Automated Music Genre Classification Using Spotify Data

Contents

Problem 1

First the file is read:

```
path = "SpotifyFeatures.csv"
df = pd.read_csv(path, header=0)
```

While the total number of samples is 232725, only 9386 samples belongs to the pop genre and 9256 samples belongs to the classical genre:

```
pop_samples = len(df[df['genre'] == 'Pop'])
classical_samples = len(df[df['genre'] == 'Classical'])
filtered_df = df[(df['genre'] == 'Pop') | (df['genre'] == 'Classical')]
conditions = [ (filtered_df['genre'] == 'Pop'), (filtered_df['genre'] ==
'Classical')] # 1 for pop, 0 for classical
choices = [1, 0] # 1 for pop, 0 for classical
filtered_df['label'] = np.select(conditions, choices)
```

Then the data frame is filtered for the necessary parameters, since there are 18 parameters in total available. Only `'liveness'`, `'loudness'` are of interest to us:

```
parameters = ['liveness', 'loudness']
array_liveness_loudness = filtered_df[parameters]
vector_label = filtered_df['label']
```

Since the ‘loudness’ column isn’t normalized, this is done now. Without this the results would be biased:

```
#normalize the data using min max normalization
array_liveness_loudness['loudness'] = (array_liveness_loudness['loudness'] -
array_liveness_loudness['loudness'].min()) /
(array_liveness_loudness['loudness'].max() -
array_liveness_loudness['loudness'].min())
```

Now the data is split into training and test sets using the sklearn library. The function ensures that the class distribution is maintained during the split:

```
stratifiedsplit = StratifiedShuffleSplit(n_splits=1, test_size=0.2,
random_state=28)
for trainindex, testindex in stratifiedsplit.split(array_liveness_loudness,
vector_label):
    X_train_data, X_test_data = array_liveness_loudness.iloc[trainindex],
array_liveness_loudness.iloc[testindex]
    y_train_labels, y_test_labels = vector_label.iloc[trainindex],
vector_label.iloc[testindex]
```

Now a plot of both classes is created:

```
plt.figure(figsize=(10, 6))

# Plot with distinct colors for each class
scatter = plt.scatter(X_train_data['liveness'], X_train_data['loudness'],
c=y_train_labels, cmap='viridis')

# Adding labels
plt.xlabel('Liveness', fontsize=12)
plt.ylabel('Loudness', fontsize=12)
plt.title('Liveness vs Loudness', fontsize=15)

# Create the legend of classical and pop songs
plt.legend(['Classical', 'Pop'])
# Show plot
plt.show()
```

In Figure 1 a plot of the samples is shown. The pop songs are rather loud, while the majority of classical songs are rather not so lively. Still, the classification isn't easy, since many dots of pop and classical songs do overlap.

