UiT

THE ARCTIC UNIVERSITY OF NORWAY

Assignment 1, Tibo Joris

FYS-2021

classification of classical and pop songs using machine learning.

This page was intentionally left blank.

table of contents

Code

The code can be found on the following link:

- https://github.com/TJ4848/FYS-2021

1a) Loading the dataset.

Code explanation:

```
# 1a) Loading the dataset.
data = pd.read_csv('SpotifyFeatures.csv')

# Display the number of samples and features
num_samples, num_features = data.shape
print("Number of samples:", num_samples)
print("Number of features:", num_features)
```

In this step the data from the csv file containing all the songs is loaded in. Both the number of rows and columns are also displayed to get a grasp on the size of the provided data.

1b) Creating labels.

Code explanation:

```
# 1b) Creating labels.
pop_classical = data[(data['genre'] == 'Pop') | (data['genre'] ==
'Classical')].copy()

# Create labels: Pop = 1, Classical = 0
pop_classical['label'] = pop_classical['genre'].apply(lambda x: 1 if
x == 'Pop' else 0)

# Count the number of samples in each class and display them
pop_count = pop_classical[pop_classical['label'] == 1].shape[0]
classical_count = pop_classical[pop_classical['label'] ==
0].shape[0]
print("Number of Pop samples:", pop_count)
print("Number of Classical samples:", classical_count)
```

Here, the pop songs get a label of 1 whilst the classical songs get a label of 0. These values are stored in a new column named label. This label can then be used to calculate the amount of pop and classical songs and then display them.

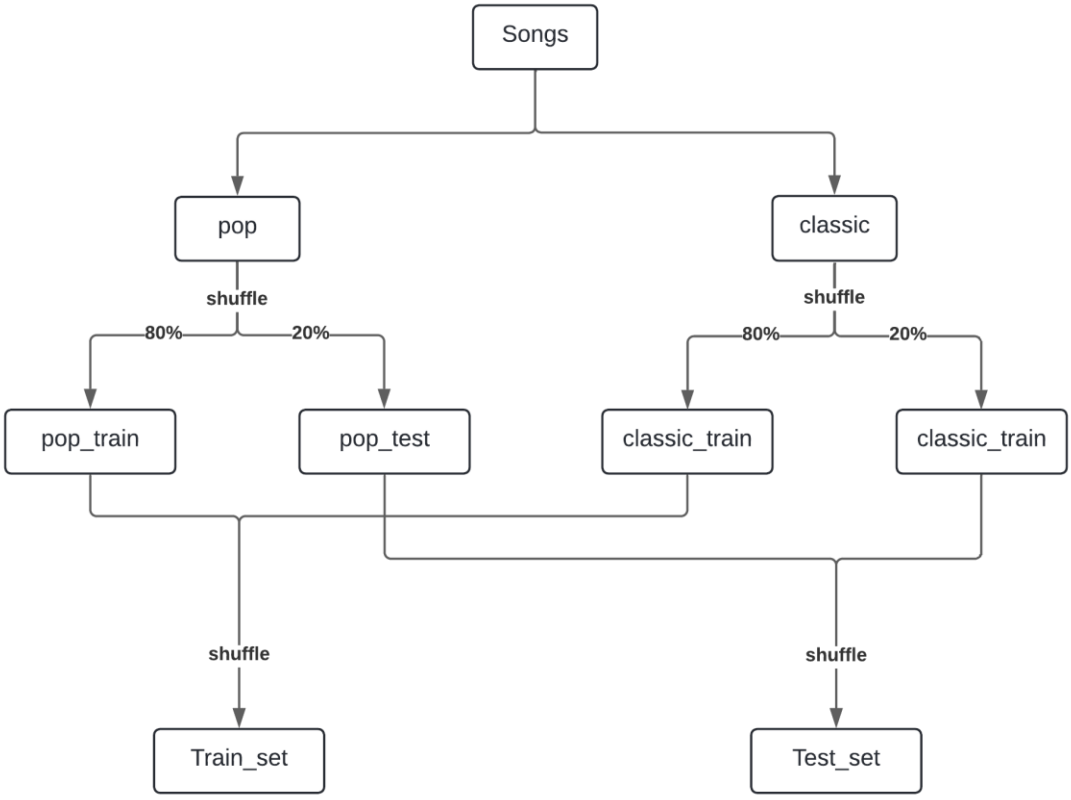
1c) Splitting into test and training sets.

