hw2_code

February 1, 2026

```python
[1]: import numpy as np
     import matplotlib.pyplot as plt
```

1 2

```python
[2]: # total 25 samples
     # 16 positive (filled)
     # 9 negative (empty)

     c1_tp = 16
     c1_tn = 0
     c1_fp = 9
     c1_fn = 0

     c2_tp = 16
     c2_tn = 4
     c2_fp = 5
     c2_fn = 0

     c3_tp = 13
     c3_tn = 6
     c3_fp = 3
     c3_fn = 3

     c4_tp = 10
     c4_tn = 7
     c4_fp = 2
     c4_fn = 6

     c5_tp = 4
     c5_tn = 9
     c5_fp = 0
     c5_fn = 12

     c6_tp = 0
     c6_tn = 9
     c6_fp = 0
```

```
c6_fn = 16
```

1.1   a

```
[9]: plt.figure(figsize=(8, 8))
     plt.plot([0, c1_fp/(c1_fp + c1_tn), 1], [0, c1_tp/(c1_tp + c1_fn), 1],␣
      ↪label='Classifier 1')
     plt.plot([0, c2_fp/(c2_fp + c2_tn), 1], [0, c2_tp/(c2_tp + c2_fn), 1],␣
      ↪label='Classifier 2')
     plt.plot([0, c3_fp/(c3_fp + c3_tn), 1], [0, c3_tp/(c3_tp + c3_fn), 1],␣
      ↪label='Classifier 3')
     plt.plot([0, c4_fp/(c4_fp + c4_tn), 1], [0, c4_tp/(c4_tp + c4_fn), 1],␣
      ↪label='Classifier 4')
     plt.plot([0, c5_fp/(c5_fp + c5_tn), 1], [0, c5_tp/(c5_tp + c5_fn), 1],␣
      ↪label='Classifier 5')
     plt.plot([0, c6_fp/(c6_fp + c6_tn), 1], [0, c6_tp/(c6_tp + c6_fn), 1],␣
      ↪label='Classifier 6')
     plt.plot([0, 1], [0, 1], 'k--', label='Random Guess')
     plt.xlabel('False Positive Rate')
     plt.ylabel('True Positive Rate')
     plt.title('ROC Curves for Classifiers')
     plt.legend()
     plt.grid()
     plt.show()
```

