University of New Mexico

Project 2: Logistic Regression

CS 529: Introduction to Machine Learning
Professor Trilce Estrada-Piedras

Prasanth Guvvala

Thomas Fisher

10 April 2024

Methodology

Feature Extraction/Transformation

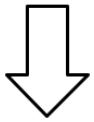
We chose to use all of the features presented in the project assignment as part of our feature extraction pipeline. We also added 2 features beyond those mentioned in the assignment to reduce underfitting. The table below lists all of the features (from the `librosa` library) employed by our feature extraction process.

Features Extracted from Audio Files
`librosa.feature.zero_crossing_rate`
`librosa.feature.mfcc`
`librosa.feature.chroma_stft`
`librosa.feature.chroma_cqt`
`librosa.feature.spectral.spectral_contrast`
`librosa.feature.chroma_cens`
`librosa.feature.tonnetz`

Table 1: Features used in logistic regression feature extraction process.

Because the features we extracted using the functions above are temporal data, we added a transformation on the data. We extracted the mean, variance, min, and max of each feature column returned by the feature call. In this way, we reduced the columns of each file's feature matrix to just four, while still maintaining enough information for reasonable accuracy. This process, which is shown below in Figure 1, was repeated independently for each of the 7 features used per file.

	Feature 1	Feature 2	…	Feature *n*
Coefficient 1				
Coefficient 2				
…				
Coefficient *n*				



	Mean	Variance	Max	Min
Coefficient 1				
Coefficient 2				
…				
Coefficient *n*				

Figure 1: Transformation from feature matrix to statistical values

Each of the feature statistics matrices was then flattened by appending the rows end-to-end. Flattened statistics matrices from different features (mfcc, stft, etc.) were then concatenated to produce our final matrix of unprocessed features.

The unprocessed feature matrix was then passed through a train/test split, standardization by columns, and finally PCA. Normalization was not performed at this step as our softmax function used in gradient descent normalized the values produced from $\mathbf{XW}^T$ products. As mentioned in lecture, standardization and PCA fits were generated using only our training data split, and the fitted transforms were then applied to the testing split.

Gradient Ascent

Our gradient ascent proceeds through an iterative process of updating weights using gradients calculated at each step and then updates the weights and then computes the Loss on the basis of weights learned.
The algorithm tries to run for an optimal number of maximum iterations that we figured out over a series of runs that we ran to converge our gradient ascent. We also tried to set the learning rate dynamically in order to achieve a speedup in convergence.

Our gradient ascent implementation makes use of the following matrix values to train our weights and converge to a loss threshold:

Y_predicted : softmax($X * W^T$), Which represents a 2D matrix with each element representing the probability of that element belonging to a class that is mapped to the column position.

Y_training : It represents the 2D matrix of one-hot-encoded training target samples. Where each row represents a training sample and each column represents the target class.

X : a 2D matrix where each row represents a training sample and each column represents a transformed feature.

W : a 2D matrix where each row represents the set of weights used to identify a target and each column represents weights that approximate a feature in **X**.

η : learning rate

λ : regularization strength

Throughout our gradient ascent we try to not only learn the weights but also compute loss. Our gradient ascent can be divided into two processes:

1) Computing gradients and updating weights

Gradients : Σ **X * (Y_training - Y_predicted)**

where **X and Y** contain all training samples and target values that are one-hot-encoded.

Update rule : $\mathbf{W = W + \eta * (\ gradients\ ) - \lambda * W}$

2) After we update weights the immediate step is to compute loss.

Loss : Σ **(Y_training - Y_predicted)** ^{2}

Convergence Criteria

1) The number of gradients ascent iterations has reached a predefined threshold OR
2) The loss of our model is smaller than a predefined threshold

The loss is computed in each iteration and the gradient ascent proceeds till it finds a minimal loss or number of iterations equals to maximum iterations.
If we can observe, the magnitude of the **Loss** function reduces as the gradient ascent proceeds because we try to find the direction of the gradient that maximizes the likelihood of the correct target given the data.

As a result throughout the process we will end up learning the change in probabilities and update weights with the changes. It can be observed that **Loss** strictly decreases as the gradient ascent proceeds for a suitable learning rate.
The reason we call it gradient ascent is because we are trying to maximize the likelihood by adding up to the weights so maximizing likelihood minimizes the loss. It is the exact reason why we have Loss value being minimum as one of our **Convergence Criteria**. We also tried to add Δ **Loss < Epsilon** to our **Convergence Criteria** but observed our gradient ascent converging before loss was suitably reduced.

