Assignment 1
Machine Learning

Erriadi Amine

September 7, 2025

Problem 1

1a

The number of samples and features correspond to the number of rows and columns, respectively.

```
df = pd.read_csv("SpotifyFeatures.csv")
print("Number␣of␣samples␣(rows):", df.shape[0])
print("Number␣of␣features␣(columns):", df.shape[1])
```

This gives us 232,725 for the samples and 18 for the features.

1b

Since we only work with samples whose "genre" is Pop or Classical, we will only keep these.

```
df = df[(df["genre"] == "Pop") | (df["genre"] == "Classical")]
```

We create the binary categories "Pop" and "Classical". The value is 1 or 0 depending on if the sample is from that genre. Then we count how many samples are in each genre.

```
df["Pop"] = (df["genre"] == "Pop").astype(int)
df["Classical"] = (df["genre"] == "Classical").astype(int)

print("Number␣of␣Pop␣samples:", df[(df["Pop"] == 1)].shape[0])
print("Number␣of␣Classical␣samples:", df[(df["Classical"] == 1)
   ].shape[0])
```

We obtain 9,386 for the number of Pop samples and 9,256 for the Classical ones. Since we should be able to separate the two classes by using only the features "loudness" and "liveness", we only keep these with "Pop" and "Classical".

```
df = df[["Pop","Classical","liveness", "loudness"]]
```

1c

We create a vector for our target "y" and a matrix for our features "X".

```
y = df["Pop"].to_numpy()
X = df[["liveness", "loudness"]].to_numpy()
```

We want to split the dataset while keeping the same class distribution (proportion of Pop and Classical songs). We will take 80% of the Pop songs and 80% of the Classical songs.

```
# Find the indices of each genre
pop_index = np.where(y==1)[0]
classical_index = np.where(y==0)[0]

# Index for where we split
split_pop = int(0.8 * len(pop_index))
split_classical = int(0.8 * len(classical_index))
```

We combine the features and the labels of the 80% of the first Pop songs and the 80% of the first Classical songs.

```
# Create the training set
X_train = np.vstack([X[pop_index[:split_pop]], X[
   classical_index[:split_classical]]])
y_train = np.concatenate([y[pop_index[:split_pop]], y[
   classical_index[:split_classical]]])
```

The same here but with the 20% last remaining:

```
# Create the test set
X_test = np.vstack([X[pop_index[split_pop:]], X[classical_index
   [split_classical:]]])
y_test = np.concatenate([y[pop_index[split_pop:]], y[
   classical_index[split_classical:]]])
```

1d

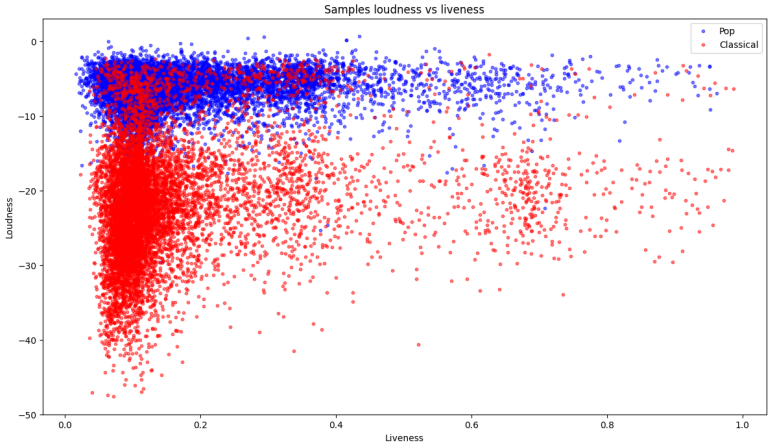

Figure 1: Each sample on the liveness vs loudness plane

From Figure 1, we can deduce that the classification might be globally easy except for the songs between 0 and −10 loudness. The scatters mixe together.

Problem 2

2a

To implement the logistic regression we'll use the sigmoid function to get a probability between 0 and 1:

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

where $z = w \cdot x + b$ with the weights $w \in R^n$ and the bias $b \in R$.

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

The output $\hat{y}$ is a probability: $0 \leq \hat{y} \leq 1$
The prediction rule is:

$$\hat{y} \geq 0.5 \quad \Rightarrow \quad \text{Pop } (1), \quad \hat{y} < 0.5 \quad \Rightarrow \quad \text{Classical } (0).$$

To optimize the parameters w and b, we use gradient descent. For this, we need to compute the partial derivatives of the loss function with respect to the weights and the bias.

The loss function used in logistic regression is the cross-entropy loss:

$$\mathcal{L}(w, b) = -\sum_{i=1}^{N} \Big[y_i \log \hat{y}_i + \big(1 - y_i\big) \log \big(1 - \hat{y}_i\big) \Big],$$

where

$$\hat{y}_i = \sigma(z_i) = \frac{1}{1 + e^{-(w \cdot x_i + b)}}.$$

For a single sample (x_i, y_i), the loss is:

$$\mathcal{L}_i = -\Big[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \Big], \quad \hat{y}_i = \sigma(w \cdot x_i + b).$$

