UiT The Arctic University of Norway

Faculty of Science and Technology

Assignment 1

Logistic Regression

Victor Zimmer

FYS-2021 Machine Learning, Autumn 2024

Introduction

In this assignment we will read data from a CSV file and train a computer on it. The data consists of Spotify tracks and our goal is to use the liveness and loudness features of the tracks to classify them in the genres “Pop” and “Classical”. To do this we will use logistic regression.

The assignment consists of preprocessing data, implementing a logistic discrimination classifier, training the classifier, and evaluating the resulting model.

Problem 1: Preprocessing the data

The data needed for logistic regression is separated into inputs with features as columns and samples as rows, and labels for their expected outputs with a single column containing the label for each sample along the rows.

Problem 1a

To load the dataset we use the Pandas library and its method for reading CSV files

```
# Load dataset
SpotifyDataset = pd.read_csv("data/SpotifyFeatures.csv")
```

Problem 1b

We want to extract only the rows in the dataset belonging to the genres “Pop” and “Classical”. The dataset contains 9386 songs in the Pop genre and 9256 songs in the Classical genre. We also want to reduce the dataset to only contain the features we’re interested in (“liveness” & “loudness”) as well as a label which should be 1 for Pop songs and 0 for Classical songs.

```
# Separate Pop and Classical into datasets
PopDataset = SpotifyDataset.loc[SpotifyDataset["genre"] == "Pop"]
ClassicalDataset = SpotifyDataset.loc[SpotifyDataset["genre"] == "Classical"]

# Set label column in both datasets
PopDataset = PopDataset.assign(label=1.0)
ClassicalDataset = ClassicalDataset.assign(label=0.0)

# Drop all unused columns from both datasets
neededColumns = ["label", "liveness", "loudness"]
PopDataset = PopDataset.drop([x for x in list(PopDataset.columns) if (x not in
neededColumns)], axis=1)
ClassicalDataset = ClassicalDataset.drop([x for x in list(ClassicalDataset.columns)
if (x not in neededColumns)], axis=1)
```

Problem 1c

Finally to get the data ready for training and testing we’ll convert them to NumPy arrays and split them 80% / 20% between training and testing sets respectively, such that we end up with four final arrays, x values for test and train, and y values for test and train. In the x values features will be columns and samples will be rows. The y values have a single column, the label, and one row for each sample.

```
# Calculate split point for 80/20 distribution of both classes
splitPointPop = int(len(PopDataset) * 0.8)
splitPointClassical = int(len(ClassicalDataset) * 0.8)

# Create train and test datasets by concatenating pop and classical based on split
point
TrainDataset = pd.concat([PopDataset.iloc[0:splitPointPop],
```

