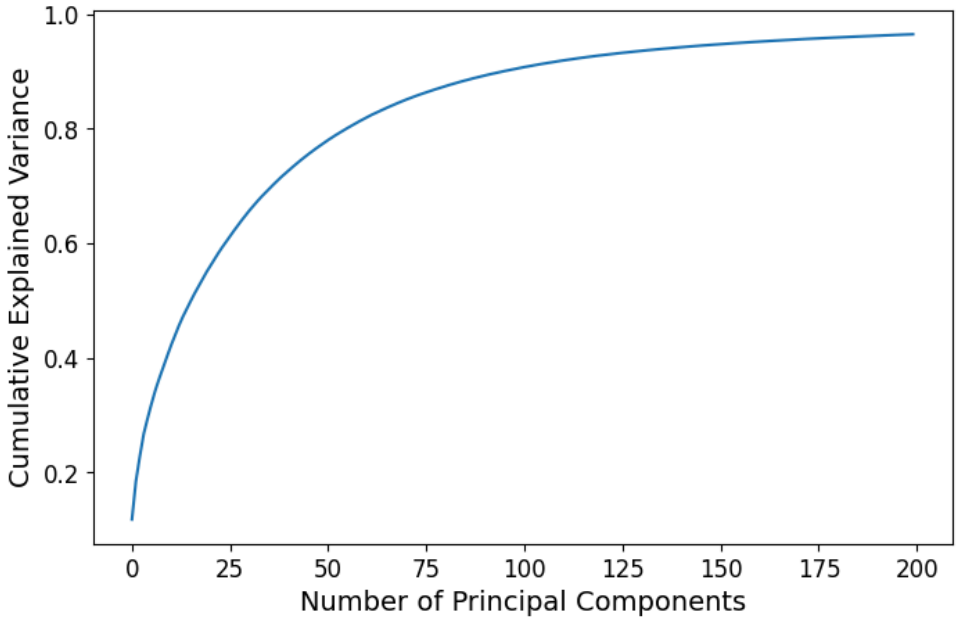
The *English-Lyrical-Origins* dataset is designed for a binary classification task, where the goal is to predict whether the lyrics of a song written in English were created by a *native* English speaker or a *non-native* one. The songs originate from 34 different nationalities.

Preprocessing

I have performed some basic preprocessing techniques:

- Lowercasing: converting all data to lowercase.
- Punctuation removal: keeping only alpha-numeric characters.
- Stopwords removal: removing the most common words (“the”, “is”, etc.)

Along with those, I used **PCA** (Principal Component Analysis) with up to 500 components, but looking at the explained variance prompted me to keep less:



Despite capturing less than 30% cumulative variance, I have found that **PCA** with ~5 components yields the best classification scores. The reason for this I believe lies in the fact that the two categories of the data (*native* and *non-native*) are, in reality, very similar, and separating them is non-trivial, therefore any feature that can differentiate them even a little is a significant win. The scores themselves are pretty low, but this usage of **PCA** proved to make the greatest difference between unsupervised learning and random choice, meaning it did capture 1-2 relevant features. By looking at the variance of each feature, we see that the variance of the latter ones becomes small enough to be considered noisy/insignificant:

Explained variance ratio: [0.11836 0.06628 0.04354 0.03946 0.02742]

Lyrics most aligned with Principal Component 1 (highest values):

- ➔ *lost nowhere searching home still turning past gone time omen showed took away preparations done can...*
- ➔ *final destiny sunrise never came still night lamp never faded away farewell word afterglow brave mor...*
- ➔ *cleared fog veiled around blurred sights suddenly im longer aching honor plights rising moon skin pe...*
- ➔ *beloved name inside heart fleeting glance became start missing word still awaiting wretched deceptio...*
- ➔ *abode mongst stars waiting long enough last breath life stare nothing right times resembling devils ...*

Lyrics least aligned with Principal Component 1 (lowest values):