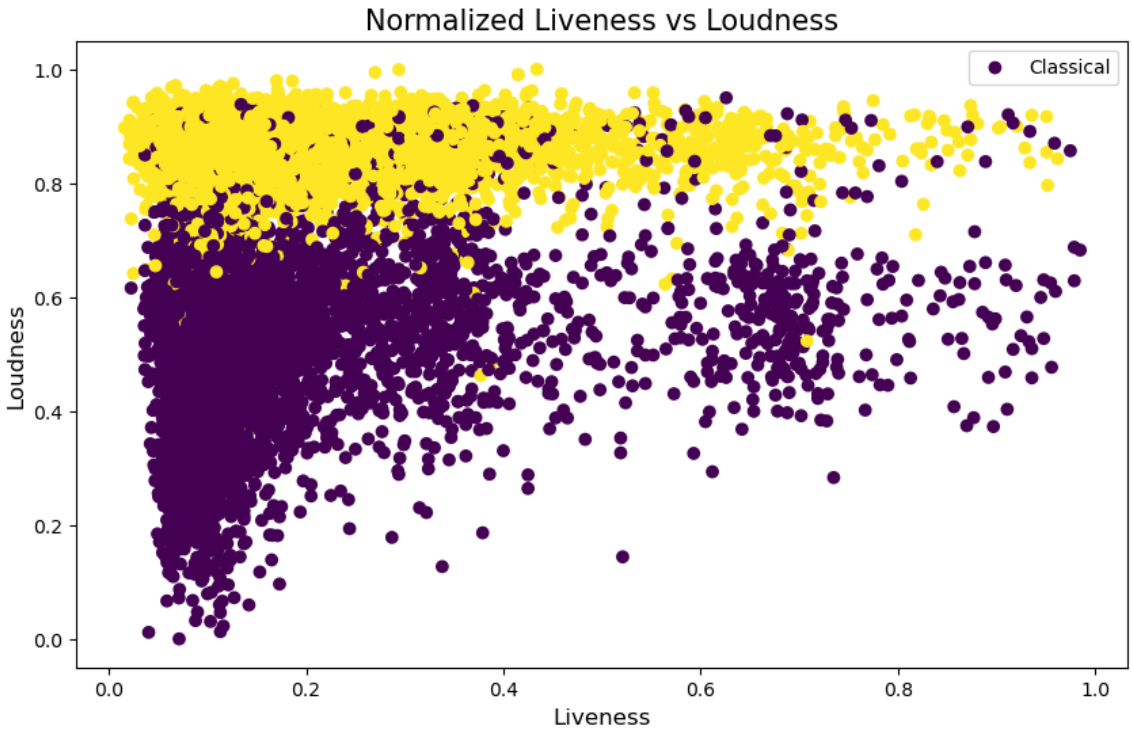


Figure 1 Plot of the samples

Problem 2 and 3

First the stochastic gradient descent function is implemented.

```
def stochastic_gradient_descent(X_train_data, y_train_labels, X_test_data,
y_test_labels, learning_rate=0.01, maximum_epochs=1000, batch_size=32,
tolerance=1e-3):
    number_of_samples, number_of_features = X_train_data.shape
```

We initialize the weights and bias as zeros (or alternatively random values). Training_errors and validation_errors is initialized to later plot the errors vs epochs.

```
    weights = np.zeros(number_of_features)
    bias = 0
    training_errors = []
    validation_errors = []
```

For each epoch $e \in [1,\ldots,E]$ it is necessary to repeat the following calculations. We shuffle the data to prevent the model from learning the order of the data.

```
    for epoch in range(maximum_epochs):
        shuffled_index = np.random.permutation(number_of_samples)
        X_train_shuffled = X_train_data[shuffled_index]
        y_labels_shuffled = y_train_labels[shuffled_index]
```

Having done this, we iterate over the data in batches for every ⟨x[i],y[i]⟩∈D:

```
        for start in range(0, number_of_samples, batch_size):
            end = start + batch_size
            X_batch = X_train_shuffled[start:end]
            y_batch = y_labels_shuffled[start:end]
```

Next we compute the prediction $y\hat{}[i]:=h(\mathbf{x}[i])$ and we compute the errors to later be able to calculate loss $L[i]:=L(y\hat{}[i],y[i])$:

```
            z = np.dot(X_batch, weights) + bias
            y_prediction = sigmoid(z)
            error = y_prediction - y_batch
            epoch_error_sum += np.sum(error**2)
```

Now that the error has been calculated, we compute the gradient weights and bias:

```
            gradient_weights = np.dot(X_batch.T, error) / X_batch.shape[0]
            gradient_bias = np.mean(error)
```

After this step we update the parameters $w:=w+\Delta w, b:=+\Delta b$:

```
            weights = weights - learning_rate * gradient_weights
            bias = bias - learning_rate * gradient_bias
```

Once the second loop ends, the epoch error is calculated and appended for later plotting:

```
        epoch_error = epoch_error_sum / number_of_samples
        training_errors.append(epoch_error)
```

Lastly the convergence is checked, so as not to compute longer than necessary:

```
        if np.linalg.norm(gradient_weights) < tolerance:
            break
    return weights, bias
```

The sigmoid function is defined separately:

