Song's mode classification problem

Benfenati Luca
Student ID: 286582
Politecnico di Torino
Torino, Italia
s286582@studenti.polito.it

***Abstract*—The report addresses a binary classification problem of song's mode. Different approaches are proposed, both for the preprocessing step and the model selection. In particular, in the preprocessing pipeline, *multi-label binarizer* and *tf-idf vectorizer* are exploited, as well as different standardization and normalization functions. Then, some considerations are carried out on the best performing model, based on gradient boosting, which allows to reach satisfactory results.**

I. Problem overview

For the project, we have been provided with a dataset of songs, each one described by several features. Our goal is to develop a classification model which is able to assess, given the other descriptors, the mode of each song in the dataset. The mode defines whether a song starts with a Major or Minor chord and it can be encoded either with 1 or 0, respectively. The dataset is a sample of the *Spotify Database for Developers*, and it is divided into two different sets:

1) a *development set*, containing 136522 songs for which the mode is present;
2) a *evaluation set*, containing 34131 for which is not.

Our model will be trained and validated on the former and tested on the latter.

For what concerns the development set, first of all it is worth noticing that none of the columns contains missing values. Therefore, we do not have to deal with the problem of filling them. Neither there are duplicates, so we can skip the deduplicated preprocessing step. Only two columns are non-numerical: *artists* and *id*. In table I we reported some useful information about the numerical ones. Most of these are self-explanatory, such as *year*, *acousticness*, *danceability*, *duration_ms*, *key*, *liveness*, *loudness*, *popularity*. The meaning of the others is more subtle:

- *valence* describes the musical positiveness conveyed by a song. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed,a angry;
- *energy* represents a perceptual measure of intensity and activity. Tracks with high energy feel fast, loud and noisy (e.g. death metal has high energy, while Mozart prelude scores low on the scales);
- *explicit* specifies whether or not the song contains explicit language, such as strong words or reference to violence;
- *instrumentalness* describes whether a track contains no vocals. Track with low instrumentalness may be rap or spoken words;

TABLE I
Some useful information on numerical attributes

Column	Range	Mean	Std
valence	[0.0, 1.0]	0.53	0.26
year	[1921, 2020]	1976.8	25.92
acousticness	[0.0, 0.99]	0.50	0.38
danceability	[0.0, 0.98]	0.54	0.18
duration_ms	[5.9x10^3, 5.4x10^6]	2.31x10^5	1.28x10^5
energy	[0.0, 1.0]	0.48	0.27
explicit	[0, 1]	0.08	0.28
instumentalness	[0.0, 1.0]	0.17	0.31
key	[0, 11]	5.19	3.51
liveness	[0.0, 1.0]	0.21	0.17
loudness	[-60.0, 1.0]	-11.47	5.69
popularity	[0.0, 100.0]	31.43	21.83
speechiness	[0.0, 0.97]	0.10	0.16
tempo	[0, 243.5]	116.91	30.71

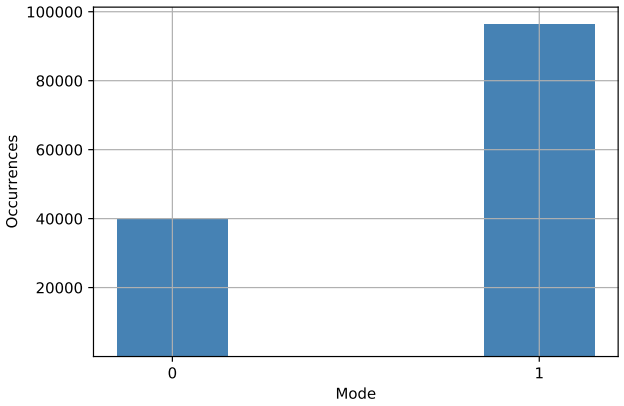

Fig. 1. Mode distribution

- *speechiness* detects the presence of spoken words in a song.

For what concerns the non-numerical columns, it is worth noticing that the *artists*, even if more than one, are stored as strings. In the preprocessing step II-A, the handling of this attribute will be vastly addressed. The *id* is an alpha-numerical code that uniquely identifies the song in the *Spotify Database*.

The *mode* column, the one to predict, can have values either 1 or 0. The ratio 1:0, i.e. Major to Minor Cord, is 2.4:1. As we can see in figure 1, we have indeed 96508 '1's and 40014 '0's.

II. Proposed approach

A. Data preprocessing

The input data contains a mix of numerical and non-numerical attributes. The preprocessing pipeline is therefore split into two separate sections.