- ➔ *mo bounce bounce bounce bounce bounce bounce bounce bounce motherfuckin house mo bounce mothe...*
- ➔ *get want like click want pic like click cheers glass like click cash register goes click cant fuck c...*
- ➔ *intro tony yayo 50 cent shady yeah run know actin like dont know run yeah know actin like dont know ...*
- ➔ *uhoh runnin breath oh got stamina uhoh running close eyes well oh got stamina uhoh see another mount...*
- ➔ *told boy kiss girl take trip around world hey hey bop shuop mbop bop shuop hey hey bop shuop mbop bo...*

By intuitively analyzing data in relation to the first principal component, it seems that it avoids “slangy” and colloquial terms, which are normally not present in literature or other common sources for learning English, pointing therefore towards native speakers. Some examples are the use of interjections such as “yeah”, as well as the auxiliary verb “got”.

Common words in high PC1 lyrics:
still, would, past, away, cant, last, around, never, word, rising

Common words in low PC1 lyrics:
bounce, dont, yeah, like, know, give, greatest, im, got, boys

***** Additionally, the dataset specifies that for classification tasks all text between brackets (example: “[Verse 1]”, “[Outro]”, etc.) should be removed as it may contain artists names (“[Chorus - John Lennon]”).

Vectorization

After preprocessing the data, the next step involves converting song lyrics into numerical vectors that can be used by the models. For this, I have tried two approaches:

- I have used **`Sentence-BERT`**, a transformer-based approach that generates contextualized sentence embeddings. The pretrained model I used in this approach is ***`princeton-nlp/unsup-simcse-bert-base-uncased`***, which has been pre-trained using the English Wikipedia in an unsupervised manner.

- I have also tried **`TF-IDF`**, a traditional approach based on word frequency, which ended up performing slightly worse than **`Sentence-BERT`** in all cases. It especially affected **`PCA`**, making it yield a variance of only 1% for the first principal component, as opposed to 12% with **`Sentence-BERT`**.

Method 1: K-Medoids

General performance: To measure the performance I have mainly used two scores: **`ARI`** (Adjusted Rand Index) and **`Silhouette score`**. While they have both been quite low, for this task I chose to fine-tune hyperparameters based on **`ARI`**. The fine-tuning process has been conducted based on validation data (independent of test data). The table below contains the **`ARI`** scores for various combinations of number of **`PCA`** components, number of clusters, and distance functions:

ARI Score Visualization

PCA Comps	Clusters	euclidean	manhattan	cosine	chebyshev	correlation
2	2	0.05	0.00	0.10	0.06	0.00
2	3	0.07	0.08	0.09	0.10	0.08
2	4	0.06	0.06	0.07	0.05	0.08
5	2	0.10	0.03	0.13	0.13	0.16
5	3	0.08	0.09	0.09	0.08	0.12
5	4	0.07	0.06	0.06	0.05	0.10
50	2	0.05	0.00	0.00	0.06	0.00
50	3	0.10	0.00	0.07	0.08	0.07
50	4	0.07	0.01	0.06	0.08	0.06
10	2	0.00	0.00	0.00	0.00	0.00
10	3	0.09	0.09	0.09	0.09	0.09
10	4	0.08	0.09	0.08	0.08	0.08
20	2	0.12	0.11	0.10	0.10	0.11
20	3	0.09	0.10	0.09	0.10	0.09
80	2	0.00	0.00	0.00	0.00	0.00
100	2	0.00	0.00	0.00	0.00	0.00

I have implemented this using the **`sklearn_extra.cluster`** module and I experimented with two values for the **`'method'`** parameter (specifying the algorithm used to perform the clustering), **`'alternate'`** and **`'pam'`**, but have found little to no difference between them, so I generally went with **`'alternate'`** since it had slightly lower computing time.