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

To plot the data for various learning rates, a special function is defined:

```
def solve_for_learning_rate_and_create_subplot(X_train_data, y_train_labels,
X_test_data, y_test_labels, learning_rates):
    fig, axes = plt.subplots(len(learning_rates), 4, figsize=(20, 15))

    for i, lr in enumerate(learning_rates):
        # measure the time taken to train the model
        time_start = time.time()
        weights, bias, training_errors, validation_errors =
stochastic_gradient_descent(X_train_data.values, y_train_labels.values,
X_test_data=X_test_data.values, y_test_labels=y_test_labels.values,
learning_rate=lr, maximum_epochs=1000, batch_size=32, tolerance=1e-3)

        y_pred_train = sigmoid(np.dot(X_train_data.values, weights) + bias)
```

```
        y_pred_test = sigmoid(np.dot(X_test_data.values, weights) + bias)
        y_pred_train = np.where(y_pred_train > 0.5, 1, 0)  # convert the
prediction to 0 or 1
        y_pred_test = np.where(y_pred_test > 0.5, 1, 0)  # convert the
prediction to 0 or 1

        train_accuracy = accuracy_score(y_train_labels, y_pred_train)
        test_accuracy = accuracy_score(y_test_labels, y_pred_test)

        confusion = confusion_matrix(y_test_labels, y_pred_test)
        time_end = time.time()
        time_taken = time_end - time_start

        print(f"Confusion matrix for learning rate {lr} is:
\n{confusion}\nTrain accuracy: {train_accuracy}\nTest accuracy:
{test_accuracy}\nTime taken is {round(time_taken, 2)} seconds.\n--------------
---")

        # Suggesting Classical songs
        suggested_classical_songs = filtered_df[(filtered_df['label'] ==
0)].sample(1) # Random suggestions, can be made more sophisticated.
        print("Suggested Classical song for Pop fans: \n",
suggested_classical_songs[['track_name', 'artist_name']])



        # Plotting the decision boundary
        axes[i, 0].scatter(X_train_data['liveness'], X_train_data['loudness'],
c=y_train_labels, cmap='viridis')
        axes[i, 0].set_xlabel('Liveness', fontsize=12)
        axes[i, 0].set_ylabel('Loudness', fontsize=12)
        axes[i, 0].set_title(f'Normalized Liveness vs Loudness (LR={lr})',
fontsize=12)
        x = np.linspace(0, 1, 100)
        y = -(weights[0] * x + bias) / weights[1]
        axes[i, 0].plot(x, y, '-r', label='Decision boundary')
        axes[i, 0].legend(loc='upper left')

        # Plotting the training error vs epochs
        axes[i, 1].plot(training_errors, label=f'Learning rate {lr}')
        axes[i, 1].set_xlabel('Epochs')
        axes[i, 1].set_ylabel('Training Error')
        axes[i, 1].set_title(f'Training Error vs Epochs (LR={lr})',
fontsize=12)
        axes[i, 1].legend()

        # Plotting the validation error vs epochs
        axes[i, 2].plot(validation_errors, label=f'Learning rate {lr}')
        axes[i, 2].set_xlabel('Epochs')
```

```
        axes[i, 2].set_ylabel('Validation Error')
        axes[i, 2].set_title(f'Validation Error vs Epochs (LR={lr})',
fontsize=12)
        axes[i, 2].legend()

        # Plotting the confusion matrix
        cmd = ConfusionMatrixDisplay(confusion_matrix=confusion,
display_labels=[0, 1])
        cmd.plot(ax=axes[i, 3], values_format='d', cmap='viridis')
        axes[i, 3].set_title(f'Confusion Matrix (LR={lr})')

    plt.tight_layout()
    plt.show()
```

The resulting plots can be seen in Figure 2. For a learning rate of 0.1 the training error vs epochs is steeper then in the other cases. There are more true positives then in all other plots. The learning rate 0.1 is better suited then the other ones, as it has the highest train accuracy of 92.3% and the curve in both error plots vs epochs is steeper. The test accuracy is approximately 0.3% worse to the train accuracy. This discrepancy is minimal, which suggests that overfitting is not an issue.

```
Confusion matrix for learning rate 0.1 is:
[[1643  208]
 [  75 1803]]
Train accuracy: 0.9275129082008986
Test accuracy:  0.9241083400375436
Time taken is 0.62 seconds.
```

Once the learning rate is lowered to 0.01 there are more true negatives then before, while the true positive rate dropped slightly. The test accuracy is approximately 0.4% worse to the train accuracy.

