Load Data

```python
import pandas as pd
import numpy as np
enroll_df = pd.read_csv('IRIS.csv')
```
✓ 0.4s Python

```python
enroll_df
```
✓ 0.0s Python

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns

1.[1 point] Prepare the data in one-against-the-rest strategy. This can be done by converting the "Species" column into 3 binary columns.

```python
data = pd.get_dummies(enroll_df)
```
✓ 0.0s

+ Code + Markdown

```python
data
```
✓ 0.0s

	sepal_length	sepal_width	petal_length	petal_width	species_Iris-setosa	species_Iris-versicolor	species_Iris-virginica
0	5.1	3.5	1.4	0.2	1	0	0
1	4.9	3.0	1.4	0.2	1	0	0
2	4.7	3.2	1.3	0.2	1	0	0
3	4.6	3.1	1.5	0.2	1	0	0
4	5.0	3.6	1.4	0.2	1	0	0
...	...	...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	0	0	1
146	6.3	2.5	5.0	1.9	0	0	1
147	6.5	3.0	5.2	2.0	0	0	1
148	6.2	3.4	5.4	2.3	0	0	1
149	5.9	3.0	5.1	1.8	0	0	1

150 rows × 7 columns

2. [2 points] Formulate the error function of the logistic regression with ridge regularization criterion.

$$L(w) = \frac{1}{n}\sum_{x=1}^{n}\left[-y_i \log H_w(x_i) - (1-y_i)\log(1-H_w(x_i))\right]$$

$$E(w) = L(w) + \lambda\|w\|_2^2$$

$$= L(w)_n + \lambda\sum_{i=1}^{k} w_i^2$$

$$= \frac{1}{n}\sum_{x=1}\left[-y_i \log H_w(x_i) - (1-y_i)\log(1-H_w(x_i))\right] + \lambda\sum_{i=1}^{k} w_i^2$$

3. [2 points] Derive the gradient of the error function by deriving the partial derivative of the error function in Task 2.

$$E(w) = \frac{1}{n}\sum_{i=1}^{n}\left[-y_i \log H_w(x_i) - (1-y_i)\log(1-H_w(x_i))\right] + \lambda\sum_{i=1}^{k} w_i^2$$

$$\frac{\partial E(w)}{\partial w} = \frac{\partial}{\partial w}\left(\frac{1}{n}\sum_{i=1}^{n}(-y \log H_w(x_i) - (1-y_i)\log(1-H_w(x_i))) + \lambda\sum_{i=1}^{k} w_i^2\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(-y_i \frac{1}{H_w(x_i)}\frac{\partial H_w(x_i)}{\partial w} - (1-y_i)\frac{1}{1-H_w(x_i)}\frac{\partial(1-H_w(x_i))}{\partial w}\right) + \lambda\sum_{i=1}^{k} 2w_i$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(\frac{-y_i}{\cancel{H_w(x_i)}}\,\cancel{H_w(x_i)}(1-H_w(x_i))x_i\right) - \frac{(1-y_i)}{\cancel{1-H_w(x_i)}}\,(1-\cancel{H_w(x_i)})x_i) + 2\lambda\sum_{i=1}^{k} w_i$$

$$= \frac{1}{n}\sum_{i=1}^{n}(-y_i(1-H_w(x_i)) + (1-y_i)H_w(x_i))x_i + 2\lambda\sum_{i=1}^{k} w_i$$

$$= \frac{1}{n}\sum_{i=1}^{n}(-y_i + y_i H_w(x_i) + H_w(x_i) - y_i H_w(x_i))x_i + 2\lambda\sum_{i=1}^{k} w_i$$

$$= \frac{1}{n}\sum_{i=1}^{n}(H_w(x_i) - y_i)x_i + 2\lambda\sum_{i=1}^{k} w_i$$

4. [2 point] Implement the gradient descent using all of the dataset in each iteration. (Use equation from Task 3)

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

def gradient_descent(x, y, learning_rate, lambda_param, epochs):
    w = np.zeros((x.shape[1], 3))  # Initialize weights for each feature and class
    n = x.shape[0]  # Number of data points
    loss_values = []

    for i in range(epochs):
        z = np.dot(x, w)
        H = sigmoid(z)

        # Calculate the gradient with respect to all weights using the derived formula
        gradient = (1/n) * np.dot(x.T, H - y) + 2 * lambda_param * w