ROC Curves for Classifiers
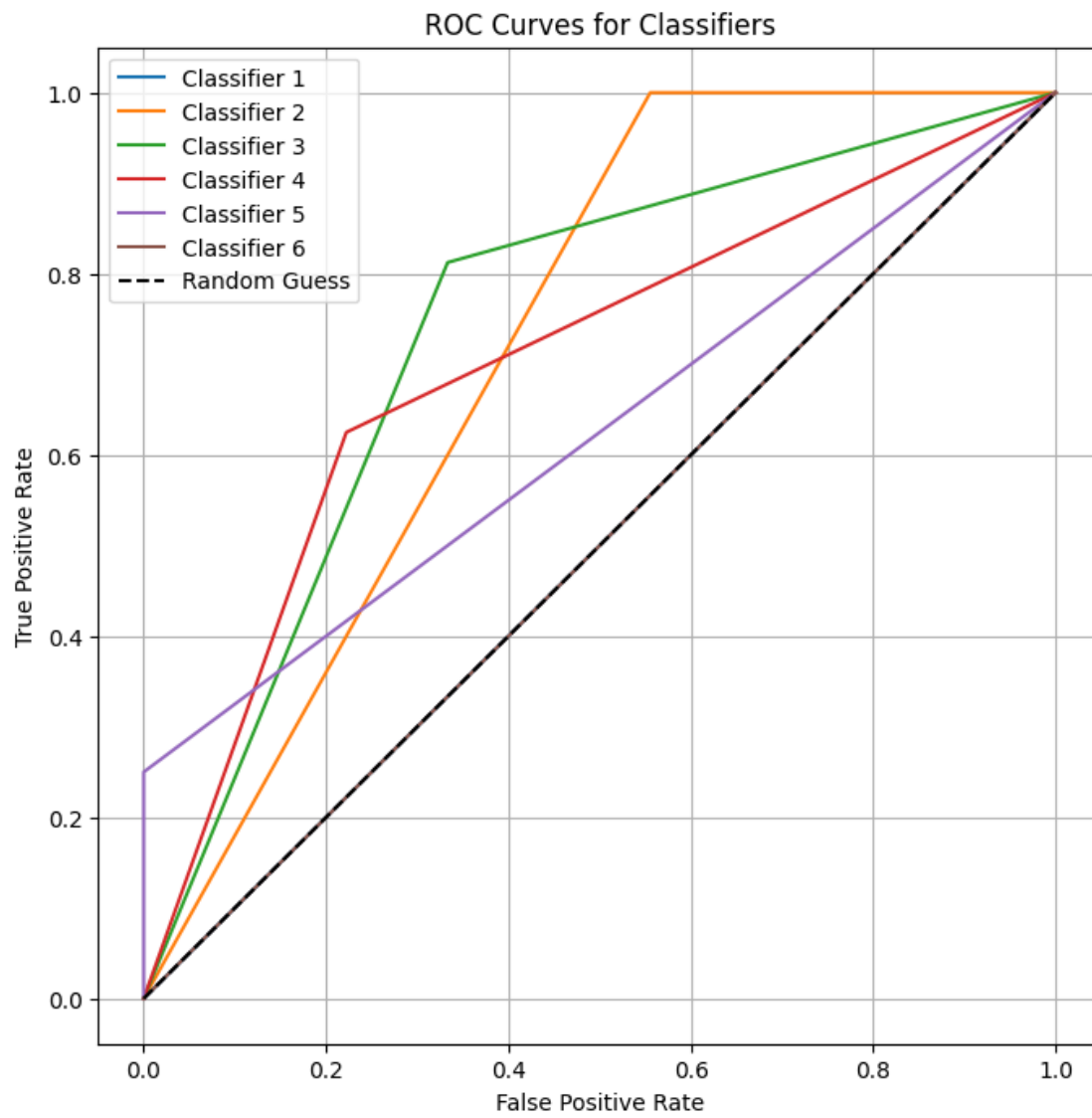
1.2 b

```
[5]: acc_c1 = (c1_tp + c1_tn) / 25
     acc_c2 = (c2_tp + c2_tn) / 25
     acc_c3 = (c3_tp + c3_tn) / 25
     acc_c4 = (c4_tp + c4_tn) / 25
     acc_c5 = (c5_tp + c5_tn) / 25
     acc_c6 = (c6_tp + c6_tn) / 25

     print(f'Classifier 1 Accuracy: {acc_c1:.2f}')
     print(f'Classifier 2 Accuracy: {acc_c2:.2f}')
     print(f'Classifier 3 Accuracy: {acc_c3:.2f}')
```

```
print(f'Classifier 4 Accuracy: {acc_c4:.2f}')
print(f'Classifier 5 Accuracy: {acc_c5:.2f}')
print(f'Classifier 6 Accuracy: {acc_c6:.2f}')
```

```
Classifier 1 Accuracy: 0.64
Classifier 2 Accuracy: 0.80
Classifier 3 Accuracy: 0.76
Classifier 4 Accuracy: 0.68
Classifier 5 Accuracy: 0.52
Classifier 6 Accuracy: 0.36
```