When using the best found hyperparameters, the scores and the clusters look as follows:

Train data `ARI`: 0.10, **Train data `Silhouette`:** 0.22

Test data `ARI`: 0.07, **Test data `Silhouette`:** 0.22

Train Cluster Distributions:

Cluster 0: Native = 630, Non-Native = 1246

Cluster 1: Native = 1345, Non-Native = 720

Test Cluster Distributions:

Cluster 0: Native = 167, Non-Native = 304

Cluster 1: Native = 333, Non-Native = 196

It is easy to notice that cluster 0 corresponds to the *non-native* label, while cluster 1 corresponds to the *native* label.

Comparison to random choice: Since we are doing binary classification, random choice should have around 50% accuracy. Based on the results above, we can compute the following scores:

Metric	Non-Native	Native
Precision	0.645	0.629
Recall	0.608	0.666
F1 Score	0.625	0.647
Accuracy	**63.7%**	

With an accuracy of 63.7%, the model surpasses random choice significantly.

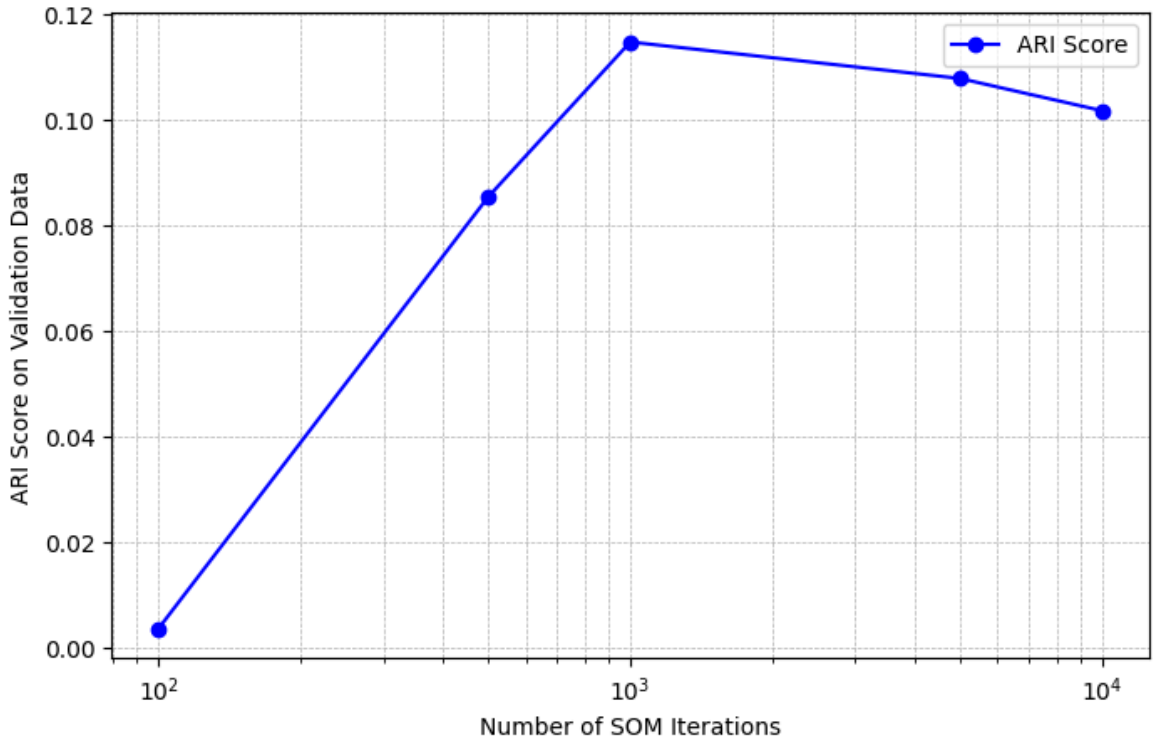
Comparison to supervised baseline: Obtained with a very basic, not tuned, pretrained model **`'bert-base-uncased'`**, the supervised baseline is just slightly higher than random choice, with a value of 55% accuracy. With an accuracy of 63.7%, our K-Medoids model transcends this baseline.