        # Update weights using gradient descent
        w = w - learning_rate * gradient

        # Calculate the loss with ridge regularization
        loss = (1/n) * np.sum((-1 * y) * np.log(H) - (1 - y) * np.log(1 - H)) + (lambda_param * np.sum(w**2))
        loss_values.append(loss)

    return w, loss_values
```

```
learning_rate = 0.1
lambda_param = 0.001
epochs = 1000
print('Implement Gradient descent for Logistic Regression with Ridge regularization')
w_gd, loss_values_gd = gradient_descent(X, Y, learning_rate, lambda_param, epochs)
for i in range(len(loss_values_gd)):
    if i % 100 == 0 or i == 999:
        print(f"Iteration {i+1}, Loss: {loss_values_gd[i]}")
```

✓ 0.0s

```
Implement Gradient descent for Logistic Regression with Ridge regularization
Iteration 1, Loss: 2.079503881268725
Iteration 101, Loss: 0.9311673877006876
Iteration 201, Loss: 0.8446191660996369
Iteration 301, Loss: 0.801632724177872
Iteration 401, Loss: 0.77511864389027
Iteration 501, Loss: 0.7573177629839778
Iteration 601, Loss: 0.7447757986894198
Iteration 701, Loss: 0.7356431812712463
Iteration 801, Loss: 0.7288261097415312
Iteration 901, Loss: 0.7236357151154029
Iteration 1000, Loss: 0.7196539266370896
```

5. [1 point] Implement the stochastic gradient descent using the subset of dataset in each iteration. (Use equation from Task 3)

```
import numpy as np
def stochastic_gradient_descent(x, y, learning_rate, lambda_param, epochs, batch_size):
    w = np.zeros((x.shape[1], 3))  # Initialize weights for each feature and class
    n = x.shape[0]  # Number of data points
    loss_values = []

    for i in range(epochs):
        permutation = np.random.permutation(n)
        x = x[permutation]
        y = y[permutation]

        for j in range(0, n, batch_size):
            # Select a mini-batch of data
            x_batch = x[j:j + batch_size]
            y_batch = y[j:j + batch_size]

            z = np.dot(x_batch, w)
            H = sigmoid(z)

            # Calculate the gradient with respect to the mini-batch
            gradient = (1 / batch_size) * np.dot(x_batch.T, H - y_batch) + 2 * lambda_param * w

            # Update weights using gradient descent
            w = w - learning_rate * gradient

            # Calculate the loss with ridge regularization for the mini-batch
            loss = (1 / batch_size) * np.sum((-1 * y_batch) * np.log(H) - (1 - y_batch) * np.log(1 - H)) + (lambda_param * np.sum(w**2))

        # Calculate the loss for the entire dataset at the end of the epoch
        z = np.dot(x, w)
        H = sigmoid(z)
        loss = (1 / n) * np.sum((-1 * y) * np.log(H) - (1 - y) * np.log(1 - H)) + (lambda_param * np.sum(w)*2)
        loss_values.append(loss)