1.3 c

```
[ ]: prec_c1 = c1_tp / (c1_tp + c1_fp)
     prec_c2 = c2_tp / (c2_tp + c2_fp)
     prec_c3 = c3_tp / (c3_tp + c3_fp)
     prec_c4 = c4_tp / (c4_tp + c4_fp)
     prec_c5 = c5_tp / (c5_tp + c5_fp)
     prec_c6 = 0 # undefined

     print(f'Classifier 1 Precision: {prec_c1:.2f}')
     print(f'Classifier 2 Precision: {prec_c2:.2f}')
     print(f'Classifier 3 Precision: {prec_c3:.2f}')
     print(f'Classifier 4 Precision: {prec_c4:.2f}')
     print(f'Classifier 5 Precision: {prec_c5:.2f}')
     print(f'Classifier 6 Precision: Undefined')
```

```
Classifier 1 Precision: 0.64
Classifier 2 Precision: 0.76
Classifier 3 Precision: 0.81
Classifier 4 Precision: 0.83
Classifier 5 Precision: 1.00
Classifier 6 Precision: Undefined
```

1.4 d

```
[7]: f1_c1 = 2 * (prec_c1 * (c1_tp / (c1_tp + c1_fn))) / (prec_c1 + (c1_tp / (c1_tp↵
     ↪+ c1_fn)))
     f1_c2 = 2 * (prec_c2 * (c2_tp / (c2_tp + c2_fn))) / (prec_c2 + (c2_tp / (c2_tp↵
     ↪+ c2_fn)))
     f1_c3 = 2 * (prec_c3 * (c3_tp / (c3_tp + c3_fn))) / (prec_c3 + (c3_tp / (c3_tp↵
     ↪+ c3_fn)))
     f1_c4 = 2 * (prec_c4 * (c4_tp / (c4_tp + c4_fn))) / (prec_c4 + (c4_tp / (c4_tp↵
     ↪+ c4_fn)))
     f1_c5 = 2 * (prec_c5 * (c5_tp / (c5_tp + c5_fn))) / (prec_c5 + (c5_tp / (c5_tp↵
     ↪+ c5_fn)))
     f1_c6 = 0 # undefined since precision is undefined
```

```
print(f'Classifier 1 F1 Score: {f1_c1:.2f}')
print(f'Classifier 2 F1 Score: {f1_c2:.2f}')
print(f'Classifier 3 F1 Score: {f1_c3:.2f}')
print(f'Classifier 4 F1 Score: {f1_c4:.2f}')
print(f'Classifier 5 F1 Score: {f1_c5:.2f}')
print(f'Classifier 6 F1 Score: Undefined')
```

```
Classifier 1 F1 Score: 0.78
Classifier 2 F1 Score: 0.86
Classifier 3 F1 Score: 0.81
Classifier 4 F1 Score: 0.71
Classifier 5 F1 Score: 0.40
Classifier 6 F1 Score: Undefined
```

2 3

```
[3]: data = [
         (2, 2, 3),
         (3, 3, 2),
         (1, 2, 3),
         (1, 4, 1),
         (4, 4, 4),
         (2, 2, 2),
         (3, 3, 1),
         (1, 1, 1),
         (3, 2, 2),
         (0, 4, 2),
         (4, 0, 0),
         (0, 0, 3)
     ]

     labels = [
         1,
         1,
         1,
         1,
         1,
         1,
         -1,
         -1,
         -1,
         -1,
         -1,
         -1
     ]
```

2.1 a

```
[4]: print("Margins for each data point:")
     for point, label in zip(data, labels):
         x1, x2, x3 = point
         decision_value = 3*x1 + 2*x2 + 4*x3
         margin = label * (decision_value - 18) / np.sqrt(3**2 + 2**2 + 4**2)
         print(f'Point {point} with label {label}: {margin:.2f}')
```

```
Margins for each data point:
Point (2, 2, 3) with label 1: 0.74
Point (3, 3, 2) with label 1: 0.93
Point (1, 2, 3) with label 1: 0.19
Point (1, 4, 1) with label 1: -0.56
Point (4, 4, 4) with label 1: 3.34
Point (2, 2, 2) with label 1: 0.00
Point (3, 3, 1) with label -1: -0.19
Point (1, 1, 1) with label -1: 1.67
Point (3, 2, 2) with label -1: -0.56
Point (0, 4, 2) with label -1: 0.37
Point (4, 0, 0) with label -1: 1.11
Point (0, 0, 3) with label -1: 1.11
```