Each numerical column has very different range of values (see table I). Then, the first stage in the preprocessing pipeline is a feature scaling step. This step is one of the most significant during the preprocessing of data, as shown in [1]: we need to exploit scaling so that one significant value does not impact, skew or dominate our model just because of their large magnitude. Even if some model are not affected by this problem, the performance of the ones we have chosen are heavily influenced. We considered three different scaler functions from scikit-learn [2] library:

1) *StandardScaler*, which standardize features by removing the mean and scaling to unit variance. The standard score of a sample x is:
$$z = \frac{x + \mu}{\delta} \tag{1}$$
where μ and δ are, respectively, the mean and the standard deviation of the training sample.
2) *RobustScaler*, which scale features removing the median and scaling the data according to the quantile range. The scaled score of a sample x, assuming the default interquantile range IQR, is:
$$x_{scaled} = \frac{x - 50^{th}perc}{75^{th}perc - 25^{th}perc} \tag{2}$$
3) *MinMaxScaler*, which transform features by scaling each features to a fixed given range. The scaled score of a sample x is:
$$x_{scaled} = \frac{x - min(x)}{max(x) - min(x)} \tag{3}$$

Standard scaler function is very well suited if individual features are distributed as the Gaussian distribution, or similarly. Robust scaler function takes into account the presence of outliers: the outliers are left out from the computation of the mean and the standard deviation. MinMax scaler function is the normalization of the attributes from the original range to a new defined range, typically $[0, 1]$. Analyzing the effect of each one of these scaler functions, we immediately discard the last one, since it is strongly influenced by outliers. On the other hand, the choice between the first two is not trivial, since their behaviors are similar. Even if not all the attributes are Gaussian distributed, we choose to implement the Standard scaler function. For most of the attributes, this one tends to find a wider range to arrange the scaled values, without concentrating them excessively. In figure 2 we reported a representative example of what described, in particular considering the *key* attribute. It is easy to notice that the Standard scaler function behaves better than the others, since it does not substantially increase the density values of the distribution. The behavior of these three functions with the other attributes is almost comparable to the one shown in this figure.

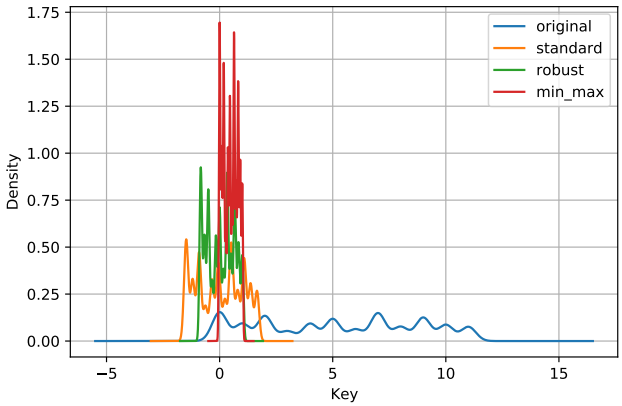

Fig. 2. Feature scaling effect on the *key* distribution

For what concerns the non-numerical attributes, we disregard the *id*, since it is unique for each row of our dataset and it is not possible to find common patterns among them. For the *artists* column, we may have more artists for each song, so firstly we clean each string from special characters, standardizing both different spacing and separation characters. Then we explore two different handling methods: we want to consider *artists* either as a categorical attribute or as a textual attribute.

For the first one, we exploit a custom *Multi-Label Binarizer* (MLB) from scikit-learn library, which allows us to encode multiple labels per instance. In our case, the labels are the different artists, while the instance is a single song. In order to implement it, we split each row of the artists column into a list of strings. Since we are dealing with lists of different lengths, the MLB is perfectly suited for this task: for each row of our dataset, it scans every element of the list, i.e. every artist, and encode it as if it were a category. Its behaviour is very similar to the most well-known *One-Hot Encoder*, but for multiple labels. The only parameter differing from the defaults is {*sparse_output=True*}, which creates a 23241-wide sparse feature matrix when training on cross-validation folds.

The second method, on the other hand, treats the *artists* column as textual data: after the cleaning process, each row is a string containing all artists. For the extraction of useful information from this kind of data, we exploit the scikit-learn's *Tf-idf Vectorizer*. Since we are dealing with a collection of documents ranging over the same topic, i.e. song's artists, the best choice seems to be considering the term frequency document frequency (tf-df). We are indeed interested in frequently occurring terms, both in a single document and in the whole collection. The tf-df of a term t in document d of a collection D corresponds to:

$$tf\text{-}idf(t) = freq(t, d) \cdot \log(freq(t, D)) \tag{4}$$

To remove the not meaningful words we set {*stop-words="english"*}. Training on our cross-validation folds generates a 20979-wide sparse matrix.