Code explanation:

```
# 1c) splitting into test and training sets.
# Select 'liveness' and 'loudness' as features
X_pop = pop_classical[pop_classical['label'] == 1][['liveness',
'loudness']].values
y_pop = pop_classical[pop_classical['label'] == 1]['label'].values

X_classical = pop_classical[pop_classical['label'] ==
0][['liveness', 'loudness']].values
y_classical = pop_classical[pop_classical['label'] ==
0]['label'].values

# Function to split data into 80% training and 20% test
def train_test_split_manual(X, y, test_size=0.2):
    n = X.shape[0]
    n_train = int(n * (1 - test_size))

    # Shuffle
    indices = np.arange(n)
    np.random.shuffle(indices)

    # Split based on shuffle
    X_train, X_test = X[indices[:n_train]], X[indices[n_train:]]
    y_train, y_test = y[indices[:n_train]], y[indices[n_train:]]

    return X_train, X_test, y_train, y_test

# Split Pop and Classical data separately
X_train_pop, X_test_pop, y_train_pop, y_test_pop =
train_test_split_manual(X_pop, y_pop)
X_train_classical, X_test_classical, y_train_classical,
y_test_classical = train_test_split_manual(X_classical, y_classical)

# Combine Pop and Classical splits to form final training and test
sets
X_train = np.vstack((X_train_pop, X_train_classical))
y_train = np.hstack((y_train_pop, y_train_classical))

X_test = np.vstack((X_test_pop, X_test_classical))
y_test = np.hstack((y_test_pop, y_test_classical))
```

```
# Shuffle the combined training set
train_indices = np.arange(X_train.shape[0])
np.random.shuffle(train_indices)
X_train = X_train[train_indices]
y_train = y_train[train_indices]

# Shuffle the combined test set
test_indices = np.arange(X_test.shape[0])
np.random.shuffle(test_indices)
X_test = X_test[test_indices]
y_test = y_test[test_indices]

# Display the number of samples of the training and test sets
print("Training set size:", X_train.shape[0] , "samples")
print("Test set size:",  X_test.shape[0] , "samples")
```

In this step the data is split into test and training sets according to the following diagram. This way the test and training sets will have the same split between pop and classical songs. The test set will always have the same 20% of the total of applicable songs.

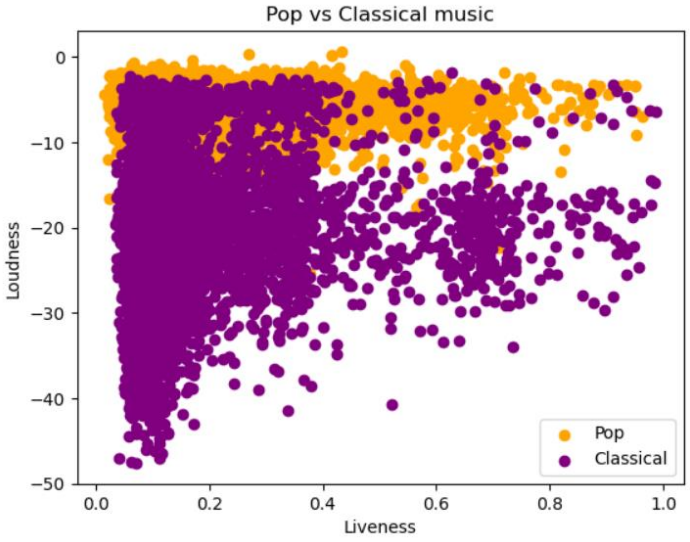


1d) Plotting the data.

