FYS-2021: Assignment 1

Tor-Ivar Hassfjord

UiT - Norges Arktiske Universitet

08.09.2024

Abstract

This report explores how logistic regression can be applied to classify Spotify songs into either Pop or Classical using stochastic gradient descent (SGD). We focus on two key features: *liveness* and *loudness*. Throughout this report, I attempt to apply what I learned in class, such as using SGD, feature scaling, and evaluating models using metrics like accuracy, precision, recall, and F1 score.

Contents

Introduction

In this assignment, I was asked to classify songs from the Spotify dataset into Pop or Classical genres using logistic regression. I decided to use stochastic gradient descent (SGD) as the optimization algorithm, as this was required by the assignment. I used two features: *liveness* and *loudness*. The model's performance was evaluated using confusion matrices, and several metrics like accuracy, precision, recall, and F1 score. In the sections that follow, I'll walk through the methods, implementation, and results.

The code for this assignment, including all preprocessing, model training, and analysis, can be found in my GitHub repository at the following link: [Hassfjord 2024]

1 Method/Theory

In this section, I want to explain some of the key concepts that I learned in my course and how they apply to the work I did in this report. I'll go through how logistic regression works, why we use stochastic gradient descent (SGD), the importance of feature scaling, and the metrics used to evaluate the model.

1.1 Logistic Regression

Logistic regression is used when we want to classify data into two categories, which is what I am trying to do here with Pop and Classical songs. The idea is to fit a line that best separates the two classes. This line comes from the equation:

$$\text{sigmoid}(z) = \frac{1}{1 + e^{-z}}$$

This sigmoid function outputs a value between 0 and 1, which we interpret as the probability of the data belonging to a certain class (in this case, Pop). Logistic regression is useful here because it's simple and interpretable, but I learned it's only good for problems where the classes are linearly separable, which might not be the case here.

1.2 Stochastic Gradient Descent (SGD)

The assignment asked us to use SGD to train our logistic regression model. SGD is an optimization method that updates the model's weights after each training sample, rather than waiting for the entire dataset (like regular gradient descent). I learned from the course slides that this makes it faster, especially with large datasets, but it's also noisier because it's based on individual samples.

The learning rate is an important part of SGD. It controls how big of a step we take in the direction of the gradient. In my findings, I observed that a learning rate of 0.0001 combined with 15 epochs resulted in the most optimal performance for this logistic regression model. This combination provided a good balance between training time and accuracy.

When I experimented with lower learning rates, such as 0.00001, the model still reached a similar accuracy, but it required significantly more epochs to achieve the same result. This extended training time without any noticeable improvement in performance made using lower learning rates inefficient and not worth the additional time.

1.3 Feature Scaling

I used a custom version of the standard scaler to normalize my features before feeding them into the model. This was important because features like loudness and liveness are on very different scales, and logistic regression assumes that features are scaled similarly. Without scaling, one feature might dominate the others. The 'CustomStandardScaler' I used ensures that the features are centered around 0 and have unit variance, which is a common preprocessing step for models like logistic regression.

1.4 Shuffling the Data

Before splitting the data into training and test sets, I shuffled the entire dataset. This was an important step to make sure the model doesn't learn any biases from the order of the data. I also shuffled the data before training to ensure the model wasn't influenced by any patterns that could exist in the order of the samples.

1.5 Overfitting

One thing we talked about in class is the problem of overfitting. Overfitting happens when the model performs really well on the training data but poorly on the test data, meaning it learned the "noise" in the training data instead of the actual patterns. I saw this happening a bit in my results because the training accuracy was a bit higher than the test accuracy (92.10% vs. 91.85%).

Overfitting is common when there's not enough data, or when the model is too complex, but in my case, logistic regression is a pretty simple model, so maybe it's because my features (loudness and liveness) don't completely capture the difference between Pop and Classical. In the slides, they suggested using cross-validation or adding more features to help with overfitting, so that might be something to try next time.