Since we are dealing with matrices that are comprised of mostly zero values, for both the MLB and the TF-DF methods we exploit sparse matrices. Sparse matrices use an alternate data structure to represent the sparse data: the zero values can be ignored and only the non-zero values are stored and acted upon.

The pipeline for the preprocessing and the following implementation of the classification model was built using scikit-learn's *Pipeline* and *ColumnTransformer*. This last one was deployed to deal with different preprocessing steps for different attribute types, i.e. numerical and non-numerical.

B. Model selection

The following classification algorithms have been tested:

- *Naive Bayes* [3] is a simple probabilistic classifier based on applying Bayes' theorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{5}$$

  where $P(A|B)$ is the posterior probability, i.e. the probability of hypothesis A given the data B. *Naive Bayes Classifiers* relies on the strong assumption that each feature is conditionally independent from the others. This could be counterproductive with real data. However, as shown in [4], the approach performs surprisingly well on data where this assumption does not hold.
  Among the different types, we selected the *Multinomial Naive Bayes*, mostly used in document classification along with TF-DF. In this case, feature vectors represent the frequency with which certain events have been generated by a multinomial $(p_1, ..., p_n)$, where p_i is the probability that event i occurs. The events, in the document classification case, are the occurrences of a word in a single document.
- *SVMs* [5] are a family of supervised machine-learning algorithms which find the hyperplanes based on the margin maximization principle. The optimal hyperplane is generated in an iterative manner, finding the maximum marginal hyperplane (MMH) that best divides the dataset into classes and minimize the error.
  For our project we selected the *LinearSVC*. It is similar to SVC with parameter {*kernel='linear'*}, but more scalable for large dataset. We choose it because of its higher flexibility in the choice of penalties and loss functions, and its higher training speed.
- *Gradient Boosting Machines* [6] are a tree-based ensemble models, where the learning procedure consecutively fits new models to provide a more accurate estimate of the response variable. At each iteration, a new weak, base-learner model is trained on a bootstrapped version of the original features, taking into account the error of the whole ensemble learnt so far. Differently from *Random Forest*, the trees are not independent, but rather, they are grown sequentially, using information of previous iteration. More specifically, each new model is fit on the residuals (i.e. estimates of experimental error) of the old model. These two are then ensembled together and the procedure is repeated until the error is below a certain threshold.
  Among the available algorithms, we selected the widely used *LightGBM* [7], because of its faster training speed and higher efficiency.

For all the three classifiers, the best-working configuration of hyperparameters has been identified through a Grid Search, as shown in the following section.

TABLE II
GRID SEARCH PARAMETERS

Model	**Parameters**	**Values**
MultinomialNB	α	**1x10^{-10}**, 1x10^{-5}, 1
LinearSVC	penalty	**l1**, l2
	loss	hinge, **squared_hinge**
	tol	0.0001, 0.001, **0.01**, 1
	C	**1**, 10, 20
	class_weight	None, **balanced**
	max_iter	1000, **2000**
LightGBM	boosting_type	gbdt, rf, **dart**
	num_leaves	16, 32, 64, 128, **256**, 512
	min_child_samples	2, **5**, 10, 20, 30
	class_weight	None, is_unbalance, **balanced**
	n_estimators	500, **1000**
	max_bin	500, **1000**
	objective	**binary**

TABLE III
F1 MACRO SCORES FOR DIFFERENT MODELS AND PREPROCESSING STEPS

	MLB		TF-DF	
	private	public	private	public
MultinomialNB	0.5731	0.570	0.561	0.558
LinearSVC	0.604	0.610	0.598	0.594
LigthGBM	0.670	0.681	0.674	**0.686**

C. Hyperparameter tuning

For tuning our models, a 5-fold Cross Validation Grid Search is run to obtain the optimal hyperparameter configuration for each model. The performance are assessed based on the resulting macro f_1 score. The search space of the grid search is summarized in table II. As we can see, one of the main drawbacks of *LightGBM* is the complex and vast choice of parameters to be tuned. The most important ones turned out to be {*boosting_type*, *num_leaves*}.

III. RESULTS

The **best configuration** of the hyperparameters is the one in boldface type in table II. We tested our model both on the private validation set, the one obtained from the 75/25 train/test split, and on the public test set, the one represented by the *evaluation dataset*. The performance are computed in terms of macro f_1 score and they are reported in table 3. For the competition the baseline is 0.611. This was likely obtained with a *Random Forest Classifier*, considering only the numerical attributes, without addressing the handling of the *artists* column.