Code explanation:

```
# 1d) Plotting the data.
# Plot datapoints for pop and classical
plt.scatter(X_train[y_train == 1][:, 0], X_train[y_train == 1][:,
1], color='orange', label='Pop')
plt.scatter(X_train[y_train == 0][:, 0], X_train[y_train == 0][:,
1], color='purple', label='Classical')

# Generate labels and title
plt.xlabel('Liveness')
plt.ylabel('Loudness')
plt.title('Liveness vs Loudness: Pop vs Classical')
plt.legend()

# Display the plot
plt.show()
```

With this piece of code the data is plotted on a graph to give a visualization of the problem, the graph can be found below.



Easy to classify?, why?

As can be seen on the previously provided graph not all data will be easy to separate due to some of the classical songs having a similar location as the pop songs. A 100% accuracy in separating the two genres will thus never be achievable with only the liveness and loudness features. But a large portion of songs will be able to be separated by a linear separation boundary.

2a) Logistic regression using SGD.

Code explanation:

```
# 2a) Logistic regression using SGD.
class LogisticRegression:
    def __init__(self, learning_rate, epochs):
        # Initialize learning rate and epochs
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = None
        self.bias = None
        self.training_errors = []

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

    def fit(self, X, y):
        n_samples, n_features = X.shape  # Get number of samples and
features
        self.weights = np.zeros(n_features)  # Initialize weights to
zero
        self.bias = 0  # Initialize bias to zero

        # Loop over the epochs
        for epoch in range(self.epochs):
            linear_model = np.dot(X, self.weights) + self.bias
            y_predicted = self.sigmoid(linear_model)

            # Compute the loss
            loss = np.sum((y - y_predicted) ** 2)
            self.training_errors.append(loss)

            # Calculate gradients for weights and bias
            dw = -(2/n_samples) * np.dot(X.T, (y - y_predicted))
            db = -(2/n_samples) * np.sum(y - y_predicted)

            # Update the weights and bias
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db
```

```
        # Plot the training error over epochs
        plt.plot(range(self.epochs), self.training_errors)
        plt.xlabel("Epochs")
        plt.ylabel("Training error")
        plt.title("Training error per Epoch")
        plt.show()

    def predict(self, X):
        # Predict binary labels based on learned weights
        linear_model = np.dot(X, self.weights) + self.bias
        y_predicted = self.sigmoid(linear_model)
        return [1 if i > 0.5 else 0 for i in y_predicted]

    def accuracy(self, y_true, y_pred):
        # Calculate the accuracy by comparing true and predicted
labels
        return np.mean(y_true == y_pred)


# Train the model
model = LogisticRegression(learning_rate=0.01, epochs=1000)
model.fit(X_train, y_train)

# Predict and display training accuracy
y_train_pred = model.predict(X_train)
train_accuracy = model.accuracy(y_train, y_train_pred)
print("Training Accuracy:" , str(round(train_accuracy * 100, 2)) +
"%")
```

This code performs logistic regression using stochastic gradient descent to predict if a sample belongs to a group, in this case it would predict if a certain song belongs to pop or classical music. Using the two provided features (liveness and loudness) this model "draws" a linear separation line between the two groups (pop and classical music) during training, this separation line can then be used as a way of predicting the group during testing and possibly in a later implementation of this model.