2.2 b

```
[5]: print("0-1 Loss for each data point:")
     for point, label in zip(data, labels):
         x1, x2, x3 = point
         decision_value = 3*x1 + 2*x2 + 4*x3
         prediction = 1 if decision_value >= 18 else -1
         loss = 0 if prediction == label else 1
         print(f'Point {point} with label {label}: {loss}')
```

```
0-1 Loss for each data point:
Point (2, 2, 3) with label 1: 0
Point (3, 3, 2) with label 1: 0
Point (1, 2, 3) with label 1: 0
Point (1, 4, 1) with label 1: 1
Point (4, 4, 4) with label 1: 0
Point (2, 2, 2) with label 1: 0
Point (3, 3, 1) with label -1: 1
Point (1, 1, 1) with label -1: 0
Point (3, 2, 2) with label -1: 1
Point (0, 4, 2) with label -1: 0
Point (4, 0, 0) with label -1: 0
Point (0, 0, 3) with label -1: 0
```

2.3 c

```
[7]: print("Hinge Loss for each data point:")
     for point, label in zip(data, labels):
         x1, x2, x3 = point
         decision_value = 3*x1 + 2*x2 + 4*x3
         hinge_loss = max(0, 1 - label * (decision_value - 18))
         print(f'Point {point} with label {label}: {hinge_loss}')
```

```
Hinge Loss for each data point:
Point (2, 2, 3) with label 1: 0
Point (3, 3, 2) with label 1: 0
Point (1, 2, 3) with label 1: 0
Point (1, 4, 1) with label 1: 4
Point (4, 4, 4) with label 1: 0
Point (2, 2, 2) with label 1: 1
Point (3, 3, 1) with label -1: 2
Point (1, 1, 1) with label -1: 0
Point (3, 2, 2) with label -1: 4
Point (0, 4, 2) with label -1: 0
Point (4, 0, 0) with label -1: 0
Point (0, 0, 3) with label -1: 0
```

2.4 d

```
[8]: print("Squared Loss for each data point:")
     for point, label in zip(data, labels):
         x1, x2, x3 = point
         decision_value = 3*x1 + 2*x2 + 4*x3
         squared_loss = (label - (decision_value - 18))**2
         print(f'Point {point} with label {label}: {squared_loss}')
```

```
Squared Loss for each data point:
Point (2, 2, 3) with label 1: 9
Point (3, 3, 2) with label 1: 16
Point (1, 2, 3) with label 1: 0
Point (1, 4, 1) with label 1: 16
Point (4, 4, 4) with label 1: 289
Point (2, 2, 2) with label 1: 1
Point (3, 3, 1) with label -1: 4
Point (1, 1, 1) with label -1: 64
Point (3, 2, 2) with label -1: 16
Point (0, 4, 2) with label -1: 1
Point (4, 0, 0) with label -1: 25
Point (0, 0, 3) with label -1: 25
```

