A6 Code Implementation

```python
from typing import Dict, List, Tuple

import matplotlib.pyplot as plt
import numpy as np

from utils import load_dataset, problem

# When choosing your batches / Shuffling your data you should use this RNG variable, and not
`np.random.choice` etc.
RNG = np.random.RandomState(seed=446)
Dataset = Tuple[Tuple[np.ndarray, np.ndarray], Tuple[np.ndarray, np.ndarray]]


def load_2_7_mnist() -> Dataset:
    """
    Loads MNIST data, extracts only examples with 2, 7 as labels, and converts them into -1,
1 labels, respectively.

    Returns:
        Dataset: 2 tuples of numpy arrays, each containing examples and labels.
            First tuple is for training, while second is for testing.
            Shapes as as follows: ((n, d), (n,)), ((m, d), (m,))
    """
    (x_train, y_train), (x_test, y_test) = load_dataset("mnist")
    train_idxs = np.logical_or(y_train == 2, y_train == 7)
    test_idxs = np.logical_or(y_test == 2, y_test == 7)

    y_train_2_7 = y_train[train_idxs]
    y_train_2_7 = np.where(y_train_2_7 == 7, 1, -1)

    y_test_2_7 = y_test[test_idxs]
    y_test_2_7 = np.where(y_test_2_7 == 7, 1, -1)

    return (x_train[train_idxs], y_train_2_7), (x_test[test_idxs], y_test_2_7)


class BinaryLogReg:
    @problem.tag("hw2-A", start_line=4)
    def __init__(self, _lambda: float = 1e-3):
        """Initializes the Binary Log Regression model.
        Args:
            _lambda (float, optional): Ridge Regularization coefficient. Defaults to 1e-3.
        """
        self._lambda: float = _lambda
        # Fill in with matrix with the correct shape
        self.weight: np.ndarray = None  # type: ignore
        self.bias: float = 0.0
        #raise NotImplementedError("Your Code Goes Here")

    @problem.tag("hw2-A")
    def mu(self, X: np.ndarray, y: np.ndarray) -> np.ndarray:
        """Calculate mu in vectorized form, as described in the problem.
        The equation for i^th element of vector mu is given by:

        $$ \mu_i = 1 / (1 + \exp(-y_i (bias + x_i^T weight))) $$

        Args:
            X (np.ndarray): observations represented as `(n, d)` matrix.
                n is number of observations, d is number of features.
                d = 784 in case of MNIST.
            y (np.ndarray): targets represented as `(n, )` vector.
                n is number of observations.
        Returns:
            np.ndarray: An `(n, )` vector containing mu_i for i^th element.
        """
        linear_term = self.bias + X @ self.weight
        return 1 / (1 + np.exp(-y * linear_term))

    @problem.tag("hw2-A")
    def loss(self, X: np.ndarray, y: np.ndarray) -> float:
        """Calculate loss J as defined in the problem.

        Args:
            X (np.ndarray): observations represented as `(n, d)` matrix.
                n is number of observations, d is number of features.
                d = 784 in case of MNIST.
            y (np.ndarray): targets represented as `(n, )` vector.
                n is number of observations.

        Returns:
            float: Loss given X, y, self.weight, self.bias and self._lambda
        """
        n = X.shape[0]
        mu_vals = self.mu(X, y)
        log_likelihood = np.log(1 + np.exp(-y * (self.bias + X @ self.weight)))
        return np.mean(log_likelihood) + self._lambda * np.linalg.norm(self.weight) ** 2


    @problem.tag("hw2-A")
    def gradient_J_weight(self, X: np.ndarray, y: np.ndarray) -> np.ndarray:
        """Calculate gradient of loss J with respect to weight.

        Args:
            X (np.ndarray): observations represented as `(n, d)` matrix.
                n is number of observations, d is number of features.
                d = 784 in case of MNIST.
            y (np.ndarray): targets represented as `(n, )` vector.
                n is number of observations.
        Returns:
            np.ndarray: An `(d, )` vector which represents gradient of loss J with respect to
self.weight.
        """
        n = X.shape[0]
        mu_vals = self.mu(X, y)
        gradient = -(1 / n) * (X.T @ (y * (1 - mu_vals))) + 2 * self._lambda * self.weight
        return gradient

    @problem.tag("hw2-A")
    def gradient_J_bias(self, X: np.ndarray, y: np.ndarray) -> float:
        """Calculate gradient of loss J with respect to bias.

        Args:
            X (np.ndarray): observations represented as `(n, d)` matrix.
                n is number of observations, d is number of features.
                d = 784 in case of MNIST.
            y (np.ndarray): targets represented as `(n, )` vector.
                n is number of observations.

        Returns:
            float: A number that represents gradient of loss J with respect to self.bias.
        """
        n = X.shape[0]
        mu_vals = self.mu(X, y)
        return -(1 / n) * np.sum(y * (1 - mu_vals))

    @problem.tag("hw2-A")
    def predict(self, X: np.ndarray) -> np.ndarray:
        """Given X, weight and bias predict values of y.

        Args:
            X (np.ndarray): observations represented as `(n, d)` matrix.
                n is number of observations, d is number of features.
                d = 784 in case of MNIST.

        Returns:
            np.ndarray: An `(n, )` array of either -1s or 1s representing guess for each
observation.
        """
        return np.sign(self.bias + X @ self.weight)

    @problem.tag("hw2-A")
    def misclassification_error(self, X: np.ndarray, y: np.ndarray) -> float:
        """Calculates misclassification error (the rate at which this model is making
incorrect predictions of y).
        Note that `misclassification_error = 1 - accuracy`.