1.6 Confusion Matrix and Evaluation Metrics

The confusion matrix helps us see where the model is making mistakes. In my case, it looks like the model is better at identifying Pop songs (high true positives) than Classical ones, but it sometimes misclassifies Classical songs as Pop. I used a few different metrics to understand this better:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Accuracy tells us how often the model was right overall, but precision and recall give us more insight into how it handles the two classes. Precision is about how many of the predicted Pop songs were actually Pop, while recall tells us how many of the real Pop songs were correctly identified. The F1 score balances precision and recall.

In my case, I had a recall of 98.47%, meaning the model found most of the Pop songs, but precision was lower at 87.16%, meaning it predicted some Classical songs as Pop. The F1 score of 92.47% shows the model strikes a decent balance between these two.

1.7 Connection to Course Content

Throughout this report, I tried to apply what we learned in class. The use of SGD, understanding overfitting, and using evaluation metrics like precision and recall all connect back to the core machine learning concepts we covered. I learned that machine learning is not just about building models, but also about understanding how and why they work, and how to evaluate them properly. This project helped me see how things like learning rates and feature scaling, which we talked about in the optimization and gradient descent lectures, make a big difference in practice.

2 Implementation

2.1 Task 1: Data Preprocessing

2.1.1 1a) Loading the Data

```python
import pandas as pd
# Load the dataset
data = pd.read_csv('../data/SpotifyFeatures.csv')
```

```python
pop_classical_df = data[data['genre'].isin(['Pop', 'Classical'])].copy()
pop_classical_df.loc[:, 'label'] = pop_classical_df['genre'].apply(lambda x: 1 if x == 'Pop' else
    0)
```

The dataset contains 232725 songs with 18 features. After filtering for Pop and Classical, I ended up with 9386 Pop songs and 9256 Classical songs.

2.1.2 1b) Shuffling the Data

Before splitting the data into training and test sets, I shuffled the entire dataset to prevent any biases from the order of the songs.

```python
import numpy as np
# Shuffle before splitting
np.random.seed(42)
shuffled_indices = np.random.permutation(len(pop_classical_df))
data_shuffled = pop_classical_df.iloc[shuffled_indices]
```

2.1.3 1c) Train-Test Split

I split the data into 80% for training and 20% for testing.

```python
train_ratio = 0.8
# Split the data
train_size = int(train_ratio * len(data_shuffled))
X_train = features_shuffled[:train_size]
y_train = labels_shuffled[:train_size]
X_test = features_shuffled[train_size:]
y_test = labels_shuffled[train_size:]
```

2.1.4 1d) Feature Scaling

To ensure that both features were on the same scale, I used a custom standard scaler.

```python
class CustomStandardScaler:
    def fit(self, X):
        self.mean_ = np.mean(X, axis=0)
        self.std_ = np.std(X, axis=0)
    def transform(self, X):
        return (X - self.mean_) / self.std_
    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)

scaler = CustomStandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

2.2 Task 2: Model Training

2.2.1 2a) Implementing Logistic Regression with SGD

To train the logistic regression model, I used stochastic gradient descent.

```python
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def logistic_sgd(X, y, learning_rate, epochs):
    weights = np.zeros(X.shape[1])
    errors = []

    for epoch in range(epochs):
        for i in range(X.shape[0]):
            z = X[i] @ weights
            h = sigmoid(z)
            gradient = (h - y[i]) * X[i]
            weights -= learning_rate * gradient

```

```python
        y_pred = predict(X, weights)
        error = np.mean(y_pred != y)
        errors.append(error)

    return weights, errors
```

2.2.2 2b) Training the Model

I trained the model using a learning rate of 0.0001 and ran it for 20 epochs.

```python
weights, errors = logistic_sgd(X_train_scaled, y_train, learning_rate=0.0001, epochs=20)
```

2.2.3 2c) Visualizing the Decision Boundary

For the bonus task, I extracted the learned parameters (weights) from the logistic regression model to draw the decision boundary. The plot below shows how the model separates the two classes, Pop and Classical, based on the features *liveness* and *loudness*.

The green line in the plot represents the decision boundary learned by the model. The red and blue dots represent Pop and Classical songs, respectively. Ideally, the decision boundary should divide the Pop songs (red) from the Classical songs (blue) as cleanly as possible. The positioning of this line tells us how well the model is performing.

