FYS-2021: Machine Learning

Assignment 1

Stine Fjeldsrud Karlsen

Github Repository

September 8, 2024

Project Objective

The project objective is to create a machine that will be able to classify and categorize music genre. The dataset given contains several different features that should be used to set the boundaries for the classification. Through machine learning of the a training set of the sample data, the machine should be able to classify the genre from a test set.

Technical Background

Logistic Regression with Stochastic Gradient Descent

Logistic regression is a machine learning algorithm that analyzes the relationship between two factors and determines the accuracy of the predicted value vs. the actual value. It's used for binary classification where the return value 0 or 1 determines which class an instance belongs to. [1]

$$\text{Class}(x) = \begin{cases} \text{Class 1} & \text{if } f(x) > 0.5 \\ \text{Class 0} & \text{if } f(x) \leq 0.5 \end{cases}$$

A gradient descent is an algorithm that descends down a slope and the aim is to find the lowest point. [2] Stochastic Gradient Descent (SGD) chooses a random training example, hence the stochastic addition to the algorithm, instead of the entire data set. It's an iterative algorithm, and is used to optimize learning models by finding the optimum value for each individual data point.[3]

Dealing with the entire data set through logistic regression is computationally expensive. When using a combination of the two algorithms, the training process will only deal with one data point at a time, making the process more efficient and less expensive. It is therefore optimal for bigger data sets.

Log Loss

Log loss is a logarithmic loss function used to evaluate the performance for probabilistic classification models. It calculates the difference between the predicted and actual values of the data points. If the log loss result is 0, the predictions matches the actual outcome perfectly - so the lower log loss, the better the machine is trained.[4]

Confusion Matrix

A confusion matrix summarizes the performance of the test data set and displays the accurate and inaccurate predictions. It is used to evaluate the accuracy and reliability of a machine learning implementation. The following metrics produced by the confusion matrix can be used to calculate for example **accuracy**, **precision** and **recall**:

- **True Positive (TP)**
  Correctly predicted a hit

- **True Negative (TN)**
  Correctly predicted a miss
- **False Positive (FP)**
  Falsely predicted a hit
- **False Negative (FN)**
  Falsely predicted a miss

[5]

Problem 1 - Implementation Preparation

Before we start to train our learning machine, the data set needs to be processed.

