A1

(a)

L1 regularization results in sparsity because it applies a constant shrinkage force to all weights, regardless of their size. When a weight is small enough, this force pushes it past zero.

L2 regularization, in contrast, applies a proportional shrinkage force, meaning the smaller a weight is, the smaller the force acting on it. As a result, weights gradually approach zero but never become exactly zero, leading to small but nonzero values instead of sparsity.

(b)

A potential upside of the given regularizer is that it penalizes large weights less than L1 or L2 regularization because the sum $\sum_i |w_i|^{0.5}$ grows more slowly than $\sum_i |w_i|$ or $\sum_i w_i^2$, meaning large weights contribute less to the total penalty. This allows important features to remain nonzero while still enforcing sparsity.

A downside is that it may lead to over-sparsification because it disproportionately penalizes smaller weights, potentially eliminating moderately useful features that would have been retained under L1 or L2 regularization, which can harm model performance.

(c)

True. If the step size in gradient descent is too large, the updates may overshoot the optimal solution, causing the loss function to oscillate or even diverge instead of converging. This prevents the algorithm from settling at a minimum, making training unstable and ineffective.

(d)

An advantage of SGD over Batch Gradient Descent is that it updates the model parameters after each data point, making it significantly faster and more scalable for large datasets. A disadvantage of SGD is that its updates have higher variance, causing the optimization process to oscillate and making convergence less stable compared to Batch Gradient Descent.

(e)

Gradient Descent is necessary for logistic regression because the loss function (log-loss) is non-linear and does not have a closed-form solution due to the presence of the sigmoid function, which introduces exponentials into the optimization equation. In contrast, linear regression has a closed-form solution, allowing the optimal weights to be computed directly without iterative optimization.

A2

(I)

The set is **not convex** because there exist points within the set whose connecting line segment partially lies outside the set. Specifically, selecting points b and c, the straight line segment between them passes through the missing cutout region, which is not included in the set. Since a convex set must contain the entire line segment between any two of its points, this violation confirms that the set is non-convex.

(II)

The set is **not convex** because there exist points within the set whose connecting line segment partially exits the set. Specifically, selecting points a and d, the straight line segment between them passes through the indented cutout region, which is not part of the set. Since convexity requires that all such line segments remain entirely within the set, this counterexample proves that the set is non-convex.

A3

(a) Function in Panel I on $[a, c]$

The function in Panel I is convex on $[a, c]$ because for any two points in this interval, the function always lies below or on the line segment connecting them. Since convexity requires that the function does not rise above any such line segment, the function satisfies the convexity condition on this interval.

(b) Function in Panel II on $[a, d]$

The function in Panel II is not convex on $[a, d]$ because there exist points where the function is above the straight line segment connecting two points in the interval. Specifically, if we draw a line connecting $(a, f(a))$ and $(d, f(d))$, there are points around b where the function curves above this line, violating the convexity condition. Since convex functions must always stay below or on such line segments, this counterexample shows that the function is not convex on $[a, d]$.

A4

(a) Lasso Regularization Path

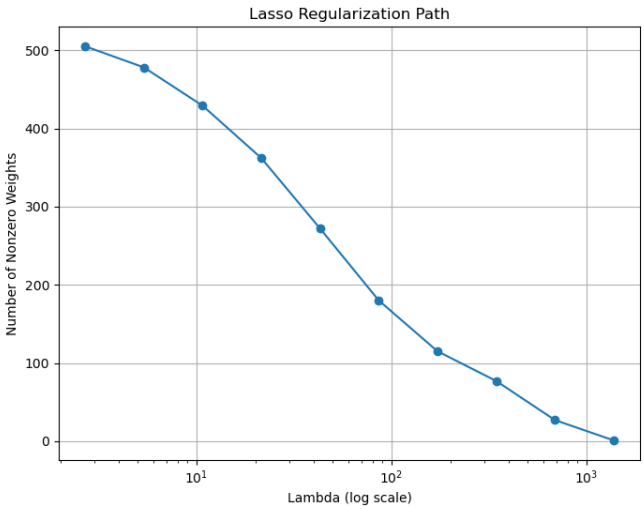


Figure 1: Lasso Regularization Path: Number of nonzero weights as a function of λ on a log scale.