    return w, loss_values
```

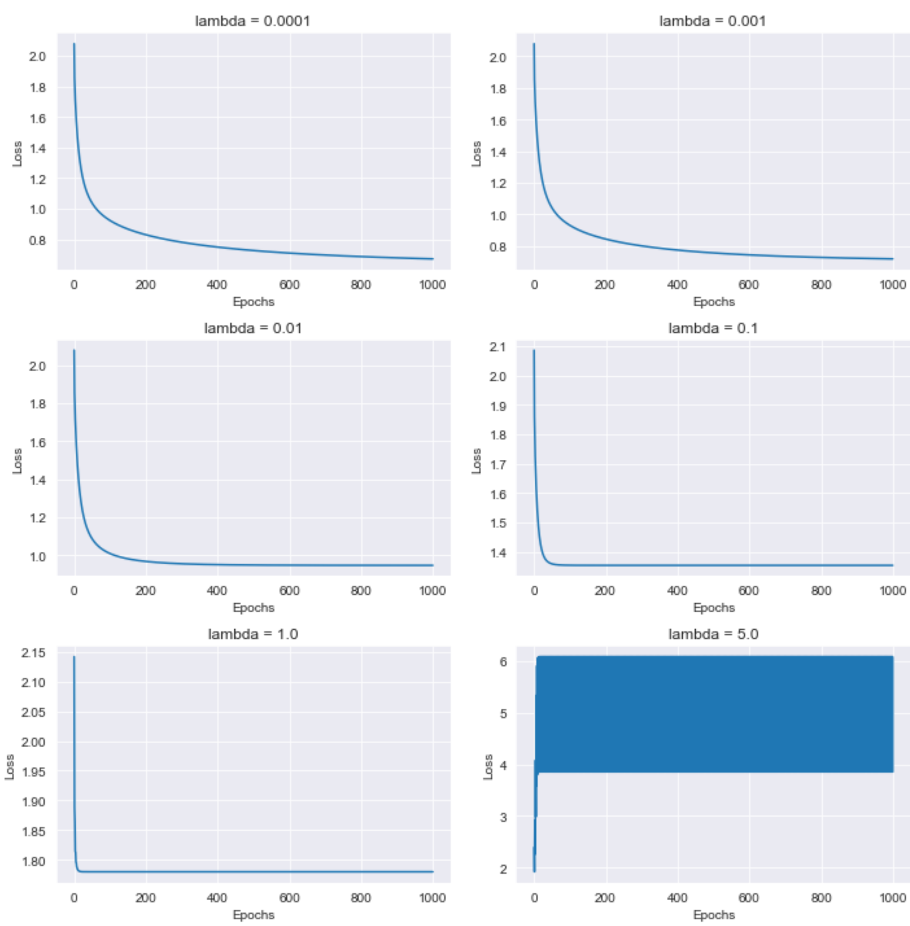
```
learning_rate = 0.1
lambda_param = 0.001
epochs = 1000
batch_size = 32


print('Implement Gradient descent for Logistic Regression with Ridge regularization')
w_sgd, loss_values_sgd = stochastic_gradient_descent(X, Y, learning_rate, lambda_param, epochs, batch_size)

for i in range(len(loss_values_sgd)):
    if i % 100 == 0 or i == 999:
        print(f"Iteration {i+1}, Loss: {loss_values_sgd[i]}")
```

✓ 0.1s

```
Implement Gradient descent for Logistic Regression with Ridge regularization
Iteration 1, Loss: 1.6987674089871094
Iteration 101, Loss: 0.7403831122887613
Iteration 201, Loss: 0.6832734239064758
Iteration 301, Loss: 0.6898555943641357
Iteration 401, Loss: 0.6639819503489421
Iteration 501, Loss: 0.6467345826657445
Iteration 601, Loss: 0.6590108965410247
Iteration 701, Loss: 0.6381831019999711
Iteration 801, Loss: 0.6453761926355454
Iteration 901, Loss: 0.6475774444014177
Iteration 1000, Loss: 0.6579142660885449
```

6. [1 point] Test to see the effect of l on the training process.

```python
import matplotlib.pyplot as plt
import seaborn as sns
learning_rate = 0.1
lambda_values = [0.0001, 0.001, 0.01, 0.1, 1.0, 5.0]
epochs = 1000
sns.set_style("darkgrid")
fig, axs = plt.subplots(3, 2, figsize=(10, 10))
for i, lambda_param in enumerate(lambda_values):
    w, loss_values = gradient_descent(X, Y, learning_rate, lambda_param, epochs)
    row = i // 2
    col = i % 2
    axs[row, col].plot(loss_values)
    axs[row, col].set_title(f'lambda = {lambda_param}')
    axs[row, col].set_xlabel('Epochs')
    axs[row, col].set_ylabel('Loss')
plt.tight_layout()
plt.show()
```



7. [1 point] Test to see the effect of sampling proportion in Task 5

```
# Define your stochastic_gradient_descent function here if it's not already defined

learning_rate = 0.1
lambda_param = 0.001
epochs = 1000
batch_sizes = [1, 16, 32, 64, 128, 150]

sns.set_style("darkgrid")
fig, axs = plt.subplots(3, 2, figsize=(10, 10))
for i, batch_size in enumerate(batch_sizes):
    w, loss_values = stochastic_gradient_descent(X, Y, learning_rate, lambda_param, epochs, batch_size)
    row = i // 2
    col = i % 2
    axs[row, col].plot(loss_values)
    axs[row, col].set_title(f'batch size = {batch_size}')
    axs[row, col].set_xlabel('Epochs')
    axs[row, col].set_ylabel('Loss')
plt.tight_layout()
plt.show()
```