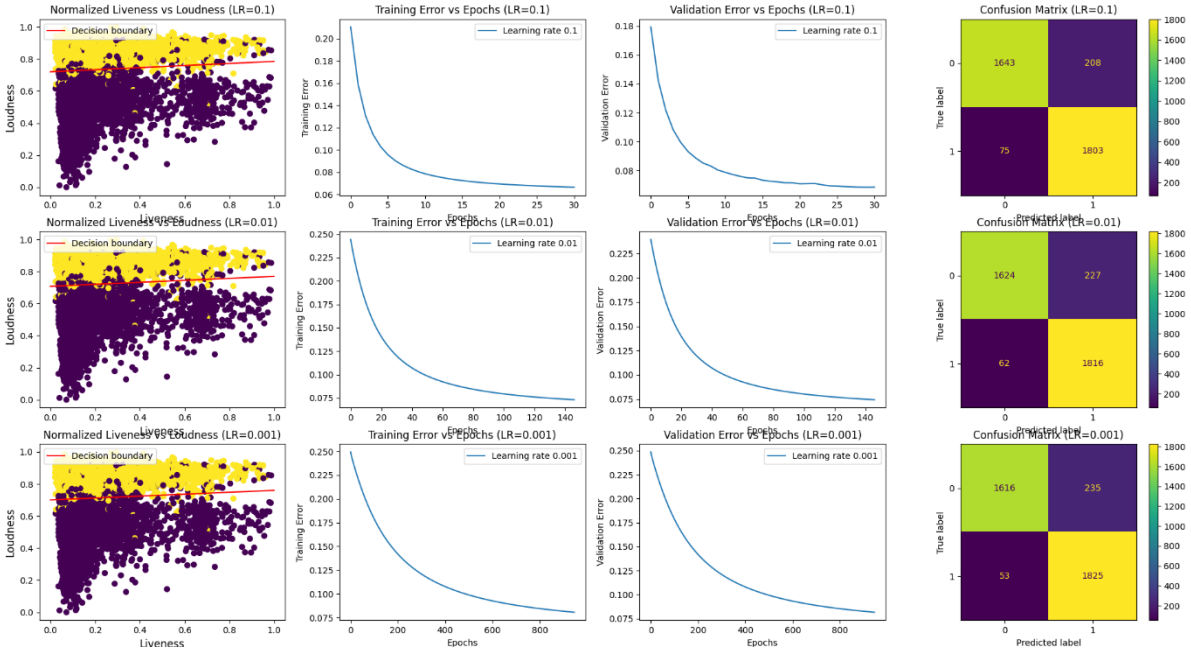
```
Confusion matrix for learning rate 0.01 is:
[[1624  227]
 [  62 1816]]
Train accuracy: 0.9259706296519815
Test accuracy:  0.9224993295789756
Time taken is 2.72 seconds.
```

This same trend is further seen at a learning rate 0.001. Because of the small learning rate it takes much more time to compute then in the other cases. The test accuracy is approximately 0.2% worse to the train accuracy.

```
Confusion matrix for learning rate 0.001 is:
[[1616  235]
 [  53 1825]]
Train accuracy: 0.9245624622812312
Test accuracy:  0.9227674979887369
Time taken is 17.62 seconds.
```

When the data is shuffled, the results remain nearly identical.

The accuracy score shows the proportion of total predictions that are correct, while the confusion matrix provides the exact counts of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Most notably the rate of true negatives is better at 0.01 and 0.001 learning rates with our data. The learning rate could be further be optimized around 0.1 for better results.



a) b) c) d)

Figure 2 Plot of loudness vs liveness decision boundary(a)), training error vs epochs (b)), validation error vs epochs (c)) and the confusion matrices (d)) for different learning rates.

```
One suggested Classical song for Pop fans is The Whispers from Harold Bud:
        # Suggesting Classical songs
        suggested_classical_songs = filtered_df[(filtered_df['label'] ==
0)].sample(1)
        print("Suggested Classical song for Pop fans: \n",
suggested_classical_songs[['track_name', 'artist_name']])
```

Code

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.metrics import accuracy_score, confusion_matrix,
ConfusionMatrixDisplay
from sklearn.metrics import accuracy_score
import time

# 1a
path = "Assignment 1\SpotifyFeatures.csv"
df = pd.read_csv(path, header=0)
print(f"Number of samples (songs): {df.shape[0]}")
print(f"Number of features (song properties): {df.shape[1]}")