Using the following formula the loss is calculated. This formula compares the true labels to the labels the model predicted, if they are vastly different then the loss will be higher. A high loss would indicate that a model isn't performing well, meaning that the model has to go through more iterations (epochs) or needs al lager training sample. Furthermore the loss together with the learning rate are used to update the weights and biases to get a better performing model epoch after epoch.

$$E = \sum_i (y_i - y_{pred})^2$$

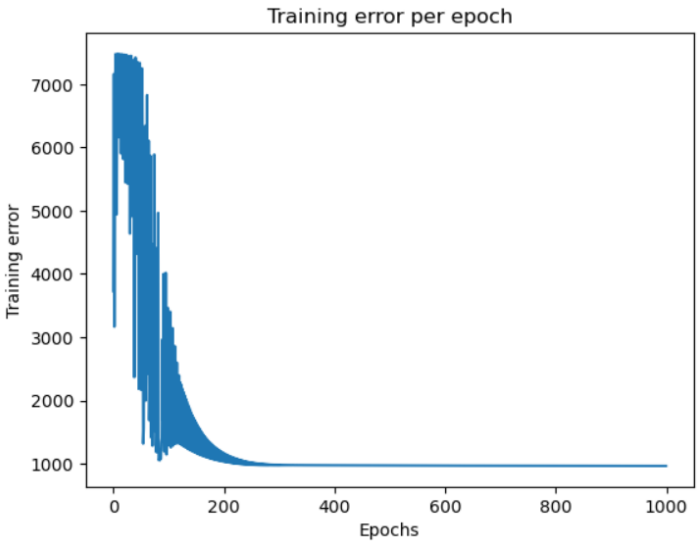
After every epoch the model calculates its training error and plots this on a graph to get a visualization of how the model is progressing. At the end of the training the training accuracy is also calculated by taking the average amount of times the true labels are equal to the predicted labels. The calculated accuracy is a good indicator if the model did well during the training.

Effect of different learning rates?

All of these results are after 1000 epochs.

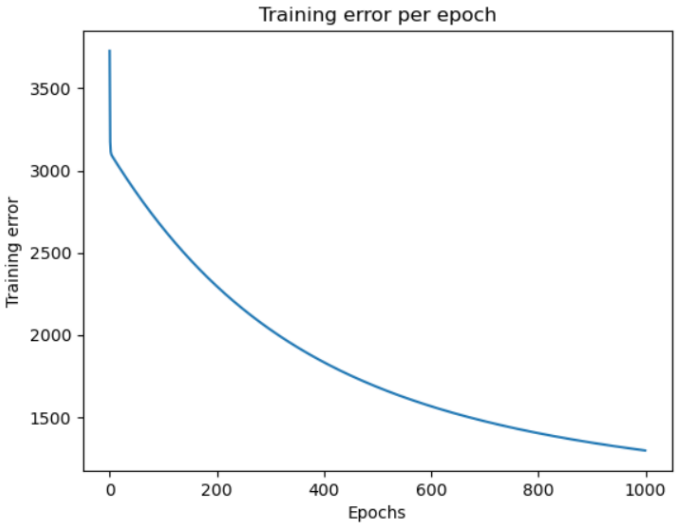
Learning rate	Training accuracy	Testing accuracy	Confusion Matrix
0,1	92,62%	92,84%	[[1660 192] [75 1803]]
0,01	91,91%	92,52%	[[1678 174] [105 1773]]
0,001	58,74%	59,14%	[[1811 41] [1483 395]]
0,0001	49,68%	49,73%	[[1852 0] [1875 3]]

Learning rate 0,1:



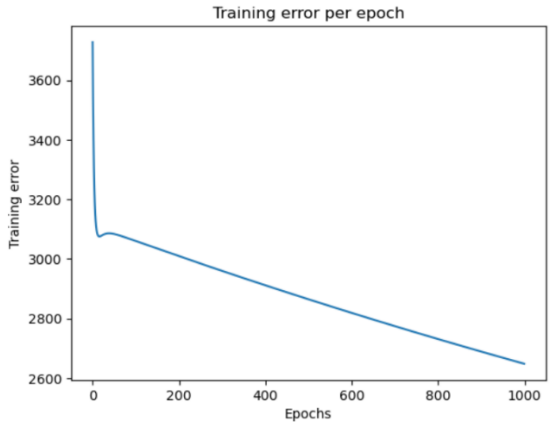
At first the training and testing accuracy as well as the confusion matrix at learning rate 0.1 look great, but if you look at the training error per epoch graph it tells a different story. The giant fluctuations in training error are because the learning rate is to big, this causes the error to have too much "impact" causing those giant fluctuations. The only thing "saving" this model is the fact that is has to go through 1000 epochs giving the model time to sort itself out.