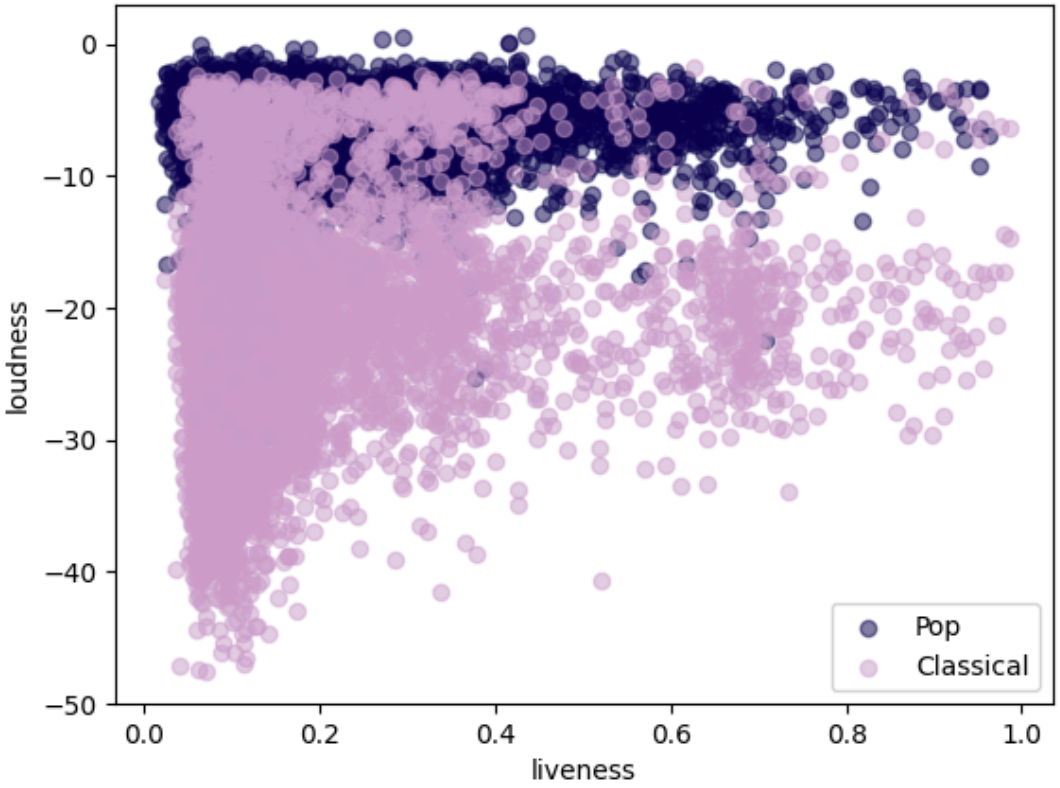
```
ClassicalDataset.iloc[0:splitPointClassical]])
TestDataset = pd.concat([PopDataset.iloc[splitPointPop:len(PopDataset)],
ClassicalDataset.iloc[splitPointClassical:len(ClassicalDataset)]])

# Convert training dataset to numpy arrays for X and Y
TrainArrayX = np.array(TrainDataset.iloc[:, 0:2])
TrainArrayY = np.array(TrainDataset.iloc[:, 2:3])

# Convert testing dataset to numpy arrays for X and Y
TestArrayX = np.array(TestDataset.iloc[:, 0:2])
TestArrayY = np.array(TestDataset.iloc[:, 2:3])
```

Problem 1d

We will plot the liveness vs. loudness of the dataset using a scatterplot, with a different color for each class.



From the plot it is appearent that there is a difference between the classes where a possible decision boundary could be, however it isn't likely to be 100% precise as there are some samples in each class that have overlapping values for both features.

Problem 2: Teaching a machine

Now we get to the actual machine learning and will implement a logistic regression discriminator, which will fit a function to the data to classify samples between Pop and Classical based on the loudness and liveness.

To implement a logistic discrimination classifier we will first need to define it mathematically. Logistic discrimination classifiers are often referred to as logistic regression, even though they are

classifiers, not regressions. We also assume it is sufficient to implement a binary classifier as we only have two classes, Pop and Classical.

Problem 2a

We should implement a logistic discrimination classifier, using stochastic gradient descent for optimization. It should be implemented with the learning rate as a hyperparameter and in a manner where it is possible to report the error as a function of epochs during training.

Our classifier will learn the function $g(\mathbf{x}) \in [0, 1]$, which will return a value estimating the class of the input data $\mathbf{x}$.

Linear Regression

We begin with definitions from linear regression where the goal is to fit a linear function $y = \mathrm{a}x + b$ to best predict the output given the training data. For a single sample it looks like $f(\mathbf{x}) = y = \boldsymbol{\beta}_0 + \mathbf{x}_1\boldsymbol{\beta}_1 + ... + \mathbf{x}_f\boldsymbol{\beta}_f + \varepsilon$ for every f feature in the input. where $\mathbf{x}$ is a vector containing the features of the input and $\boldsymbol{\beta}$ is a vector of the parameters/weights. $\boldsymbol{\beta}_0$ is the bias, corresponding to the b in the linear function. Finally epsilon is included to represent the error.

With a single sample the linear regression prediction can then be calculated as $f(\mathbf{x}) = y = 1\boldsymbol{\beta}_0 + \mathbf{x}_1\boldsymbol{\beta}_1 + ... + \mathbf{x}_f\boldsymbol{\beta}_f + \varepsilon = \boldsymbol{\beta}\mathbf{x}$, assuming the first value of $\mathbf{x}$ is always 1.