(b) False Discovery Rate vs. True Positive Rate

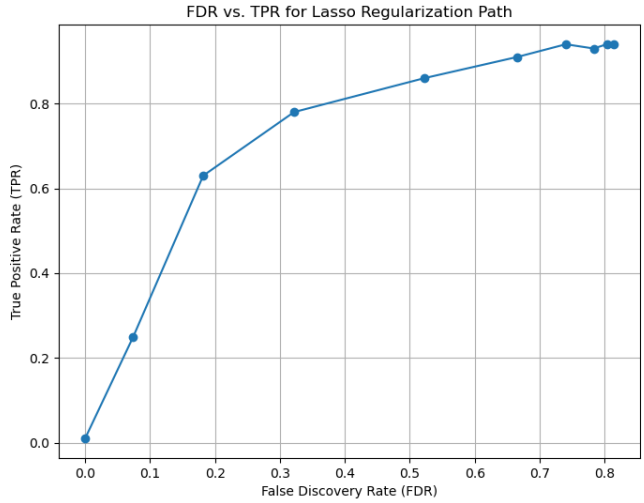


Figure 2: FDR vs. TPR for Lasso Regularization Path. The trade-off between selecting true features and avoiding false discoveries is observed.

(c) Effect of λ

As λ increases, the Lasso penalty forces more weights to zero, leading to fewer selected features, as seen in the Lasso regularization path. In the FDR vs. TPR plot, increasing λ initially reduces false discoveries (FDR) but at the cost of also reducing true positives (TPR), illustrating the trade-off between sparsity and correct feature selection.

Code Implementation

```
from typing import Optional, Tuple

import matplotlib.pyplot as plt
import numpy as np

from utils import problem

@problem.tag("hw2-A")
def precalculate_a(X: np.ndarray) -> np.ndarray:
    return 2 * np.sum(X ** 2, axis=0)

@problem.tag("hw2-A")
def step(
    X: np.ndarray, y: np.ndarray, weight: np.ndarray, a: np.ndarray, _lambda: float
) -> Tuple[np.ndarray, float]:
    n, d = X.shape
    residuals = y - (X @ weight)
    b = np.mean(residuals)
    for k in range(d):
        c_k = 2 * np.sum(X[:, k] * (y - (b + X @ weight + (-weight[k] * X[:, k]))))
        if c_k < -_lambda:
            weight[k] = (c_k + _lambda) / a[k]
        elif c_k > _lambda:
            weight[k] = (c_k - _lambda) / a[k]
        else:
            weight[k] = 0
    return weight, b

@problem.tag("hw2-A")
def loss(
    X: np.ndarray, y: np.ndarray, weight: np.ndarray, bias: float, _lambda: float
) -> float:
    mse = np.mean((X @ weight + bias - y) ** 2)
    l1_penalty = _lambda * np.sum(np.abs(weight))
    return mse + l1_penalty

@problem.tag("hw2-A", start_line=4)
def train(
    X: np.ndarray,
    y: np.ndarray,
    _lambda: float = 0.01,
    convergence_delta: float = 1e-4,
    start_weight: np.ndarray = None,
) -> Tuple[np.ndarray, float]:
    if start_weight is None:
        start_weight = np.zeros(X.shape[1])
    a = precalculate_a(X)
    old_w: Optional[np.ndarray] = None
    weight = np.copy(start_weight)
    while old_w is None or not convergence_criterion(weight, old_w, convergence_delta):
        old_w = np.copy(weight)
        weight, bias = step(X, y, weight, a, _lambda)
    return weight, bias

@problem.tag("hw2-A")
def convergence_criterion(
    weight: np.ndarray, old_w: np.ndarray, convergence_delta: float
) -> bool:
    return np.max(np.abs(weight - old_w)) < convergence_delta

def generate_synthetic_data(n=500, d=1000, k=100, sigma=1):
    np.random.seed(42)
```

```
    X = np.random.randn(n, d)
    X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)
    true_w = np.zeros(d)
    for j in range(k):
        true_w[j] = (j + 1) / k
    epsilon = np.random.randn(n) * sigma
    y = X @ true_w + epsilon
    return X, y, true_w

def compute_lambda_max(X, y):
    y_mean = np.mean(y)
    return np.max(2 * np.abs(X.T @ (y - y_mean)))

def lasso_regularization_metrics(X, y, true_w, lambda_max, num_lambdas=10, factor=2):
    lambdas = [lambda_max / (factor ** i) for i in range(num_lambdas)]
    nonzeros = []
    fdr_values = []
    tpr_values = []

    weight = np.zeros(X.shape[1])

    for _lambda in lambdas:
        weight, _ = train(X, y, _lambda, start_weight=weight)

        nonzero_indices = np.where(weight != 0)[0]  # Features selected by Lasso
        true_nonzero_indices = np.where(true_w != 0)[0]  # True relevant features

        num_false_discoveries = np.sum(np.isin(nonzero_indices, true_nonzero_indices, invert=True
            ))
        num_true_positives = np.sum(np.isin(nonzero_indices, true_nonzero_indices))

        total_nonzeros = len(nonzero_indices)
        fdr = num_false_discoveries / total_nonzeros if total_nonzeros > 0 else 0
        tpr = num_true_positives / len(true_nonzero_indices) if len(true_nonzero_indices) > 0
            else 0

        nonzeros.append(total_nonzeros)
        fdr_values.append(fdr)
        tpr_values.append(tpr)

    return lambdas, nonzeros, fdr_values, tpr_values

def plot_lasso_path(lambdas, nonzeros):
    plt.figure(figsize=(8, 6))
    plt.plot(lambdas, nonzeros, marker='o', linestyle='-')
    plt.xscale('log')
    plt.xlabel("Lambda (log scale)")
    plt.ylabel("Number of Nonzero Weights")
    plt.title("Lasso Regularization Path")
    plt.grid(True)
    plt.show(block=False)

def plot_fdr_tpr(fdr_values, tpr_values):
    plt.figure(figsize=(8, 6))
    plt.plot(fdr_values, tpr_values, marker='o', linestyle='-')
    plt.xlabel("False Discovery Rate (FDR)")
    plt.ylabel("True Positive Rate (TPR)")
    plt.title("FDR vs. TPR for Lasso Regularization Path")
    plt.grid(True)
    plt.show()

@problem.tag("hw2-A")
def main():
    X, y, true_w = generate_synthetic_data(n=500, d=1000, k=100, sigma=1)
    lambda_max = compute_lambda_max(X, y)
```

```
    lambdas, nonzeros, fdr_values, tpr_values = lasso_regularization_metrics(X, y, true_w,
        lambda_max)
    plot_lasso_path(lambdas, nonzeros)
    plot_fdr_tpr(fdr_values, tpr_values)

if __name__ == "__main__":
    main()
```