As we can see, the model which far outperforms the baseline is the *LightGBM*: it reaches a macro $f1$ score of 0.686 and

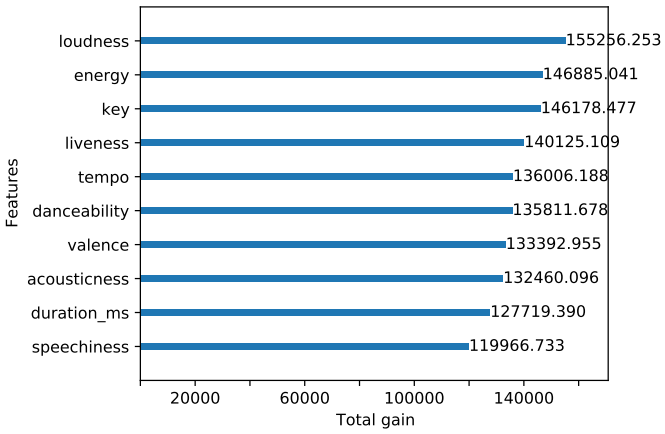

Fig. 3. Feature importance - Gain (top 10)

bring us among the top 5 results of the leaderboard. As expected, a naïve solution as *Multinomial NB* struggles to reach even the baseline, while the *LinearSVC* reaches just the *Random Forest* performance, but with a much higher number of features.

IV. Conclusion

The pipeline built considering *LightGBM*, either with MLB and TF-DF, is able to reach competitive results.

It is worth noticing that, as expected, the choice of the appropriate preprocessing step affects the overall performance. Even if, at first sight, MLB seemed to manage artists more consistently, since it was able to identify and split each name accurately, the handling of artists as textual data through TF-DF vectorizer achieves better results. This could be justified assuming that this last pipeline finds some patterns between same words composing the name of different artists, patterns which were not traceable treating the artist name as a whole. Another explanation may rely on the fact that there are artists with very long names, who are often referred to with slightly different names that do not always match with the labels assigned by the MLB. It is interesting to notice that this argument does not hold for the other two models, which apparently struggle to find these patterns.

In figure 3 we display the 10 most important features for the *LightGBM* with TF-DF model, with respect to the total gain. We can clearly see that the numerical attributes are the most relevant ones. It is also worth mentioning which are, among these, the heaviest attributes in terms of gain to define the song's mode. The mode is somehow more related to the overall loudness, to the perceptual intensity and, just as we expected, to the estimated overall key of the song.

Finally, we are aware that our solution could be improved. Future works could consider:

- exploring the hyperparameters search, both for SVC and GBM. However, it is obvious that an higher computational time must be taken into account. For the GBM running a parallel search or a hyperparameter optimization framework may be useful to find the best combination;
- employing other classification algorithms. During the development of our solution, we briefly tried other methods which have not been reported for brevity's sake. However, methods such as *Ridge Classifier*, *K-Nearest Neighbors* or even *Neural Networks* could be more thoroughly explored;
- reducing the dimensionality of our features, exploiting PCA, since we are dealing with large sparse matrix. This could vastly improve the computation time required and the usefulness of our features;
- explore *Spotipy*, a lightweight Python library for the Spotify Web API, which has several functions to deal with songs and their parameters. However, we are not sure if using this library could void the fairness of our model and violate the competition rules, since it may be considered as an access to an external dataset.

References

[1] X. Wan, "Influence of feature scaling on convergence of gradient iterative algorithm", IOP Conf. Series: Journal of Physics: Conf. Series1213 (2019) 032021

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[3] Xu, Shuo and Li, Yan and Zheng, Wang, Bayesian Multinomial Naïve Bayes Classifier to Text Classification, (2017), pp. 347-352.

[4] Rish, Irina Hellerstein, Joseph Thathachar, Jayram. (2001). An analysis of data characteristics that affect naive Bayes performance.

[5] Adankon M., Cheriet M. (2009) Support Vector Machine. In: Li S.Z., Jain A. (eds) Encyclopedia of Biometrics. Springer, Boston, MA.

[6] Natekin, Alexey and Knoll, Alois, "Gradient boosting machines, a tutorial", Frontiers in Neurorobotics, vol. 7, pp 1-21, 2013.

[7] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in Advances in Neural Information Processing Systems (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, pp. 3146–3154, Curran Associates, Inc., 2017.