For multiple samples linear regression can efficiently be computed using matrices as $f(\mathbf{X}) = \mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$ where every sample becomes $y_i = \mathbf{x}_{\mathrm{i}0}\boldsymbol{\beta}_0 + \mathbf{x}_{\mathrm{i}1}\boldsymbol{\beta}_1 + ... + \mathbf{x}_{\mathrm{if}}\boldsymbol{\beta}_f + \varepsilon_i$ with an assumption that $\mathbf{x}_{\mathrm{i}0} = 1$ for all samples i.

Optimizing linear regression is done by measureing the loss using the mean squared error (MSE), which is simply the mean of all euclidean distances from each point to the line squared. This gives use the loss function $\mathrm{MSE}(\hat{\boldsymbol{y}} = f(\mathbf{x}), \boldsymbol{y}) = \sum (\hat{y}_i - y_i)^2$, and thus fitting the linear regression becomes a problem of minimizing this loss. This has a closed-form solution that is not relevant for this assignment, but suffice to say it would be an easy task to compute.

Binary Logistic Discrimination Classifier

To go from linear regression to a binary logistic discrimination classifier we use the logistic function to change the y values of a linear regression from $[-\infty, \infty]$ to $[0, 1]$, this function is $s(z) = \frac{e^z}{1+e^z}$ and if usually referred to as the sigmoid function.

We apply the sigmoid function to our equation for linear regression $f(x) = \boldsymbol{\beta}^T\mathbf{x}$, to get a function for the prediction using logistic regression $g(x) = \frac{e^{\boldsymbol{\beta}^T\mathbf{x}}}{1+e^{\boldsymbol{\beta}^T\mathbf{x}}}$.

This also extends for multiple samples as $g(\mathbf{X}) = \frac{e^{\mathbf{X}\boldsymbol{\beta}}}{1+e^{\mathbf{X}\boldsymbol{\beta}}}$.

We have omitted the error term, which we do with an assumption that the error has mean value of $\varepsilon = 0$, allowing the bias term to account for the entire mean error.

Optimizing logistic regression, we can no longer use the mean squared error (MSE) loss function from linear regression. This is because MSE assumes a linear relationship between the predicted and true values, but logistic regression is classification where the predicted values are probabilities between 0 and 1. Thus, a more appropriate loss function is needed. For logistic regression, it is common to use the binary cross-entropy or log-loss function as the cost function. This measures the difference between the predicted probabilities and the actual labels. For a binary classifier, the log-loss is given as $L(\beta) = -\frac{1}{n}\sum_{i=1}^{n}(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i))$ where n is the samples.

To find the gradient we begin with the cost function for a single sample $L(\beta)_i = -(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$ and substitute the sigmoid function $s(z)$ for $\hat{y}_i$ for the linear prediction z_i from earlier to get $L(\beta)_i = -\Big(y_i \log\Big(\frac{e^{z_i}}{1+e^{z_i}}\Big) + (1 - y_i) \log\Big(1 - \frac{e^{z_i}}{1+e^{z_i}}\Big)\Big)$.

We then find the derivative of $L(\beta)_i$ with respect to z_i as our parameters are part of the linear predictions $z = f(x) = \boldsymbol{\beta}^T \mathbf{x}$ and therefore $z_i = f(x) = \boldsymbol{\beta}^T \mathrm{x}_i$.

We will need the derivative of the simoid function $s(z) = \frac{e^z}{1+e^z}$ which is $\frac{\mathrm{d}}{\mathrm{d}z} s(z) = s(z)(1 - s(z))$.

Then the derivative of $L(\beta)_i$ with respect to z_i is $\frac{\partial L(\beta)_i}{\partial z_i} = \frac{\partial}{\partial z_i} - (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) = \hat{y}_i - y_i$ after using the chain rule.

Then we can find the gradient of $L(\beta)_i$ with respect to our parameters β for each feature parameter β_p. Since $z_i = f(x) = \boldsymbol{\beta}^T \mathrm{x}_i$ we get $\frac{\partial z_i}{\partial \beta_p} = x_{\mathrm{ip}}$.

This vectorizes to the computation we use in the code $\frac{1}{N} \boldsymbol{X}^T (\hat{\boldsymbol{y}} - \boldsymbol{y})$ where N is the number of samples.