A5

(a)

Many features in the dataset are influenced by historical policy decisions in the U.S., as laws, government funding, and policy priorities shape economic, law enforcement, and urban development outcomes. Below are three examples of such features:

- **Per Capita Income**: Government policies such as tax regulations, minimum wage laws, social security benefits, and job creation programs directly influence income distribution across communities. Additionally, differences in *state income tax* policies can lead to significant variations in per capita income, as states with lower taxes may attract higher-income residents while others use tax revenues for social welfare programs.
- **Public Transportation Usage**: The availability and usage of public transit systems are heavily dependent on government infrastructure investments, zoning laws, and urban planning policies. Cities with significant public transit funding tend to have higher ridership, whereas communities with lower investments force residents to rely on personal vehicles, affecting commuting patterns and economic mobility.
- **Police Budget Allocation**: Law enforcement funding is determined by local and federal budget allocations, city council decisions, and crime prevention policies. Some municipalities prioritize increased policing with higher budgets, while others may divert resources toward community programs, education, or social services, leading to variation in police funding across different areas.

These examples illustrate how government decisions and policies shape the data in ways that are not purely demographic or economic but instead reflect historical and political choices.

(b)

Some features in the dataset might be found to have nonzero weights in the model, suggesting a correlation with violent crime. However, these features may actually be a *result* of crime rather than a *cause*, making it important to consider the direction of causality. Below are three such examples:

- **Police Presence**: A model may indicate that higher police presence is associated with higher crime rates. However, this is likely because areas with more crime receive increased police funding and personnel as a response, rather than police presence causing crime.
- **Vacant Housing**: High vacancy rates may appear as a predictor of crime, but in reality, crime often *causes* higher vacancy rates, as people move away from high-crime areas, leading to property abandonment.
- **Unemployment Rate**: Areas with high unemployment may exhibit higher crime rates, but rather than unemployment causing crime, businesses often avoid high-crime areas, reducing job opportunities and leading to a cycle where crime and unemployment reinforce each other.