```python
# Plot the data and linear decision boundary
def decision_boundary(x1, weights, intercept=0):
    w1, w2 = weights[0], weights[1]  # Extract weights for liveness (x1) and loudness (x2)
    return -(w1 / w2) * x1 - (intercept / w2)  # Rearrange to get loudness (x2) in terms of
    liveness (x1)

# Generate liveness values for plotting, ensuring it covers the entire x-axis range
x1_range = np.linspace(X_train_scaled[:, 0].min(), X_train_scaled[:, 0].max(), 100)

# Calculate corresponding loudness values using the decision boundary equation
x2_range = decision_boundary(x1_range, weights)

# Scatter plot of the original data
plt.scatter(X_train_scaled[y_train == 1][:, 0], X_train_scaled[y_train == 1][:, 1], color='red',
    label='Pop', alpha=0.7)
plt.scatter(X_train_scaled[y_train == 0][:, 0], X_train_scaled[y_train == 0][:, 1], color='blue',
     label='Classical', alpha=0.7)

# Plot the decision boundary
plt.plot(x1_range, x2_range, color='green', label='Decision Boundary')

# Set x and y axis limits to make sure the line and data are visible
plt.xlim(X_train_scaled[:, 0].min(), X_train_scaled[:, 0].max())
plt.ylim(X_train_scaled[:, 1].min(), X_train_scaled[:, 1].max())

# Labels and legend
plt.xlabel('Liveness')
plt.ylabel('Loudness')
plt.legend()
plt.title('Decision Boundary from Logistic Regression')
plt.show()
```

This visualization helps in understanding why the classifier performs the way it does. Ideally, most of the Pop songs (red) should fall on one side of the decision boundary and Classical songs (blue) on the other. If there's too much overlap, it suggests that the features chosen for classification may not fully capture the distinction between the genres.

2.3 Task 3: Model Evaluation

2.3.1 3a) Confusion Matrix

After training the model, I computed the confusion matrix to evaluate its performance.

```python
def confusion_matrix(y_true, y_pred):
    TP = np.sum((y_true == 1) & (y_pred == 1))
    TN = np.sum((y_true == 0) & (y_pred == 0))
    FP = np.sum((y_true == 0) & (y_pred == 1))
    FN = np.sum((y_true == 1) & (y_pred == 0))
    return np.array([[TP, FP], [FN, TN]])