Method 2: SOM (Self-Organizing Map)

General performance: Similarly to the previous method, I measured performance using primarily the **`ARI`** score (that I also used for fine tuning on validation data), and secondly the **`Silhouette`** score. Along with the number of components in **`PCA`**, these are the other parameters I experimented with:

- *Grid size*, which defines the shape of the SOM map.
- *Learning rate*, which controls how much SOM updates weights per iteration.
- *Sigma*, which determines the neighbourhood influence during training.
- *Number of iterations*, which specifies how many training steps the SOM does.

In terms of **number of iterations**, I found the best ARI score to be achieved after 1000 iterations. The fluctuation can be noticed in the table below:

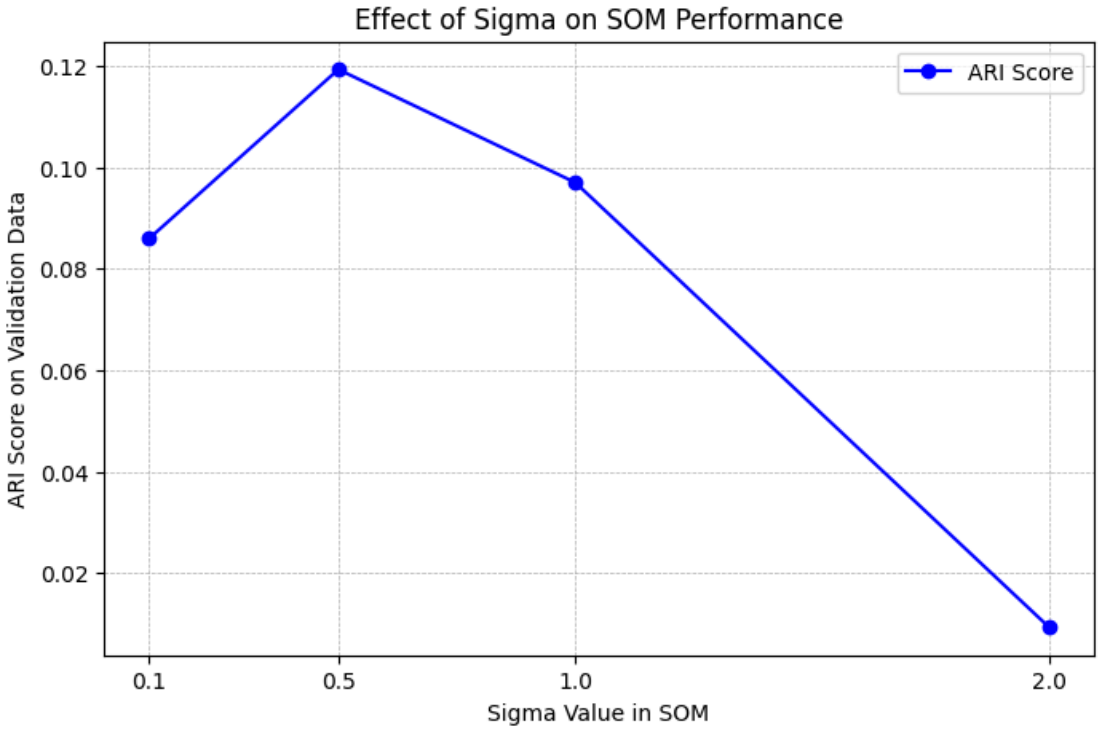


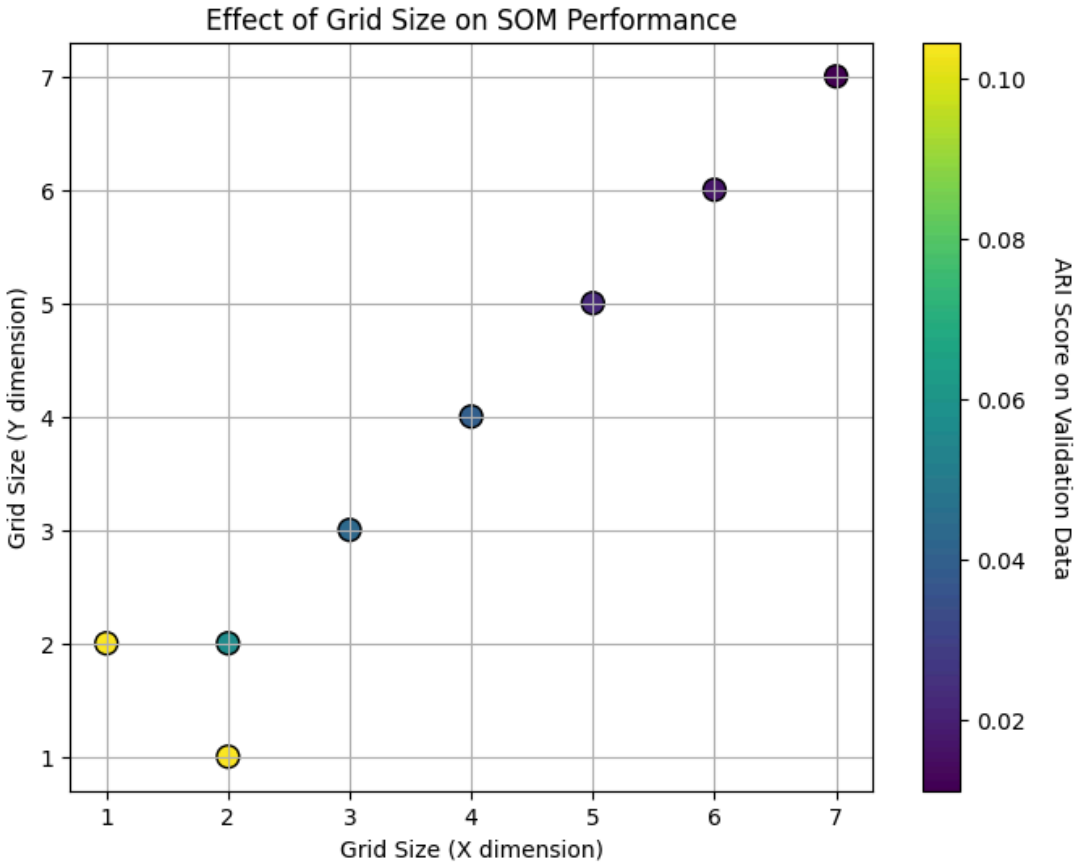
When training, there are two available methods:

- **`train_random`**: selects a random sample from the dataset in each iteration and updates the SOM (resulting in less sensitivity to the order of the data).
- **`train_batch`**: processes all samples of data in each iteration (supposedly resulting in stability).