(f)

When training the Lasso model, the feature with the **largest positive coefficient** was **PctIlleg** (percentage of births that are classified as illegitimate), while the feature with the **most negative coefficient** was **PctKids2Par** (percentage of children in two-parent households).

The positive coefficient on **PctIlleg** suggests that areas with a higher percentage of children born outside of marriage may have higher crime rates. The negative coefficient on **PctKids2Par** suggests that areas with a higher percentage of children raised in two-parent households tend to have lower crime rates.

(g)

Suppose a model finds a large negative weight on **agePct65up**, meaning that communities with a higher percentage of people aged 65 and older tend to have lower crime rates. A politician might then propose a policy to encourage elderly individuals to move into high-crime areas in an effort to reduce crime.

This reasoning is flawed because it confuses **correlation with causation**. Just because a variable is correlated with crime does not mean that changing that variable will directly affect crime rates.

This is an example of the **fire truck fallacy**: fire trucks are often seen near burning buildings, but that does not mean fire trucks cause fires. Similarly, areas with a higher percentage of elderly residents might have lower crime rates due to other underlying factors—such as socioeconomic conditions, historical crime trends, or law enforcement strategies—rather than the presence of elderly residents themselves. Encouraging elderly individuals to move to high-crime areas would not necessarily lower crime rates and could, in fact, put them at greater risk.

(c)

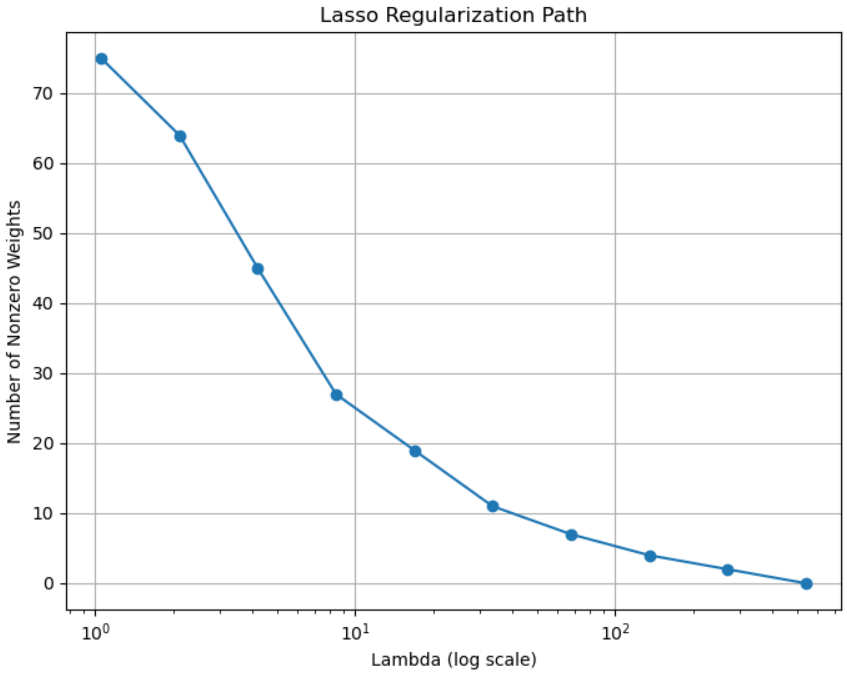


Figure 1: Lasso Regularization Path: Number of nonzero weights as a function of λ.