# 1b
pop_samples = len(df[df['genre'] == 'Pop'])
classical_samples = len(df[df['genre'] == 'Classical'])
print(pop_samples, "samples belongs to pop genre")
print(classical_samples, "samples belongs to classical genre")

filtered_df = df[(df['genre'] == 'Pop') | (df['genre'] == 'Classical')]
conditions = [
    (filtered_df['genre'] == 'Pop'),
    (filtered_df['genre'] == 'Classical')
]
choices = [1, 0]  # 1 for pop, 0 for classical

filtered_df['label'] = np.select(conditions, choices)

parameters = ['liveness', 'loudness']

# 1c
# Ensure the split maintains the same class distribution.

array_liveness_loudness = filtered_df[parameters].copy()
vector_label = filtered_df['label']
#normalize the data using min max normalization
array_liveness_loudness['loudness'] = (array_liveness_loudness['loudness'] -
array_liveness_loudness['loudness'].min()) /
(array_liveness_loudness['loudness'].max() -
array_liveness_loudness['loudness'].min())
# Split the data
# X_train_data, X_test_data, y_train_labels, y_test_labels =
train_test_split(array_liveness_loudness, vector_label, test_size=0.2,
random_state=28)
```

```
stratifiedsplit = StratifiedShuffleSplit(n_splits=1, test_size=0.2,
random_state=28)
# stratifiedsplit = StratifiedShuffleSplit(n_splits=1, test_size=0.2,
random_state=1)
for trainindex, testindex in stratifiedsplit.split(array_liveness_loudness,
vector_label):
    X_train_data, X_test_data = array_liveness_loudness.iloc[trainindex],
array_liveness_loudness.iloc[testindex]
    y_train_labels, y_test_labels = vector_label.iloc[trainindex],
vector_label.iloc[testindex]
# 1d Plot the samples on the liveness vs loudness plane, with a different
color for each class. From the plot, will the classification be an easy task?
why?
# Plotting the samples on the liveness vs loudness plane, with a different
color for each class
plt.figure(figsize=(10, 6))

# Plot with distinct colors for each class
scatter = plt.scatter(X_train_data['liveness'], X_train_data['loudness'],
c=y_train_labels, cmap='viridis')

# Adding labels
plt.xlabel('Liveness', fontsize=12)
plt.ylabel('Loudness', fontsize=12)
plt.title('Normalized Liveness vs Loudness', fontsize=15)

# Create the legend of classical and pop songs
plt.legend(['Classical', 'Pop'])
# Show plot
plt.show()

# Problem 2

def stochastic_gradient_descent(X_train_data, y_train_labels, X_test_data,
y_test_labels, learning_rate=0.01, maximum_epochs=1000, batch_size=32,
tolerance=1e-3):
    number_of_samples, number_of_features = X_train_data.shape
    # initialize the weights and bias as zeros (or alternatively random
values)
    weights = np.zeros(number_of_features)
    bias = 0
    training_errors = []
    validation_errors = []
    for epoch in range(maximum_epochs):
        # shuffle the data to prevent the model from learning the order of the
data
        shuffled_index = np.random.permutation(number_of_samples)
        X_train_shuffled = X_train_data[shuffled_index]
```

```
        y_labels_shuffled = y_train_labels[shuffled_index]
        epoch_error_sum = 0
        # iterate over the data in batches for every ⟨x[i],y[i]⟩∈D:
        for start in range(0, number_of_samples, batch_size):
            end = start + batch_size
            X_batch = X_train_shuffled[start:end]
            y_batch = y_labels_shuffled[start:end]
            # compute the prediction for the current batch
            z = np.dot(X_batch, weights) + bias
            y_prediction = sigmoid(z)
            error = y_prediction - y_batch
            epoch_error_sum += np.sum(error**2)
            # compute the gradient of the loss with respect to the weights and
bias
            gradient_weights = np.dot(X_batch.T, error) / X_batch.shape[0]
            gradient_bias = np.mean(error)

            # update parameters w:=w+Δw,b:=+Δb
            weights = weights - learning_rate * gradient_weights
            bias = bias - learning_rate * gradient_bias
        epoch_error = epoch_error_sum / number_of_samples
        training_errors.append(epoch_error)