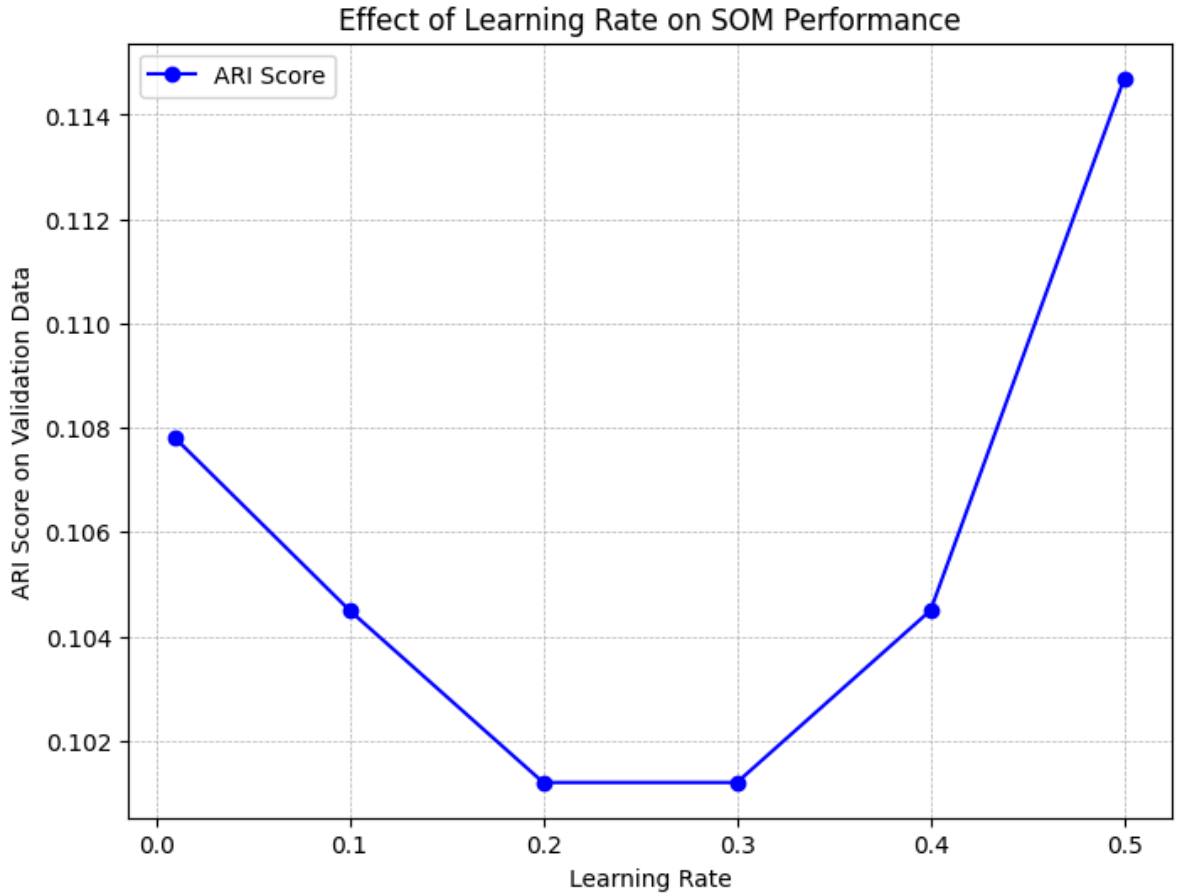
I have run all experiments with both of these options and have not noticed any slightly significant difference in performance.

The variations in ARI score for other parameters (such as Sigma, grid size, and learning rate) can be observed in the following graphs:





Although the ARI score for bigger learning rates seemed to be on an ascending trend, I decided to limit the learning rate to no more than 0.5 in order to avoid overfitting and keep the clusters stable:



I have implemented this using the **`minisom`** library.
When using the best found hyperparameters, the scores and the clusters look as follows:

Train data `ARI`: 0.08, **Train data `Silhouette`:** 0.27

Test data `ARI`: 0.07, **Test data `Silhouette`:** 0.30

Train Cluster Distributions:

Cluster 0: Native = 253, Non-Native = 660

Cluster 1: Native = 1525, Non-Native = 1108

Test Cluster Distributions:

Cluster 0: Native = 70, Non-Native = 202

Cluster 1: Native = 430, Non-Native = 298

It is easy to notice that cluster 0 corresponds to the *non-native* label, while cluster 1 corresponds to the *native* label.

Comparison to random choice: With an accuracy of 63.2%, the SOM model surpasses 50%, the value of random choice.

Metric	Non-Native	Native
Precision	0.591	0.743
Recall	0.860	0.404
F1 Score	0.700	0.523
Accuracy	**63.2%**	

Comparison to supervised baseline: With an accuracy of 63.7%, the SOM model exceeds the supervised baseline of 55% set by **`'bert-base-uncased'`**.