(d)

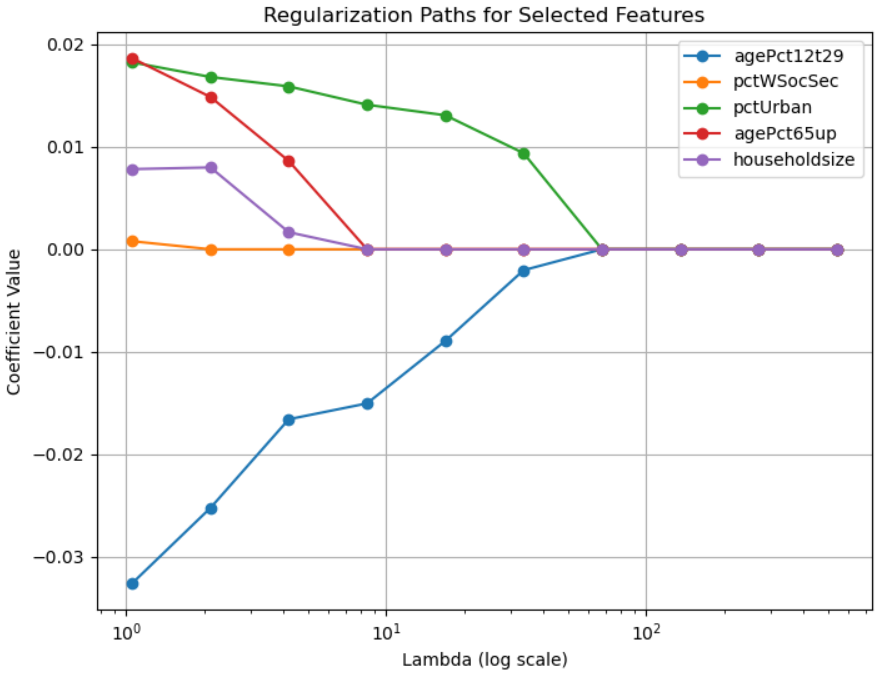


Figure 2: Regularization Paths for Selected Features: Coefficient values as a function of λ.

(e)

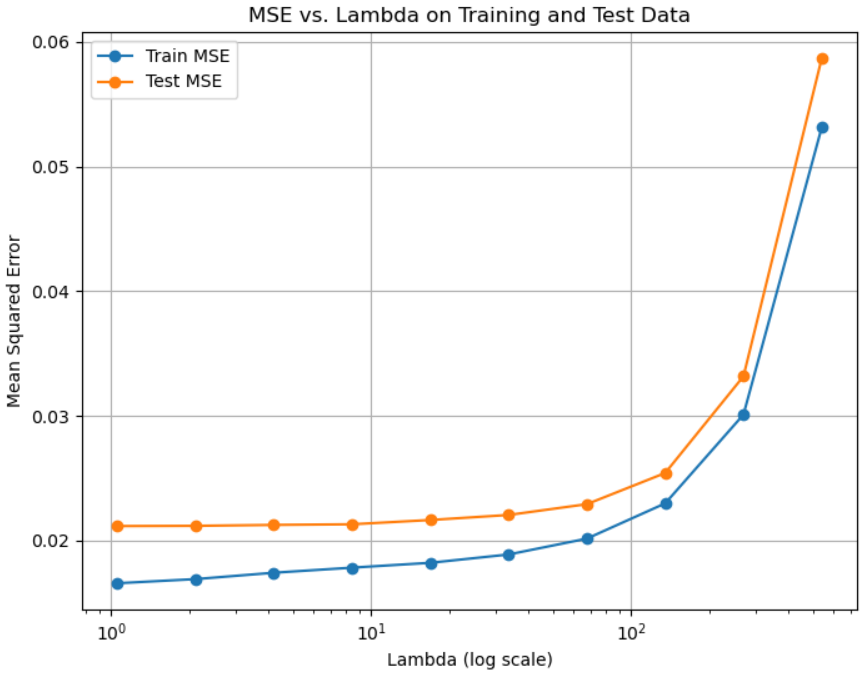


Figure 3: MSE vs. λ: Training and test mean squared error as a function of λ.