Stochastic Gradient Descent vs Gradient Descent

This implementation should use stochastic gradient descent, as opposed to gradient descent. The difference lies in the samples used to compute the gradient, in the case of gradient descent the entire dataset is used, whilst stochastic gradient descent uses random samples from the dataset re-selected every iteration.

Stochastic gradient descent can therefore deal with much larger datasets, whilst gradient descent will usually give a smoother path to the minima.

Implementation in Python

The implementation has a class *LogisticDiscriminationClassifier* that holds the logic for learning parameters and making predictions. It assumes the training loop is handled outside the class, to enable reporting per-epoch data.

```
class BinaryLogisticDiscriminationClassifier:
    def __init__(self, featureCount, learningRate):

    def learn_step(self, X, y):

    def predict(self, X):
```

Sigmoid helper function

We will need to use the sigmoid function multiple times, so to avoid writing it out every time we implement a helper function for computing it. The implementation is based on the function $s(z) = \frac{e^z}{1+e^z}$ from above. For more efficient computation we avoid computing the exponential e^z twice.

```
# Simoid helper function
def sigmoid(z):
    expZ = np.exp(z)
    return expZ / (1 + expZ)
```

Prediction

First we implement the prediction as it follows from our $g(\mathbf{X})$ above. We also need to make sure our returned values follow the expectation of being either 0 or 1, by returning 1 for values above 0.5 and 0 otherwise.

```
def predict(self, X):
    # Add column of ones to input for bias term
```

```
    X = np.hstack([np.ones((len(X), 1)), X])

    # Compute linear prediction(s)
    linear_prediction = np.dot(X, self.parameters)

    # Apply sigmoid function to get logistic prediction(s)
    logistic_prediction = sigmoid(linear_prediction)

    # Return class predictions, either 1 or 0
    return [1 if y > 0.5 else 0 for y in logistic_prediction]
```

Learning

Learning the logistic regression is done by computing the gradient for the loss with respect to the input as previously mentioned. Then the parameters are adjusted slightly, based on the learning rate, in that direction. This is repeated for any number of steps requested. In this implementation the learning function will carry out a single step, relying on an external training loop to direct it.

Stochastic Gradient Descent

Both gradient descent and stochastic gradient descent (SGD) are implemented, with SGD being the default. It also default to using 10 samples for SGD, with the option of changing the value.

```
# Gradient descent
parameter_gradient = (1/len(X)) * np.dot(X.T, (logistic_prediction - y))

# Stochastic Gradient Descent
stochasticSelection = random.choice(range(len(X)-self.stochasticSelectionSize))
parameter_gradient = (1/self.stochasticSelectionSize) *
np.dot(X[stochasticSelection:stochasticSelection+self.stochasticSelectionSize].T,
(logistic_prediction[stochasticSelection:stochasticSelection+self.stochasticSelectionSize]
- y[stochasticSelection:stochasticSelection+self.stochasticSelectionSize]))
```

This makes the content of the learning method

```
def learn_step(self, X, y):
    # Add column of ones to input for bias term
    X = np.hstack([np.ones((len(X), 1)), X])

    # Compute linear prediction(s)
    linear_prediction = np.dot(X, self.parameters)

    # Apply sigmoid function to get logistic prediction(s)
    logistic_prediction = sigmoid(linear_prediction)

    # Apply gradient descent, would use one/a few samples if SGD
    parameter_gradient = (1/len(X)) * np.dot(X.T, (logistic_prediction - y))

    self.parameters = self.parameters - (self.learningRate * parameter_gradient)
```