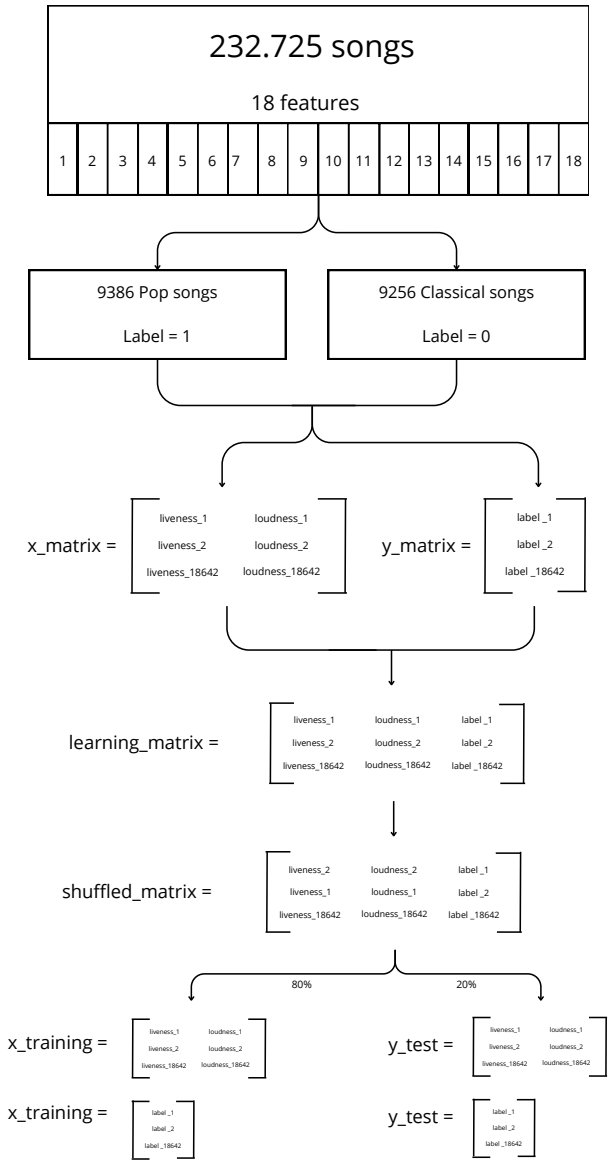

Figure 1: Problem 1 Flow Chart

The entire CSV file given is read and stored in a data set variable. There are a total of 232.725 songs with 18 features. From the data set, all Pop and Classical songs are extracted into separate data sets. There are 9386 pop songs, which are labeled with 1, and 9256 classical songs, which are labeled with 0. In this machine learning model, we are only going to use "Liveness" and "Loudness" as learning features, so when extracting and differentiating Pop and Classical songs, only these features are included in the new separate data sets.

From the separated data sets 2 arrays are created. Matrix X, where the rows are the different songs and the columns are "Loudness" and "Liveness", and a vector Y consisting of the corresponding labels. To then define the training and test sets, matrix X and vector Y is merged in a new matrix, shuffled and then split into training/test sets. The split is divided 80/20, where the training set is a matrix X and vector Y consisting of 80% of the shuffled matrix and the test set is a matrix X and vector Y consisting of 20% of the shuffled matrix.

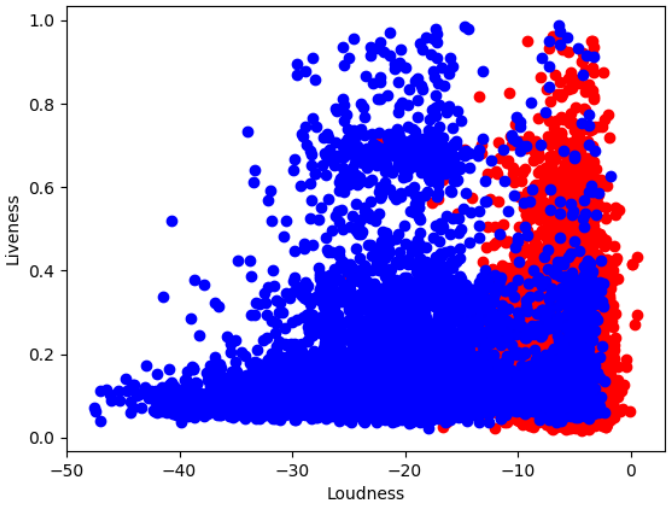

Figure 2: Plot displaying the features "Liveness" and "Loudness" for Pop and Classical songs

From figure 1, we can clearly see that a section of the selection of songs overlap in the right bottom corner. These might be difficult to classify correctly, since the values of our two chosen features are very similar. The rest of the data points seem separated enough to be able to classify correctly. To be able to correctly classify the merging data points, we should look into if there is another feature in the data set which could help achieve a better separation.

Problem 2 - Implementation of Logistic Regression with Stochastic Gradient Descent (SGD)

To start the training process of our machine, a logistic discrimination classifier needs to be implemented. The algorithm used for the classifier is logistic regression with SGD and it should predict whether a song belongs to one of our classes; Pop or Classical.

Bhardwaj [6] explains the mathematical and computational implementation of a logistical regression using SGD from scratch, and is what I will be using as a guide in my own implementation.

Sigmoids function

Sigmoids function is used to predict the value of each data point. The equation of the plane is $z = mx + c$, but since we want to find the plane that seperates our two classes we must apply the **Sigmoid function** to the equation. This will give us a generalized value of m and c.

The mathemathical function:

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

The code implementation:

```
# Define the sigmoid function
def sigmoid(z):
    return 1/ (1+np.exp(-z))
```

Implementation of Gradients

After Sigmoids function predicts the data points value, the gradients values are calculated and updated. These values show how much the predicted value differs from the actual value. By moving in the opposite direction of the loss, we reduce errors.