Learning rate 0,01:



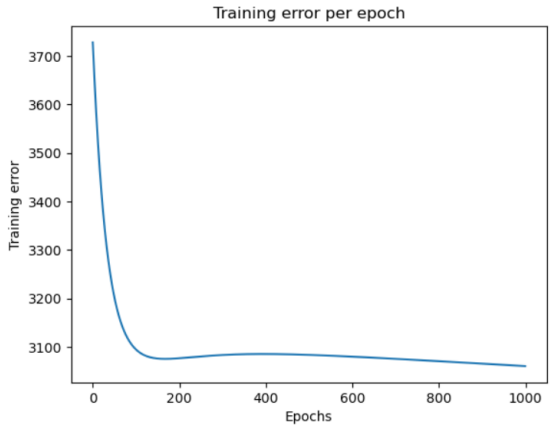
The numbers from learning rate 0,1 and 0,01 look similar, but this time the graph is much cleaner showing gradual improvement per epoch. This indicates the model is working as it should. Together with the great numbers from the accuracies and confusion matrix this learning rate was chosen to use for the rest of the assignment.

Learning rate 0,001:



Compared to learning rate 0,01, learning rate 0,001 is too low because the model was unable to successfully separate the two classes. This is reflected by the poor accuracy numbers of the testing and training data. The confusion matrix also shows that the model especially struggles with classifying pop songs, this is because in this case the decision boundary lies way to "high" on the liveness vs loudness graph. Perhaps this learning rate could be used when given more epochs since the model is still showing improvement after 1000 epochs.

Learning rate 0,0001:



Learning rate 0,0001 is the worst performer by far, this is because it got stuck in a local minima and was unable to "escape" because of the low learning rate. This conclusion was drawn because the model showed almost no improvement between epoch 200 and 1000. This is also can also be seen by the accuracies since they were also the worst amongst all tested learning rates. The only thing this model did well was not misclassify any classic songs as pop, as seen in the confusion matrix. Unfortunately this isn't of any value because of the overwhelming amount of pop songs misclassified as classical songs and the low accuracy scores.

2b) Testing the trained model

Code explanation:

```
# 2b) Testing the trained model.
y_test_pred = model.predict(X_test)
test_accuracy = model.accuracy(y_test, y_test_pred)
print("Test Accuracy: ", str(round(test_accuracy * 100, 2)) + "%")
```

Here, the test data is used to test the model for its accuracy without knowing the label of the song. The predicted and actual label are then compared to see how the model did. This gives an accuracy score which is then printed on the screen. This code can be so short because the comparing and accuracy function of 2a can be reused, the only difference is that we give those functions the test data instead of the training data.

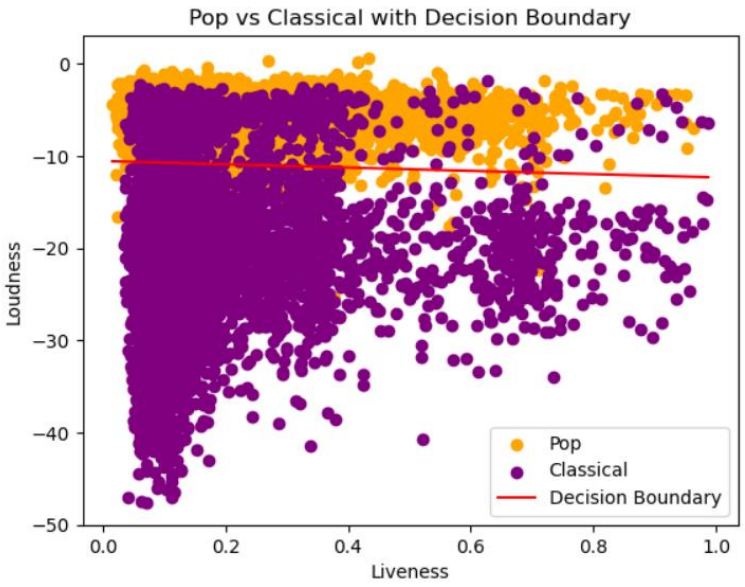
difference in accuracy of training and test set?