conf_matrix = confusion_matrix(y_test, y_test_pred)
print(conf_matrix)
```

3 Results and Discussion

3.1 Task 1 Results: Data Preprocessing

The original dataset contains 232725 songs with 18 features. After filtering for Pop and Classical genres, I ended up with 9386 Pop songs and 9256 Classical songs.

After splitting the data into training and test sets, I applied feature scaling to ensure that both *liveness* and *loudness* had zero mean and unit variance. This step was crucial because logistic regression is sensitive to feature scales, and without scaling, the larger feature (such as *loudness*) would have dominated the model training process (see Ricaud 2024b). Feature scaling allowed both features to contribute equally to the model's learning.

3.2 Task 2 Results: Model Training

3.2.1 Logistic Regression

The logistic regression model was trained using Stochastic Gradient Descent (SGD) with a learning rate of 0.0001 and 15 epochs. I experimented with different learning rates, and I found that this combination produced the best results. At this learning rate, the model converged efficiently without overshooting the optimal solution (see Ricaud 2024c).

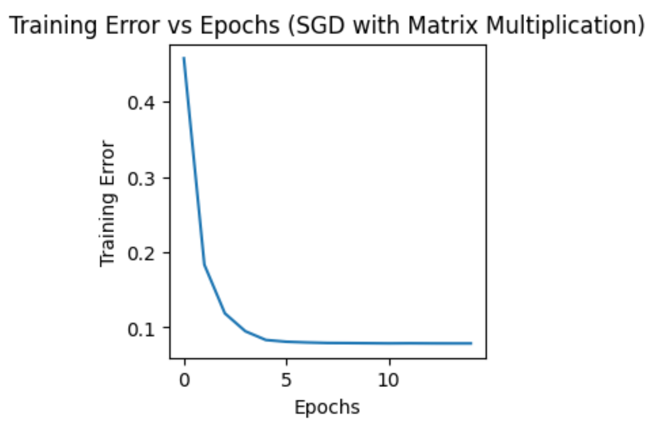

Figure 1: Training Error vs Epochs for Learning Rate 0.0001 and 15 Epochs (SGD with Matrix Multiplication)

As shown in Figure 1, the training error decreases significantly during the early epochs and stabilizes after about 10 epochs. This indicates that the model reaches a good solution quickly, and further training doesn't result in significant improvements.

When I experimented with higher learning rates, such as 0.001, I noticed instability in the early epochs, as shown in Figure 2. The model seemed a bit "jumpy" with a higher learning rate, which I think happened because it was taking too big steps when trying to adjust the weights. This caused it to miss the best values, so the training error didn't go down as smoothly.

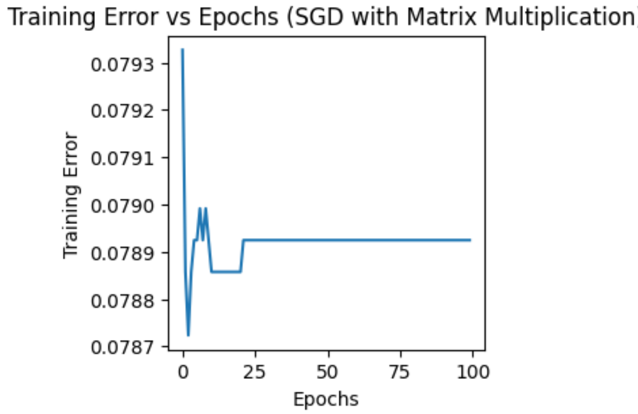

Figure 2: Training Error vs Epochs for Higher Learning Rate (SGD with Matrix Multiplication)

I also experimented with a lower learning rate (0.00001), but I had to increase the number of epochs to 50. As shown in Figure 3, the training error decreased more smoothly over the longer training period. However, it took significantly more epochs to reach a similar level of performance as the 15-epoch case. The model converged, but it required more time to train (see Ricaud 2024c).

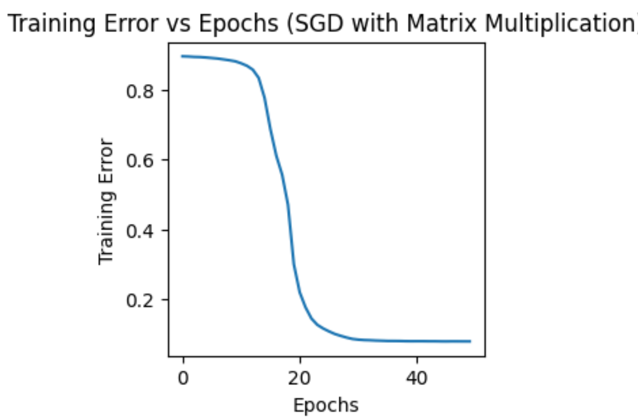

Figure 3: Training Error vs Epochs for Learning Rate 0.0001 with 50 Epochs (SGD with Matrix Multiplication)

Conclusion:
Based on my experiments, a learning rate of 0.0001 with 15 epochs seemed to work best. It allowed the model to train quickly without causing any major issues with accuracy or making the training unstable. When I tried using lower learning rates, the model was more stable, but I had to run it for a lot more epochs to get similar results, which just seemed inefficient. On the other hand, using higher learning rates made the model jump around too much during training, causing the error to fluctuate a lot (see Ricaud 2024c).