Training the classifier

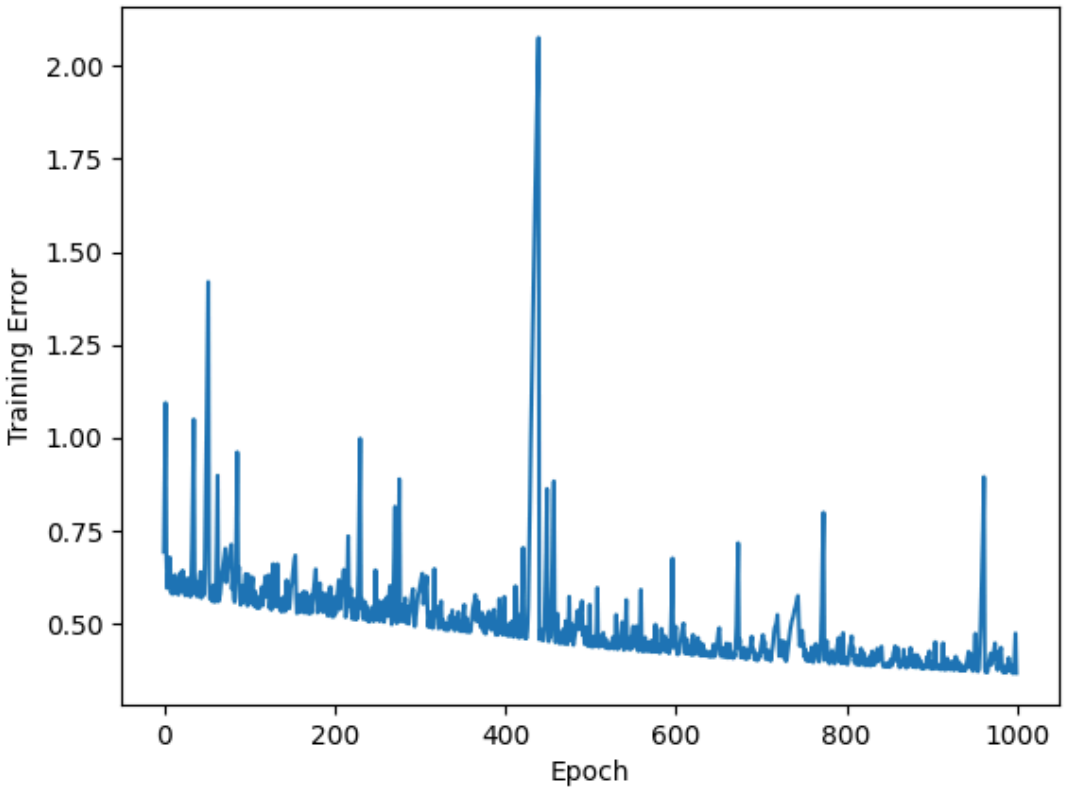
To train the classifier we need to run a loop, optimizing stepwise every turn of the loop. Usually we refer to every loop over the dataset as an epoch. The training loop for this classifier is relatively simple. We call on the *learn_step* method with the training dataset, then we ask for a prediction and calculate its accuracy on the test dataset.

```
bldc = BinaryLogisticDiscriminationClassifier(learningRate=0.01, featureCount=2)
for epoch in range(0,500):
    bldc.learn_step(TrainArrayX, TrainArrayY)
```

```
    pred = bldc.predict(TestArrayX)
    accuracy = np.sum((pred == TestArrayY)) / len(TestArrayY)

    print(f"Epoch {epoch}: accuracy {accuracy}")
```

The training loss seems to be a little nosiy, which might be related to the low amount of samples included in the calculation of SGD every step (10 samples).



Problem 2b

This imlementation results in an accuracy of 0.92 on the training dataset and 0.90 on the test dataset over 1000 epochs with a learning rate set to 0.01, avreaged over 10 runs. There is some randomness introduced by the stochastic gradient descent. It seems to be a small enough difference that the training set and test set are reversed in terms of accuracy on some runs. This points to the risk over overfitting being low, as the performance is almost as good, identical, or sometimes even better on the test dataset.