The mathematical function:

$$dm = x_n(y_n - \sigma((m)^T x_n + c))$$

$$dc = y_n - \sigma((m)^T x_n + c))$$

The code implementation:

```
#Define weight and bias
weight = np.zeros(x_training.shape[1])
bias = 0

#define learning rate and iterations
learning_rate = 0.0001 #changeable
epochs = 100 #changeable

log_loss_list = []

# Loop through all iterations to optain the optimal value of m and c
for iterations in range(epochs):

    for i in range(len(x_training)):

        #compute prediction
        z = np.dot(x_training[i], weight) + bias
        y_pred = sigmoid(z)

        # compute the gradients
        sgd_weight = (y_pred - y_training[i]) * x_training[i]
        sgd_bias = y_pred - y_training[i]

        # update weight and bias
        weight -= learning_rate * sgd_weight
        bias -= learning_rate * sgd_bias

    #Calculate log loss by calculating the new prediction
    pred_training = sigmoid(np.dot(x_training, weight) + bias)

    loss = log_loss(y_training, pred_training)
    log_loss_list.append(loss)
```

```
    #pred_training_list.append(1 if pred_training >= 0.5 else 0)

    print(f'Epoch {iterations + 1}/{epochs}, Loss: {loss}')

# Calculate the accuracy of the training test
pred_training_list = np.where(pred_training >= 0.5, 1, 0)
acc_training_set = accuracy(pred_training_list, y_training)
```

Implementation of Log Loss

The mathematical function

$$logloss = \frac{-1}{N}\sum_{i=1}^{N}(y_i\ log(\hat{y}_i) + (1-y_i)log(1-\hat{y}_i))$$

Where N is the number of observations, y_i is the binary outcome and $\hat{y}_i$ is the predicted probability. [4]

The code implementation:

```
    def log_loss(true, prediction):

    #Can clip the prediction to avoid log(0)
    prediction = np.clip(prediction, 1e-15, 1- 1e-15)
    #Returns the mathemathical calculation
    return -np.mean(true * np.log(prediction) + (1 - true) * np.log(1 - prediction))
```

How learning rate affects log loss

The learning rate affects how big of a step each data point takes in the opposite direction of loss to get closer to the actual value. With a bigger learning rate, the log loss actually increases instead of decreasing. This can be caused because with a bigger learning rate the steps might be too big. A smaller learning rate will therefore result in more accurate training results, however at some point the results will be similar enough so the learning rate wont make much of a difference. This is also reflected in the accuracy for each learning rate. From the plots below, we can determine the learning rate of 0.0001 is best suited of these models.

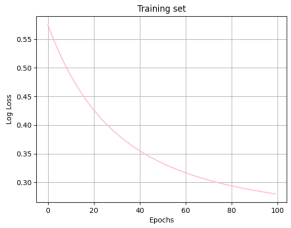

Figure 3: LR: 0.00001.

Accuracy train set: 92.5%
Accuracy test set: 92%

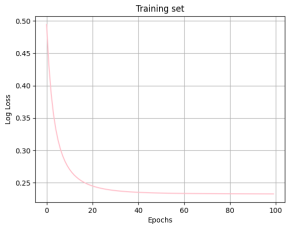

Figure 4: LR: 0.0001

Accuracy train set: 92.6%
Accuracy test set: 92.9%

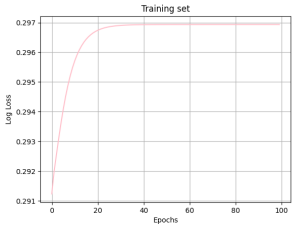

Figure 5: LR: 0.01

Accuracy train set: 90.6%
Accuracy test set: 89.3%

Problem 3 - Evaluation and results

To understand how well the machine is trained, we can test it with a **confusion matrix**. The matrix is computed by comparing the actual labels and the predicted labels.
From this confusion matrix we can extract the following data:

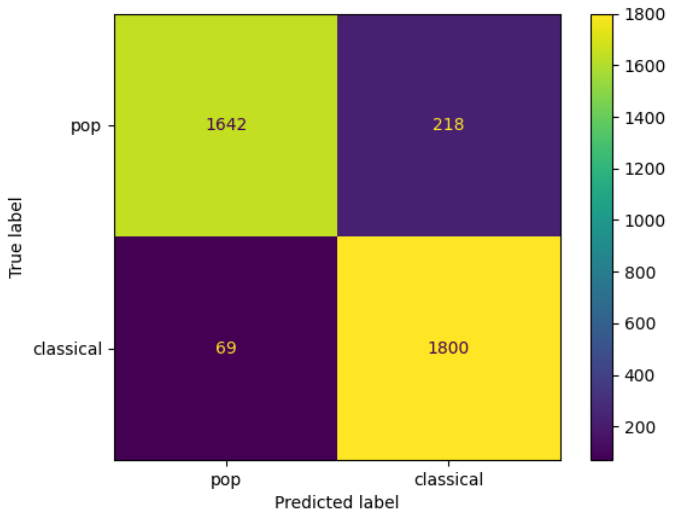

Figure 6: Confusion matrix of the test data set

- **True Negative** (Top-Left Quadrant)
  1642 instances of a correctly prediction of a negative result
- **False Positive** (Top-Right Quadrant)
  218 instances of a falsely prediction of a positive result
- **False Negative** (Bottom-Left Quadrant)
  69 instances of a falsely prediction of a negative result
- **True Positive** (Bottom-Left Quadrant)
  1800 instances of a correctly prediction of a positive result

Accuracy vs. Confusion Matrix

When calculating the accuracy of the training model, there are two values needed to calculate telling result of how well the training model works. In a training model where one class is underrepresented, the accuracy score might seem good, but since the data set is unbalanced it can give us a false hope.

$$Accuracy\ score = \frac{Correct\ Answers}{All\ answers}$$

A confusion matrix gives us a better insight into how the model works, as it highlights where the model makes mistakes. We can also compute an accuracy score based on the results from the confusion matrix:

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

In this model, by calculating the accuracy from the confusionn matrix, we get the following result:

$$Accuracy\ score = \frac{1800 + 1642}{1642 + 218 + 69 + 1800} = \frac{3442}{3729} = 92.3\%$$

Which is quite accurate to the other calculation of accuracy, and is most likely because both classes in our model have a quite similar size. [7]

Conclusion

The training and test data set has quite a high accuracy score, in both calculations. It could be directly connected to the two classes having a similar set size, or a significant difference in the features, but that would need to be tested further to conclude.

After looking at the data calculated from the training model, we can assume this implementation is quite successful.

References

[1] GeeksforGeeks. (n.d.). Understanding logistic regression. GeeksforGeeks. Retrieved September 2, 2024, from https://www.geeksforgeeks.org/understanding-logistic-regression/

[2] Steinbach, M. (2019, April 10). Stochastic gradient descent clearly explained. Towards Data Science. Retrieved September 7, 2024, from https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31

[3] GeeksforGeeks. (n.d.). ML — Stochastic gradient descent (SGD). GeeksforGeeks. Retrieved September 5, 2024, from https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/#stochastic-gradient-descent-sgd

[4] TheDataScience-ProF. (2020, July 6). Understanding log loss: A comprehensive guide with code examples. Medium. Retrieved September 5, 2024 from https://medium.com/@TheDataScience-ProF/understanding-log-loss-a-comprehensive-guide-with-code-examples-c79cf5411426

[5] GeeksforGeeks. (n.d.). Confusion matrix in machine learning. GeeksforGeeks. Retrieved September 6, 2024, from https://www.geeksforgeeks.org/confusion-matrix-machine-learning/

[6] Bhardwaj, A. (2020, February 28). Implementing logistic regression with SGD from scratch. Medium. Retrieved September 2, 2024, from https://medium.com/analytics-vidhya/implementing-logistic-regression-with-sgd-from-scratch-5e46c1c54c35

[7] Hyperskill. (n.d.). Accuracy. Hyperskill. Retrieved September 8, 2024, from https://hyperskill.org/learn/step/11245#accuracy