3.2.2 Fitting Regression Line

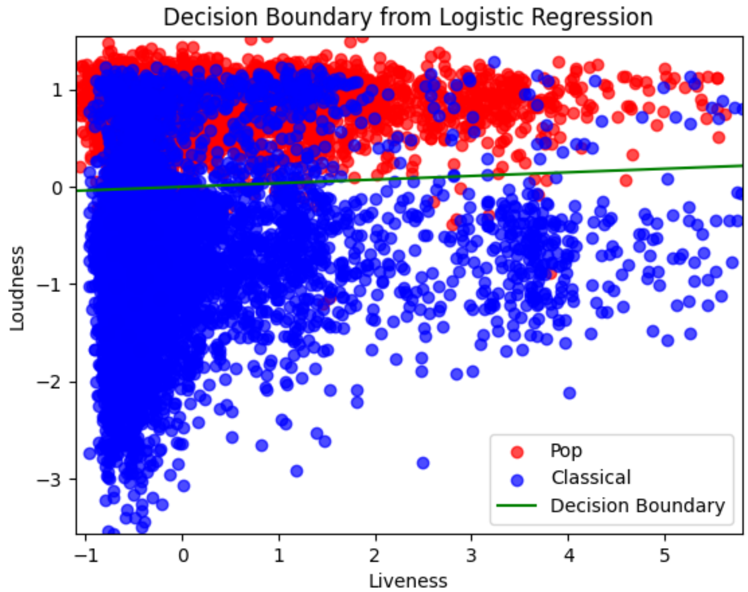

Figure 4: Decision Boundary from Logistic Regression for Pop and Classical Songs

As seen in the plot (Figure 4), the logistic regression classifier tries to draw a linear boundary between the two classes based on the input features. While it performs well for distinguishing a large portion of the songs, it may struggle in areas where there is overlap between the classes. This could explain the slightly lower precision, as some Classical songs may be misclassified as Pop (see Ricaud 2024a).

This plot helps visualize why the model performs the way it does. The separation between Pop and Classical songs isn't perfectly clear-cut, which is likely why the classifier isn't 100% accurate. Adding more features or using a more complex model might help better capture the separation.

3.3 Task 3 Results: Model Evaluation

The confusion matrix for the model on the test data is shown in Figure 5:

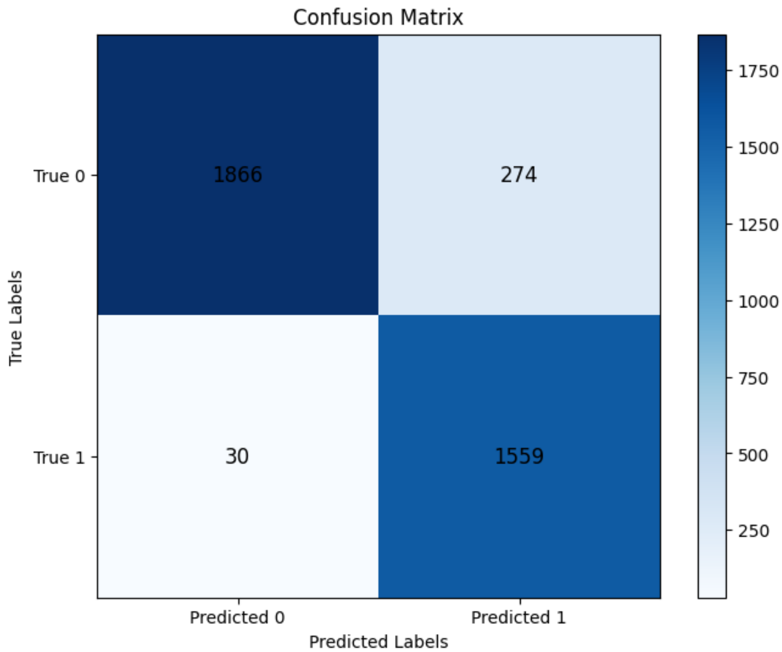

Figure 5: Confusion Matrix of the Logistic Regression Model

$$\begin{bmatrix} 1866 & 274 \\ 30 & 1559 \end{bmatrix}$$

From this matrix, we can observe that the model performs well in identifying Pop songs (high true positives, low false negatives), but it has a slightly higher rate of misclassifying Classical songs as Pop (274 false positives) (see Ricaud 2024a).

3.3.1 Evaluation Metrics

The following evaluation metrics were computed based on the confusion matrix:

- **Training Accuracy**: 92.10%
- **Test Accuracy**: 91.85%
- **Precision**: 87.16%
- **Recall (Sensitivity)**: 98.47%
- **F1 Score**: 92.47%

These metrics show that the model is very effective at identifying Pop songs (high recall), but its precision is slightly lower because of the false positives (misclassified Classical songs). The F1 Score, which balances precision and recall, is still high, indicating that the model performs well overall (see Ricaud 2024a).

3.4 Training vs Test Accuracy: Overfitting Analysis

In class, we discussed overfitting, which happens when a model performs really well on the training data but struggles on new, unseen data (see Ricaud 2024b). In my case, the training accuracy came out to be 92.10%, and the test accuracy was very close at 91.85%. The small difference between these two numbers suggests that the model is not overfitting and is likely generalizing well.

This could be due to the simplicity of logistic regression, which tends to be less prone to overfitting because it has high bias and low variance (see Ricaud 2024b). Another big factor could be the feature scaling I applied. Before scaling, the model's accuracy was around 70%, but after scaling, the accuracy jumped to over 90%. This shows how crucial scaling is for models like logistic regression because it ensures that both *liveness* and *loudness* are treated equally when learning.