3   5

```python
[11]: import time

      def gradient_descent(X, y, learning_rate=0.1, epochs=1000, regularization=1e-6):
          start_time = time.time()
          m, n = X.shape
          weights = np.zeros(n)
          obj_values = []
          for epoch in range(epochs):
              predictions = X @ weights
              errors = predictions - y
              gradient = (2/m) * (X.T @ errors) + 2 * regularization * weights
              if np.linalg.norm(gradient) < 1e-6:
                  if regularization > 0:
                      print("Completed GD w/ regularization in epoch:", epoch, "in",
      ↪time.time() - start_time, "seconds")
                  else:
                      print("Completed GD w/o regularization in epoch:", epoch, "in",
      ↪time.time() - start_time, "seconds")
                  return weights, obj_values
              weights -= learning_rate * gradient
              obj_values.append((errors @ errors) / m + regularization * (weights @
      ↪weights))
          if regularization > 0:
              print("Completed GD w/ regularization in epoch:", epoch, "in", time.
      ↪time() - start_time, "seconds")
          else:
              print("Completed GD w/o regularization in epoch:", epoch, "in", time.
      ↪time() - start_time, "seconds")
          return weights, obj_values

      def stochastic_gradient_descent(X, y, learning_rate=0.1, epochs=1000,
      ↪regularization=1e-6):
          start_time = time.time()
          m, n = X.shape
          weights = np.zeros(n)
          obj_values = []
          for epoch in range(epochs):
              for i in range(m):
                  xi = X[i]
                  yi = y[i]
                  prediction = xi @ weights
                  error = prediction - yi
                  gradient = (2/m) * xi * error + 2 * regularization * weights
                  weights -= learning_rate * gradient
              predictions = X @ weights
```

```python
        errors = predictions - y
        obj_values.append((errors @ errors) / m + regularization * (weights @␣
 ↪weights))
    if regularization > 0:
        print("Completed SGD w/ regularization in epoch:", epoch, "in", time.
 ↪time() - start_time, "seconds")
    else:
        print("Completed SGD w/o regularization in epoch:", epoch, "in", time.
 ↪time() - start_time, "seconds")
    return weights, obj_values

def closed_form_solution(X, y, regularization=1e-6):
    n = X.shape[1]
    return np.linalg.inv(X.T @ X + regularization * np.eye(n)) @ X.T @ y

X = np.load('data_X_Q5Q6.npy')
y = np.load('data_y_Q5Q6.npy').reshape(-1)

weights_gd, obj_gd = gradient_descent(X, y)
weights_sgd, obj_sgd = stochastic_gradient_descent(X, y)
weights_closed_form = closed_form_solution(X, y)

weights_gd_noReg, obj_gd_noReg = gradient_descent(X, y, regularization=0)
weights_sgd_noReg, obj_sgd_noReg = stochastic_gradient_descent(X, y,␣
 ↪regularization=0)
weights_closed_form_noReg = closed_form_solution(X, y, regularization=0)

def l2_norm_diff(w1, w2):
    return np.linalg.norm(w1 - w2)

print("L2 Norm Differences of Least Squares Regression:")
print("With Regularization:")
print("GD vs SGD:", l2_norm_diff(weights_gd, weights_sgd))
print("GD vs Closed Form:", l2_norm_diff(weights_gd, weights_closed_form))
print("SGD vs Closed Form:", l2_norm_diff(weights_sgd, weights_closed_form))
print("Without Regularization:")
print("GD vs SGD:", l2_norm_diff(weights_gd_noReg, weights_sgd_noReg))
print("GD vs Closed Form:", l2_norm_diff(weights_gd_noReg,␣
 ↪weights_closed_form_noReg))
print("SGD vs Closed Form:", l2_norm_diff(weights_sgd_noReg,␣
 ↪weights_closed_form_noReg))
```

```
Completed GD w/ regularization in epoch: 999 in 0.24744129180908203 seconds
Completed SGD w/ regularization in epoch: 999 in 79.89985132217407 seconds
Completed GD w/o regularization in epoch: 999 in 0.25933051109313965 seconds
Completed SGD w/o regularization in epoch: 999 in 87.29134202003479 seconds
L2 Norm Differences of Least Squares Regression:
```

```
With Regularization:
GD vs SGD: 1.3603089942927995
GD vs Closed Form: 5.191738520295133
SGD vs Closed Form: 5.998876146468824
Without Regularization
GD vs SGD: 0.0009529716509423173
GD vs Closed Form: 5.191620708668827
SGD vs Closed Form: 5.191819050801421
```