Problem 2c

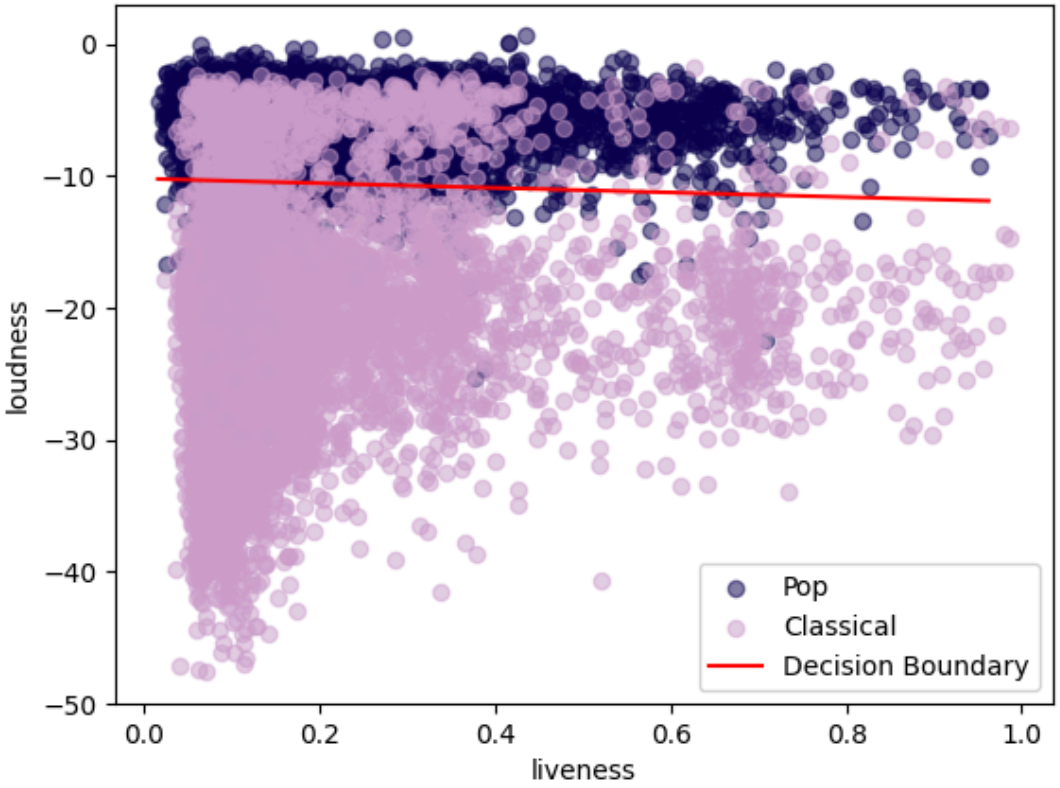
We will plot the linear line separating our data based on the parameters, this is the decision boundary.

```
# Extract learned parameters from the model [bias, liveness, loudness]
bias, liveness, loudness = bldc.parameters

# Find range of existing plotted data
print(PopDataset.iloc[:, 0].min(), PopDataset.iloc[:, 0].max())
```

```
# Compute values for the decision boundary
x_values = np.linspace(PopDataset.iloc[:, 0].min(), PopDataset.iloc[:, 0].max(), 100)
y_values = -(bias + liveness * x_values) / loudness

# Plot the decision boundary
plt.plot(x_values, y_values)
```



Problem 3

Problem 3a

We will create a confusion matrix, which is a matrix containing the count of true positives, false positives, true negatives, and false negatives It has the format

$$\begin{pmatrix} \text{truePositives} & \text{falsePositives} \\ \text{falseNegatives} & \text{trueNegatives} \end{pmatrix}$$

Reported by a run of 1000 epochs and a learning rate of 0.01 was the following confusion matrix

$$\begin{pmatrix} 1712 & 252 \\ 166 & 1712 \end{pmatrix}$$

Problem 3b

The confusion matrix gives additional insight by allowign us to see not just how often the model makes a correct or errneous prediction, but also how it fails when its wrong. In the case above it seems to make more false positives than false negatives, it is biased towards Pop music.

This could very useful for example in medical settings where a false positive would lead to more testing, while a false negative would lead to someone sick being sent home. In that case we want to optimize for less false negatives, and more false positives. In other cases it might be the other way around.

Problem 3c

Unfourtunately this implementation drops the remaining columns from the dataset used for training, so locating the song requires searching the original dataset by liveness and loudness. Luckily this produced only a single match for the random choice of false positive, as well as for the random choice of false negative.

The song Waiting Hare by Buckethead is a Classical song that was identified as Pop.

The song New Phone, Who Dis? by Flatbush Zombies is a Pop song that was identified as Classical.

Sources

Machine Learning A First Course for Engineers and Scientists, Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, Thomas B. Schön