Overall, the small gap between training and test accuracy is a good sign that the model is learning general patterns rather than just memorizing the training data (see Ricaud 2024a).

3.5 Impact of Feature Scaling

As discussed in the lecture notes, scaling features is crucial for models like logistic regression because it ensures that all features contribute equally to the model's learning process (see Ricaud 2024b). Without scaling, features with larger ranges, such as *loudness*, would dominate and lead to poor performance. By applying feature scaling to both *liveness* and *loudness*, I ensured that the model treated both features equally during training.

This step had a significant impact on model performance. Before scaling, the model's accuracy hovered around 70%, which suggests that the model struggled to learn a good decision boundary when the features were not normalized. After scaling, the accuracy jumped to over 90%, highlighting the importance of this preprocessing step in improving model performance (see Ricaud 2024a).

3.6 Model Complexity vs Overfitting Tradeoff

In class, we discussed the tradeoff between bias and variance in models (see Ricaud 2024b). Simpler models like logistic regression tend to have high bias and low variance, which can cause underfitting but reduces the risk of overfitting. The close match between my training accuracy (92.10%) and test accuracy (91.85%) suggests the model generalizes well and isn't overfitting (see Ricaud 2024b). However, using only two features (*liveness* and *loudness*) means there's still some risk of underfitting.

Logistic regression with Stochastic Gradient Descent (SGD) minimizes a loss function based on gradient descent principles (see Ricaud 2024c). I found a learning rate of 0.0001 to be optimal, balancing convergence speed and stability. Higher learning rates caused instability, while lower rates required too many epochs (see Ricaud 2024c). In the future, adding features or using more complex models could improve performance, though that comes with the risk of overfitting.

References

Hassfjord, Tor-Ivar (2024). *Machine Learning Coursework - Assignment 1*. GitHub Repository. URL: `https://github.com/tihassfjord/ML2024_tih`.

Ricaud, Benjamin (2024a). *Evaluating Machine Learning Results*. Lecture notes on evaluating machine learning results. URL: `https://uit.instructure.com/courses/34698/modules/items/1058879`.

— (2024b). *Linear Logistic Regression*. Course Lecture Notes. URL: `https://uit.instructure.com/courses/34698/modules/items/1053530`.

— (2024c). *Optimization and Gradient Descent*. Course Lecture Notes. URL: `https://uit.instructure.com/courses/34698/modules/items/1057845`.