Although the accuracy scores of the training and test set are not equal they are very close to each other, at about 92%. This is a higher score than expected because of the overlap between classical and pop songs in the liveness vs loudness graph. This also indicates that the model is performing well and not overfitting the data.

2c) Plotting the decision boundary.

Code explanation:

```
# 2c) Plotting the decision boundary.
# Plot the data points
plt.scatter(X_train[y_train == 1][:, 0], X_train[y_train == 1][:,
1], color='orange', label='Pop')
plt.scatter(X_train[y_train == 0][:, 0], X_train[y_train == 0][:,
1], color='purple', label='Classical')

# Plot the decision boundary
x_values = np.linspace(X_train[:, 0].min(), X_train[:, 0].max(),
100)
y_values = -(model.weights[0] * x_values + model.bias) /
model.weights[1]
plt.plot(x_values, y_values, label='Decision Boundary', color='red')

# Generate labels and title
plt.xlabel('Liveness')
plt.ylabel('Loudness')
plt.title('Liveness vs Loudness: Pop vs Classical with Decision
Boundary')
plt.legend()

# Display the plot
plt.show()
```



The first part of this code functions the same as in 1d, to plot the pop and classical songs. The only difference is that now a decision boundary is added, this line separates the predicted Pop and Classical songs based on the model's learned parameters (weights and bias). Data points falling on one side of the line are classified as Pop while those on the other side are classified as Classical.

3a) creating a confusion matrix.

Code explanation:

```
# 3a) Creating a confusion matrix.
# Initialize confusion matrix
cm = np.zeros((2, 2), dtype=int)

# Get predictions on the test set
y_test_pred = model.predict(X_test)

# Generate the confusion matrix
for true, pred in zip(y_true, y_pred):
    cm[int(true)][int(pred)] += 1

# Print the confusion matrix
print("Confusion Matrix:")
print(cm)
print("with the matrix format being:\n[[classic, classic labled as
pop],\n[pop labled as classic, pop]]")
```

This last piece of code makes the confusion matrixes so that the amount of correctly and wrongly classified songs can be visualized. The confusion matrix even makes a separate category for pop songs mislabeled as classical songs or classical songs mislabeled as pop songs. This has the advantage of clearly visualizing where the model is struggling.

3b) confusion matrix vs accuracy.

Advantages of confusion matrix vs accuracy?

A confusion matrix gives a more detailed evaluation of a model compared to the accuracy score. This is because the confusion matrix gives each class a different category thus the performance of one class is easily readable. Meaning in this case it is apparently visible that classical songs get misclassified as pop more often than pop songs get misclassified as classical. Furthermore the matrix visualizes a difference in class distribution really well thus making it easier to troubleshoot issues compared to the accuracy scores.

3c) difficulty to classify and song recommendations.

Which songs are difficult to classify?

As seen in the confusion matrix the classical songs that have a loudness more than −10 are difficult to classify, this is because of the overlap on the liveness vs loudness graph previously discussed.

Song recommendation:

Although no code was written to recommend classical songs to people who like pop songs here is a recommendation from the list of songs in the cvs file:

- Symphony No. 10 in E Minor, Op. 93: II. Allegro