The derivatives of this loss with respect to the weights w and bias b for the same sample are:

$$\frac{\partial \mathcal{L}_i}{\partial w} = (\hat{y}_i - y_i)\, x_i, \qquad \frac{\partial \mathcal{L}_i}{\partial b} = \hat{y}_i - y_i.$$

```
def sample_loss(x, y, w, b):
    y_hat = sigmoid(np.dot(w, x) + b)
    return - (y * np.log(y_hat + 1e-15) + (1 - y) * np.log(1 -
       y_hat + 1e-15))
    # +1e-15 to avoid 0 in the log

def gradient_dw(x, y, w, b):
    y_hat = sigmoid(np.dot(w, x) + b)
    dw = (y_hat - y) * x
    return dw

def gradient_db(x, y, w, b):
    y_hat = sigmoid(np.dot(w, x) + b)
    db = y_hat - y
    return db
```

In our logistic regression, SGD updates the weights w and bias b by considering one sample at a time. For each sample (x_i, y_i), we compute the gradients and update the parameters as

$$w \leftarrow w - \eta \frac{\partial \mathcal{L}_i}{\partial w}, \quad b \leftarrow b - \eta \frac{\partial \mathcal{L}_i}{\partial b}.$$

where η is the learning rate.

```
def logistic_sgd(X,y,epochs=100,lr=0.1):
    n_samples, n_features = X.shape
    w = np.zeros(n_features) # weights
    b = 0 #bias
    losses = []
    for epoch in range(epochs):
        #shuffle
        indices = np.random.permutation(n_samples)
        X_shuffled = X[indices]
        y_shuffled = y[indices]
        epoch_loss = 0

        for i in range(n_samples):
            x_i = X_shuffled[i]
            y_i = y_shuffled[i]

            #compute gradients
            dw = gradient_dw(x_i, y_i, w, b)
            db = gradient_db(x_i, y_i, w, b)

            #update parameters
            w -= lr * dw
            b -= lr * db

            epoch_loss += sample_loss(x_i, y_i, w, b)
        losses.append(epoch_loss / n_samples)
    return w, b, losses
```

We try different learning rates to see how the loss curve behaves:

Figure 2: Training loss for different learning rates

As shown in Figure 2, for our data, the best learning rate is 0.01, while 0.5 struggles to decrease.

2b

To test our model we create a function *predict* that will make the computation easier:

```
def predict(X, w, b):
    y_hat = sigmoid(np.dot(X, w) + b)
    return (y_hat >= 0.5).astype(int)
```

We will train our model and obtain optimized values of w and b, thanks to these we'll compute the accuracy of the model.

```
w, b, losses = logistic_sgd(X_train, y_train, epochs=10, lr
    =0.01)

y_train_pred = predict(X_train, w, b)
train_accuracy = np.mean(y_train_pred == y_train)
print("Training␣set␣accuracy:", train_accuracy)

y_test_pred = predict(X_test, w, b)
test_accuracy = np.mean(y_test_pred == y_test)
print("Test␣set␣accuracy:", test_accuracy)
```

We get for one of the tries that the training set accuracy is 0.93 and the test set accuracy is 0.90. We have a 3% difference. It's not significant enough to worry about.

2c

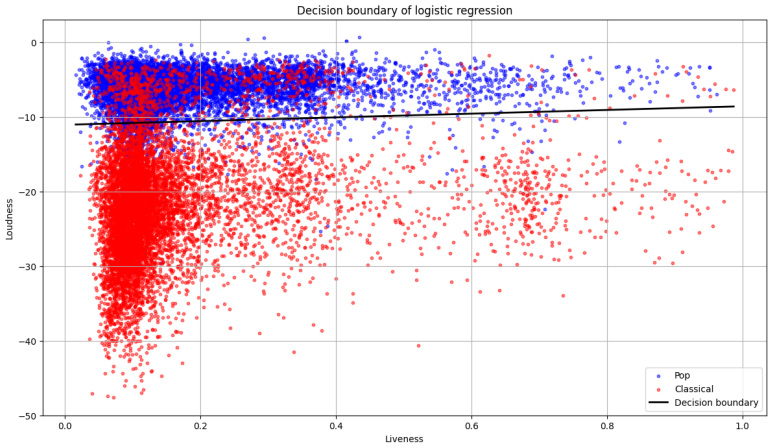

Figure 3: Decision boundary between "Pop" and "Classical"

Figure 3 shows us how the errors happen, it's because some Classical songs lie on the Pop side of the separating line.

Problem 3

3a

To create our confusion matrix we'll count the number of *True Positive*, *False Positive*, *True Negative* and *False Negative* and print it in a matrix.

```
TP = np.sum((y_test == 1) & (y_test_pred == 1)) # True
    Positives
TN = np.sum((y_test == 0) & (y_test_pred == 0)) # True
    Negatives
FP = np.sum((y_test == 0) & (y_test_pred == 1)) # False
    Positives
FN = np.sum((y_test == 1) & (y_test_pred == 0)) # False
    Negatives
```

```
confusion_matrix = np.array([[TP, FN],
                             [FP, TN]])
```

We obtain:

$$\begin{bmatrix} TruePositive & FalseNegative \\ FalsePositive & TrueNegative \end{bmatrix} = \begin{bmatrix} 1804 & 74 \\ 291 & 1561 \end{bmatrix}$$

The model performs better at recognizing Pop songs than Classical songs, with a tendency to misclassify some Classical songs as Pop. This tendency is well explained by Figure 3.

3b

Unlike accuracy, which gives a single overall score, the confusion matrix tells us where our model tends to go wrong. It shows which classes are being correctly predicted and which are being confused, helping us to see if the model tends to misclassify one class more than another.

Sources

SGD: https://www.geeksforgeeks.org/machine-learning/ml-stochastic-gradient-desce