Optimizations

- As an optimization, we wrote the entirety of our training and testing matrices to file (.npy format) each time we generated them. This allowed us to bypass feature extraction when testing changes to the logistic regression algorithm alone. Tests of changes to hyperparameters could be conducted quickly as loading training and testing data from file took only a fraction of the time required to perform the entire feature extraction/transformation process.
- All operations performed as part of the logistic regression algorithm were vectorized numpy operations on matrix (np.array) data structures.

Results:

Model Results

We assessed the performance of our model using a variety of metrics. Most directly, we computed the balanced accuracy of our model on our test split to gain an understanding of how well our model classifies unseen data. We used a 70%/30% train test split for our implementation. Table 2 below shows some statistics calculated over 5 runs of the final configuration of our logistic regression algorithm and feature extraction pipeline. We were able to achieve accuracy of over 70%, with a tight range of variation when resetting the random seed

between runs. We achieved our threshold 0.01 loss within less than 4,500 iterations of gradient descent, a process which took under 30 seconds on CS machines.

Average Accuracy	Max Accuracy	Average Recall	Average Iterations
0.7244	0.7519	0.7244	4297.4

*Table 2: Statistics over 5 runs of our logistic regression implementation. Hyperparameters were held constant and only the random seed used for the **W** matrix was changed. Note that recall and accuracy are equal as the data split is even. All accuracies above are balanced accuracies.*

We used scikit-learn's implementation of the confusion matrix to create a visual which illustrates which classes our model struggles to classify. Figure 2 below shows the correct and incorrect classifications per class. We can see that the rock and disco classes were classified most inaccurately, likely due to their similar audio features. Jazz, on the other hand, was classified most accurately, likely due to the genre's distinctive vocal and instrumental qualities.

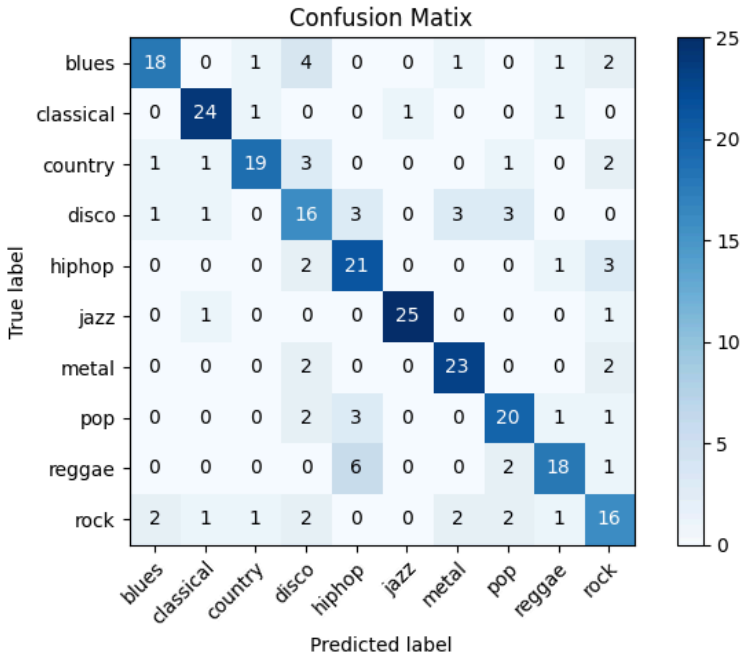


Figure 2: Confused matrix of our logistic regression model.

To investigate how learning rate affects the convergence of our gradient descent, we performed 3 runs with hyperparameters held constant, except for the value of η used. Figure 3 below shows that, up to a point, increasing η decreases the number of iterations required to converge. However, that trend only lasts for so long. As the figure below shows, the decrease in iterations

needed to converge when changing η from 1.0 to 2.0 is much less than that gained from changing η from 0.1 to 1.0.