4   6

```python
[12]: plt.figure(figsize=(12, 5))
      plt.subplot(1, 2, 1)
      plt.plot(obj_gd, label='With Regularization', linestyle='-', marker='o')
      plt.xlabel('Epochs')
      plt.ylabel('Objective Value')
      plt.title('Gradient Descent Objective Values\nwith Regularization')
      plt.subplot(1, 2, 2)
      plt.plot(obj_gd_noReg, label='Without Regularization', color='orange',␣
       ↪linestyle='-', marker='o')
      plt.xlabel('Epochs')
      plt.ylabel('Objective Value')
      plt.title('Gradient Descent Objective Values\nwithout Regularization')
      plt.show()

      plt.figure(figsize=(12, 5))
      plt.subplot(1, 2, 1)
      plt.plot(obj_sgd, label='With Regularization', linestyle='-', marker='o')
      plt.xlabel('Epochs')
      plt.ylabel('Objective Value')
      plt.title('Stochastic Gradient Descent Objective Values\nwith Regularization')
      plt.subplot(1, 2, 2)
      plt.plot(obj_sgd_noReg, label='Without Regularization', color='orange',␣
       ↪linestyle='-', marker='o')
      plt.xlabel('Epochs')
      plt.ylabel('Objective Value')
      plt.title('Stochastic Gradient Descent Objective Values\nwithout␣
       ↪Regularization')
      plt.show()
```