Code Implementation

```
if __name__ == "__main__":
    from coordinate_descent_algo import train, compute_lambda_max  # type: ignore
else:
    from .coordinate_descent_algo import train

import matplotlib.pyplot as plt
import numpy as np

from utils import load_dataset, problem

def plot_nonzero_weights(lambdas, nonzeros):
    plt.figure(figsize=(8, 6))
    plt.plot(lambdas, nonzeros, marker='o', linestyle='-')
    plt.xscale('log')
    plt.xlabel("Lambda (log scale)")
    plt.ylabel("Number of Nonzero Weights")
    plt.title("Lasso Regularization Path")
    plt.grid(True)
    plt.show(block=False)


def plot_regularization_paths(lambdas, coef_paths):
    plt.figure(figsize=(8, 6))
    for feature, path in coef_paths.items():
        plt.plot(lambdas, path, marker='o', linestyle='-', label=feature)

    plt.xscale('log')
    plt.xlabel("Lambda (log scale)")
    plt.ylabel("Coefficient Value")
    plt.title("Regularization Paths for Selected Features")
    plt.legend()
    plt.grid(True)
    plt.show(block=False)

def compute_mse(X, y, weight, bias):
    predictions = X @ weight + bias
    return np.mean((predictions - y) ** 2)

def plot_mse_vs_lambda(lambdas, mse_train, mse_test):
    plt.figure(figsize=(8, 6))
    plt.plot(lambdas, mse_train, marker='o', linestyle='-', label="Train MSE")
    plt.plot(lambdas, mse_test, marker='o', linestyle='-', label="Test MSE")

    plt.xscale('log')
    plt.xlabel("Lambda (log scale)")
    plt.ylabel("Mean Squared Error")
    plt.title("MSE vs. Lambda on Training and Test Data")
    plt.legend()
    plt.grid(True)
    plt.show()

def train_and_analyze_lasso(X_train, y_train, X_test, y_test, df_train):
    lambda_max = compute_lambda_max(X_train, y_train)
    num_lambdas = 10
    lambdas = [lambda_max / (2**i) for i in range(num_lambdas)]

    nonzeros = []
    mse_train = []
    mse_test = []
    selected_features = ["agePct12t29", "pctWSocSec", "pctUrban", "agePct65up", "householdsize"]
    coef_paths = {feature: [] for feature in selected_features}
```

```
    weight = np.zeros(X_train.shape[1])
    for _lambda in lambdas:
        weight, bias = train(X_train, y_train, _lambda, start_weight=weight)

        nonzeros.append(np.count_nonzero(weight))
        mse_train.append(compute_mse(X_train, y_train, weight, bias))
        mse_test.append(compute_mse(X_test, y_test, weight, bias))

        for feature in selected_features:
            coef_paths[feature].append(weight[df_train.columns[1:].tolist().index(feature)])

    plot_nonzero_weights(lambdas, nonzeros)
    plot_regularization_paths(lambdas, coef_paths)
    plot_mse_vs_lambda(lambdas, mse_train, mse_test)

def analyze_largest_coefficients(X_train, y_train, df_train, lambda_value=30):
    weight, _ = train(X_train, y_train, _lambda=lambda_value)

    feature_names = df_train.columns[1:].tolist()
    max_feature = feature_names[np.argmax(weight)]
    min_feature = feature_names[np.argmin(weight)]

    print(f"Largest␣positive␣coefficient␣feature:␣{max_feature}")
    print(f"Largest␣negative␣coefficient␣feature:␣{min_feature}")


@problem.tag("hw2-A", start_line=3)
def main():
    df_train, df_test = load_dataset("crime")

    X_train, y_train = df_train.iloc[:, 1:].values, df_train.iloc[:, 0].values
    X_test, y_test = df_test.iloc[:, 1:].values, df_test.iloc[:, 0].values

    X_mean = np.mean(X_train, axis=0)
    X_std = np.std(X_train, axis=0)
    X_train = (X_train - X_mean) / X_std
    X_test = (X_test - X_mean) / X_std

    train_and_analyze_lasso(X_train, y_train, X_test, y_test, df_train)
    analyze_largest_coefficients(X_train, y_train, df_train, lambda_value=30)


if __name__ == "__main__":
    main()
```

A6(a)

We have regularized negative log-likelihood function:

$$J(w,b) = \frac{1}{n}\sum_{i=1}^{n} \log\Big(1 + \exp\big(-\,y_i\big(b + x_i^T w\big)\big)\Big) \;+\; \lambda\,\|w\|_2^2.$$

Gradient with respect to w

For a single example (x_i, y_i), define:

$$\ell_i(w,b) = \log\Big(1 + \exp\big(-\,y_i\,(b + x_i^T w)\big)\Big).$$

Let

$$z_i = -\,y_i\,(b + x_i^T w).$$

Then

$$\ell_i(w,b) = \log\big(1 + \exp(z_i)\big).$$

Step 1: Differentiate $\log(1+\exp(z_i))$ w.r.t. w Using the chain rule:

$$\frac{d}{dw}\,\log\big(1 + \exp(z_i)\big) = \frac{1}{1+\exp(z_i)} \cdot \frac{d}{dw}\,\exp(z_i).$$