        Args:
            X (np.ndarray): observations represented as `(n, d)` matrix.
                n is number of observations, d is number of features.
                d = 784 in case of MNIST.
            y (np.ndarray): targets represented as `(n, )` vector.
                n is number of observations.

        Returns:
            float: percentage of times prediction did not match target, given an observation
(i.e. misclassification error).
        """
        predictions = self.predict(X)
        return np.mean(predictions != y)

    @problem.tag("hw2-A")
    def step(self, X: np.ndarray, y: np.ndarray, learning_rate: float = 1e-4):
        """Single step in training loop.
        It does not return anything but should update self.weight and self.bias with correct
values.

        Args:
            X (np.ndarray): observations represented as `(n, d)` matrix.
                n is number of observations, d is number of features.
                d = 784 in case of MNIST.
            y (np.ndarray): targets represented as `(n, )` vector.
                n is number of observations.
            learning_rate (float, optional): Learning rate of SGD/GD algorithm.
                Defaults to 1e-4.
        """
        self.weight -= learning_rate * self.gradient_J_weight(X, y)
        self.bias -= learning_rate * self.gradient_J_bias(X, y)


    @problem.tag("hw2-A", start_line=7)
    def train(
        self,
        X_train: np.ndarray,
        y_train: np.ndarray,
        X_test: np.ndarray,
        y_test: np.ndarray,
        learning_rate: float = 1e-2,
        epochs: int = 30,
        batch_size: int = 100,
    ) -> Dict[str, List[float]]:
        """Train function that given dataset X_train and y_train adjusts weights and biases
of this model.
        It also should calculate misclassification error and J loss at the END of each epoch.

        For each epoch please call step function `num_batches` times as defined on top of the
 starter code.

        NOTE: This function due to complexity and number of possible implementations will not
 be publicly unit tested.
        However, we might still test it using gradescope, and you will be graded based on the
 plots that are generated using this function.

        Args:
            X_train (np.ndarray): observations in training set represented as `(n, d)` matrix
.
                n is number of observations, d is number of features.
                d = 784 in case of MNIST.
            y_train (np.ndarray): targets in training set represented as `(n, )` vector.
                n is number of observations.
            X_test (np.ndarray): observations in testing set represented as `(m, d)` matrix.
                m is number of observations, d is number of features.
                d = 784 in case of MNIST.
            y_test (np.ndarray): targets in testing set represented as `(m, )` vector.
                m is number of observations.
            learning_rate (float, optional): Learning rate of SGD/GD algorithm. Defaults to 1
e-2.
            epochs (int, optional): Number of epochs (loops through the whole data) to train
SGD/GD algorithm for.
                Defaults to 30.
            batch_size (int, optional): Number of observation/target pairs to use for a
single update.
                Defaults to 100.

        Returns:
            Dict[str, List[float]]: Dictionary containing 4 keys, each pointing to a list/
numpy array of length `epochs`:
            {
                "training_losses": [<Loss at the end of each epoch on training set>],
                "training_errors": [<Misclassification error at the end of each epoch on
training set>],
                "testing_losses": [<Same as above but for testing set>],
                "testing_errors": [<Same as above but for testing set>],
            }
            Skeleton for this result is provided in the starter code.

        Note:
            - When shuffling batches/randomly choosing batches makes sure you are using RNG
variable defined on the top of the file.
        """
        num_batches = int(np.ceil(len(X_train) // batch_size))
        result: Dict[str, List[float]] = {
            "train_losses": [],  # You should append to these lists
            "train_errors": [],
            "test_losses": [],
            "test_errors": [],
        }
        n, d = X_train.shape
        self.weight = np.zeros(d)
        self.bias = 0.0
        for epoch in range(epochs):
            if num_batches == 1:
                self.step(X_train, y_train, learning_rate)
            else:
                indices = RNG.permutation(n)
                X_train_shuffled = X_train[indices]
                y_train_shuffled = y_train[indices]
                for i in range(num_batches):
                    start = i * batch_size
                    end = min(start + batch_size, n)
                    X_batch = X_train_shuffled[start:end]
                    y_batch = y_train_shuffled[start:end]
                    self.step(X_batch, y_batch, learning_rate)

            result["train_losses"].append(self.loss(X_train, y_train))
            result["train_errors"].append(self.misclassification_error(X_train, y_train))
            result["test_losses"].append(self.loss(X_test, y_test))
            result["test_errors"].append(self.misclassification_error(X_test, y_test))

        return result


if __name__ == "__main__":
    model = BinaryLogReg()
    (x_train, y_train), (x_test, y_test) = load_2_7_mnist()
    #history = model.train(x_train, y_train, x_test, y_test, batch_size=len(x_train)) # batch
    #history = model.train(x_train, y_train, x_test, y_test, batch_size=1, learning_rate=1e
-4) # SGD
    history = model.train(x_train, y_train, x_test, y_test, batch_size=100, learning_rate=1e
-4) # mini-batch

    # Plot J(w, b) over iterations
    plt.figure(figsize=(8, 6))
    plt.plot(history["train_losses"], label="Training Loss", linestyle='-', marker='o')
    plt.plot(history["test_losses"], label="Test Loss", linestyle='-', marker='o')
    plt.xlabel("Epochs")
    plt.ylabel("Loss J(w, b)")
    plt.title("Loss vs. Iteration Number")
    plt.legend()
    plt.grid()
    plt.show()

    # Plot misclassification error over iterations
    plt.figure(figsize=(8, 6))
    plt.plot(history["train_errors"], label="Training Error", linestyle='-', marker='o')
    plt.plot(history["test_errors"], label="Test Error", linestyle='-', marker='o')
    plt.xlabel("Epochs")
    plt.ylabel("Misclassification Error")
    plt.title("Misclassification Error vs. Iteration Number")
    plt.legend()
    plt.grid()
    plt.show()
```