        # Compute the validation error
        z_val = np.dot(X_test_data, weights) + bias
        val_predictions = sigmoid(z_val)
        val_error = val_predictions - y_test_labels
        val_error_sum = np.sum(val_error ** 2)
        val_error_mean = val_error_sum / X_test_data.shape[0]
        validation_errors.append(val_error_mean)

        # check for convergence
        if np.linalg.norm(gradient_weights) < tolerance:
            break
    return weights, bias, training_errors, validation_errors
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def solve_for_learning_rate_and_create_subplot(X_train_data, y_train_labels,
X_test_data, y_test_labels, learning_rates):
    fig, axes = plt.subplots(len(learning_rates), 4, figsize=(20, 15))

    for i, lr in enumerate(learning_rates):
        # measure the time taken to train the model
        time_start = time.time()
        weights, bias, training_errors, validation_errors =
stochastic_gradient_descent(X_train_data.values, y_train_labels.values,
X_test_data=X_test_data.values, y_test_labels=y_test_labels.values,
learning_rate=lr, maximum_epochs=1000, batch_size=32, tolerance=1e-3)
```

```
        y_pred_train = sigmoid(np.dot(X_train_data.values, weights) + bias)
        y_pred_test = sigmoid(np.dot(X_test_data.values, weights) + bias)
        y_pred_train = np.where(y_pred_train > 0.5, 1, 0)  # convert the
prediction to 0 or 1
        y_pred_test = np.where(y_pred_test > 0.5, 1, 0)  # convert the
prediction to 0 or 1

        train_accuracy = accuracy_score(y_train_labels, y_pred_train)
        test_accuracy = accuracy_score(y_test_labels, y_pred_test)

        confusion = confusion_matrix(y_test_labels, y_pred_test)
        time_end = time.time()
        time_taken = time_end - time_start

        print(f"Confusion matrix for learning rate {lr} is:
\n{confusion}\nTrain accuracy: {train_accuracy}\nTest accuracy:
{test_accuracy}\nTime taken is {round(time_taken, 2)} seconds.\n--------------
---")

        # Suggesting Classical songs
        suggested_classical_songs = filtered_df[(filtered_df['label'] ==
0)].sample(1) # Random suggestion
        print("Suggested Classical song for Pop fans: \n",
suggested_classical_songs[['track_name', 'artist_name']])



        # Plotting the decision boundary
        axes[i, 0].scatter(X_train_data['liveness'], X_train_data['loudness'],
c=y_train_labels, cmap='viridis')
        axes[i, 0].set_xlabel('Liveness', fontsize=12)
        axes[i, 0].set_ylabel('Loudness', fontsize=12)
        axes[i, 0].set_title(f'Normalized Liveness vs Loudness (LR={lr})',
fontsize=12)
        x = np.linspace(0, 1, 100)
        y = -(weights[0] * x + bias) / weights[1]
        axes[i, 0].plot(x, y, '-r', label='Decision boundary')
        axes[i, 0].legend(loc='upper left')

        # Plotting the training error vs epochs
        axes[i, 1].plot(training_errors, label=f'Learning rate {lr}')
        axes[i, 1].set_xlabel('Epochs')
        axes[i, 1].set_ylabel('Training Error')
        axes[i, 1].set_title(f'Training Error vs Epochs (LR={lr})',
fontsize=12)
        axes[i, 1].legend()

        # Plotting the validation error vs epochs
```

```
        axes[i, 2].plot(validation_errors, label=f'Learning rate {lr}')
        axes[i, 2].set_xlabel('Epochs')
        axes[i, 2].set_ylabel('Validation Error')
        axes[i, 2].set_title(f'Validation Error vs Epochs (LR={lr})',
fontsize=12)
        axes[i, 2].legend()

        # Plotting the confusion matrix
        cmd = ConfusionMatrixDisplay(confusion_matrix=confusion,
display_labels=[0, 1])
        cmd.plot(ax=axes[i, 3], values_format='d', cmap='viridis')
        axes[i, 3].set_title(f'Confusion Matrix (LR={lr})')

    plt.tight_layout()
    plt.show()
# learning_rates = [0.1, 0.01, 0.001]
learning_rates = [0.1]
solve_for_learning_rate_and_create_subplot(X_train_data, y_train_labels,
X_test_data, y_test_labels, learning_rates)
```