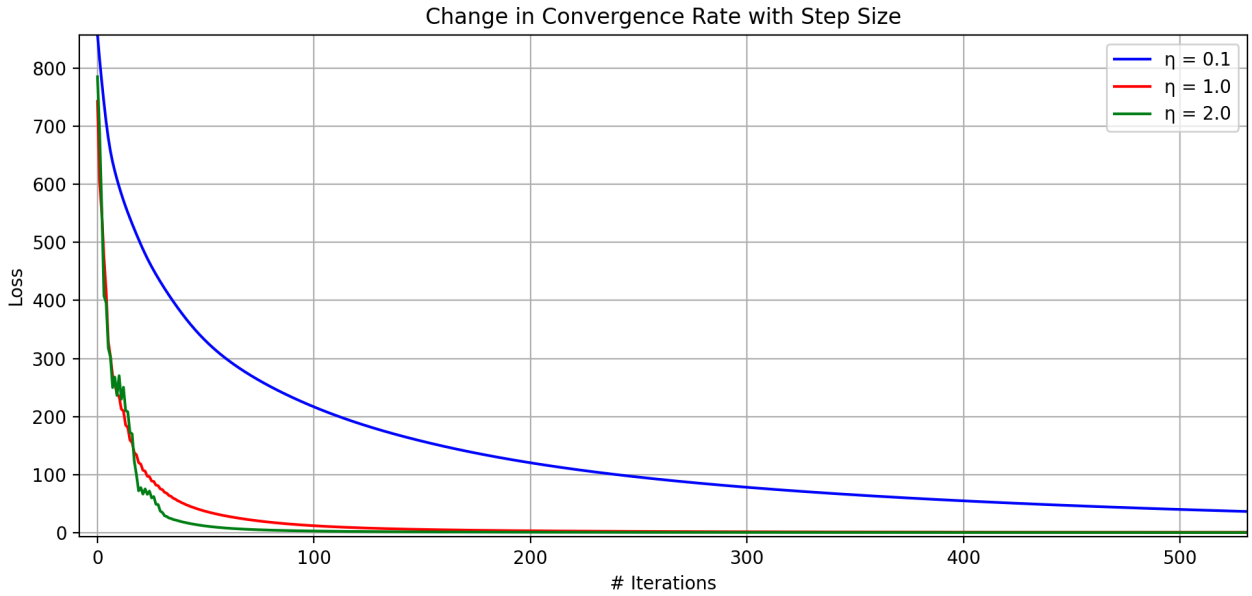


Figure 3: Change in convergence time for different learning rate values.

Comparison of sklearn ML Models

We tested the 4 library machine learning functions mentioned in the assignment using our best data extraction pipeline. We tested a variety of hyperparameters for each function and reported the average accuracy over k-fold cross validation. The code used to generate the results below can be found in function `library_model_hyperparameter_search()` of repository file `training/library_models.py`. The results of our testing is shown below in Table 2:

	Avg. Balanced Acc.	Max Balanced Acc.	# Tested Hyperparameter Sets
Random Forest	0.6704	0.69333	15
SVM	0.6456	0.7422	6
Naive Bayes	0.5974	0.6133	3
Gradient Boosting	0.6350	0.6789	12

Table 2: Comparison of balanced accuracies between sklearn ML functions averaged over the total number of hyperparameter configurations tried.

Random forests seem to perform the best over the average of the tested range of hyperparameters, indicating a superior robustness for classifying the dataset. SVM was able to achieve the highest accuracy for a particular hyperparameter set, which points to this ML model as having the greatest potential out of those tested. This is not altogether unexpected, as we learned in lecture that SVM performs well with high-dimensional data, which this dataset certainly encompasses. Naive bayes and gradient boosting performed worst out of the four. Naive bayes could be struggling due to the lack of provided priors, which we did not provide the library function. Gradient boosting is the method we are least familiar with due to not yet encountering in class.

Conclusions

For this project, we set out to build and use the logistic regression classifier for the purpose of music genre classification. To do so, we wrote logistic regression and gradient ascent implementations from scratch as well as performed feature extraction and transformation on our provided dataset of short duration audio files.

We achieved a good level of accuracy with our model and were able to compare its performance with that of other machine learning library models taken from scikit-learn. Our analysis of the results has increased our understanding of the roles of the hyperparameters involved in our algorithm as well as the relative differences in performance among machine learning classifiers. Our trial and error approach to feature extraction has broadened our understanding of how feature representation affects the convergence rate, memory efficiency, and accuracy of a model.