Gradient Descent Objective Values with Regularization

Gradient Descent Objective Values without Regularization

Stochastic Gradient Descent Objective Values with Regularization

Stochastic Gradient Descent Objective Values without Regularization
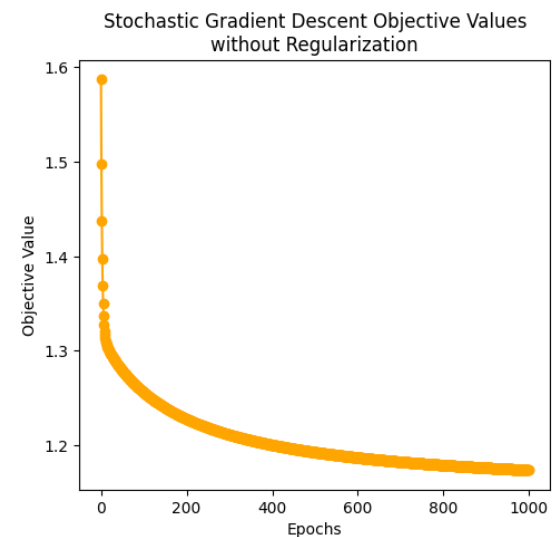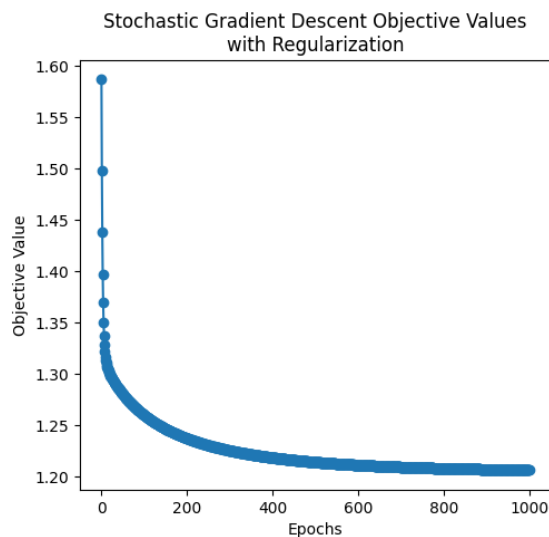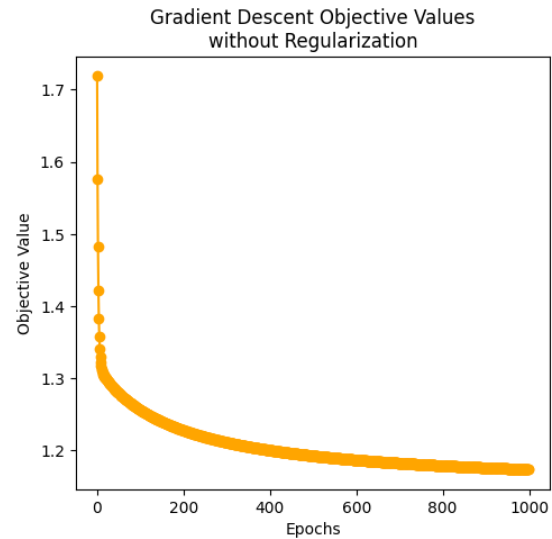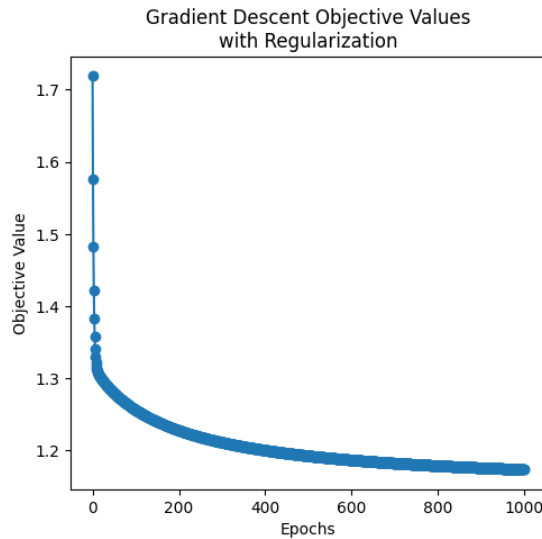
5   7

```
[17]: def batch_stochastic_gradient_descent(X, y, learning_rate=0.1, epochs=1000,
      ↪batch_size=32, regularization=1e-6):
          start_time = time.time()
          m, n = X.shape
          weights = np.zeros(n)
          obj_values = []
          for epoch in range(epochs):
              perm = np.random.permutation(m)
```

```
        X_shuffled = X[perm]
        y_shuffled = y[perm]
        for i in range(0, m, batch_size):
            xi = X_shuffled[i:i+batch_size]
            yi = y_shuffled[i:i+batch_size]
            prediction = xi @ weights
            error = prediction - yi
            gradient = (2/m) * (xi.T @ error) + 2 * regularization * weights
            weights -= learning_rate * gradient
        predictions = X @ weights
        errors = predictions - y
        obj_values.append((errors @ errors) / m + regularization * (weights @␣
  ↪weights))
    print(f"Completed Batch {batch_size} SGD w/ regularization in epoch:␣
  ↪{epoch} in {time.time() - start_time} seconds")
    return weights, obj_values

X = np.load('data_X_Q7.npy')
y = np.load('data_y_Q7.npy').reshape(-1)
batch_sgd_1, batch_obj_1 = batch_stochastic_gradient_descent(X, y, batch_size=1)
batch_sgd_10, batch_obj_10 = batch_stochastic_gradient_descent(X, y,␣
  ↪batch_size=10)
batch_sgd_100, batch_obj_100 = batch_stochastic_gradient_descent(X, y,␣
  ↪batch_size=100)
```

```
Completed Batch 1 SGD w/ regularization in epoch: 999 in 78.41542434692383
seconds
Completed Batch 10 SGD w/ regularization in epoch: 999 in 24.432676076889038
seconds
Completed Batch 100 SGD w/ regularization in epoch: 999 in 20.637977838516235
seconds
```

```python
[19]: plt.figure(figsize=(12, 15))
      plt.subplot(3, 1, 1)
      plt.plot(batch_obj_1, linestyle='-', marker='o')
      plt.xlabel('Epochs')
      plt.ylabel('Objective Value')
      plt.title('Batch Size 1 SGD Objective Values')
      plt.subplot(3, 1, 2)
      plt.plot(batch_obj_10, color='orange', linestyle='-', marker='o')
      plt.xlabel('Epochs')
      plt.ylabel('Objective Value')
      plt.title('Batch Size 10 SGD Objective Values')
      plt.subplot(3, 1, 3)
      plt.plot(batch_obj_100, color='green', linestyle='-', marker='o')
      plt.xlabel('Epochs')
      plt.ylabel('Objective Value')
```

```
plt.title('Batch Size 100 SGD Objective Values')
plt.show()
```


Batch Size 1 SGD Objective Values

Batch Size 10 SGD Objective Values

Batch Size 100 SGD Objective Values

13