Since

$$z_i = -\,y_i\big(b + x_i^T w\big),$$

we have

$$\frac{dz_i}{dw} = -\,y_i\,\frac{d}{dw}\big(b + x_i^T w\big) = -\,y_i\,x_i.$$

Thus,

$$\frac{d}{dw}\,\exp(z_i) = \exp(z_i)\,\big(-\,y_i\,x_i\big).$$

Substituting back:

$$\frac{d}{dw}\,\log\big(1 + \exp(z_i)\big) = \frac{1}{1+\exp(z_i)}\,\exp(z_i)\,\big(-\,y_i\,x_i\big).$$

Rewriting in terms of the original variable,

$$\frac{d}{dw}\,\log\Big(1 + \exp\big(-\,y_i\,(b + x_i^T w)\big)\Big) = -\,y_i\,x_i\,\frac{\exp\big(-y_i\,(b + x_i^T w)\big)}{1 + \exp\big(-y_i\,(b + x_i^T w)\big)}.$$

Step 2: We observe that

$$\frac{\exp\big(-y_i\,(b+x_i^T w)\big)}{1+\exp\big(-y_i\,(b+x_i^T w)\big)} = \frac{\big(1+\exp\big(-y_i(b+x_i^T w)\big)\big)-1}{1+\exp\big(-y_i(b+x_i^T w)\big)} = 1-\frac{1}{1+\exp\big(-y_i(b+x_i^T w)\big)}.$$

Since $\mu_i(w,b)$ is defined as

$$\mu_i(w,b) = \frac{1}{1+\exp\big(-\,y_i\,(b+x_i^T w)\big)},$$

it follows that

$$\frac{\exp\big(-y_i\,(b+x_i^T w)\big)}{1+\exp\big(-y_i\,(b+x_i^T w)\big)} = 1-\mu_i(w,b).$$

Hence,

$$-\,y_i\,x_i\,\frac{\exp\big(-y_i(b+x_i^T w)\big)}{1+\exp\big(-y_i(b+x_i^T w)\big)} = -\,y_i\,x_i\,\big[\,1-\mu_i(w,b)\big].$$

Step 3: Sum over all i and include regularization The total cost is

$$J(w,b) = \frac{1}{n}\sum_{i=1}^{n}\ell_i(w,b) + \lambda\,\|w\|_2^2.$$

Therefore,

$$\frac{\partial}{\partial w}\Big[\tfrac{1}{n}\sum_{i=1}^{n}\ell_i(w,b)\Big] = \frac{1}{n}\sum_{i=1}^{n}\Big[-\,y_i\,x_i\big(1-\mu_i(w,b)\big)\Big].$$

The derivative of $\lambda\,\|w\|_2^2$ with respect to w is $2\,\lambda\,w$. Combining these gives

$$\boxed{\nabla_w J(w,b) = \frac{1}{n}\sum_{i=1}^{n}\Big[-\,y_i\,x_i\,\big(1-\mu_i(w,b)\big)\Big] \;+\; 2\,\lambda\,w.}$$

■

Gradient with respect to b

Step 1: Differentiate w.r.t. b Again let

$$z_i = -\,y_i\,(b+x_i^T w).$$

Then

$$\frac{dz_i}{db} = -\,y_i,$$

which implies

$$\frac{d}{db}\exp(z_i) = \exp(z_i)\,\big(-\,y_i\big).$$

Thus,

$$\frac{d}{db}\log\big(1+\exp(z_i)\big) = \frac{1}{1+\exp(z_i)}\,\exp(z_i)\,\big(-y_i\big).$$

Rewriting in original variables,

$$\frac{d}{db}\,\log\Big(1+\exp\big(-y_i(b+x_i^T w)\big)\Big) = -\,y_i\,\frac{\exp\big(-y_i(b+x_i^T w)\big)}{1+\exp\big(-y_i(b+x_i^T w)\big)}.$$

Using the same observation as before,

$$= -\,y_i\,\big[\,1-\mu_i(w,b)\big].$$

Step 2: Sum over i and simplify Hence,

$$\frac{\partial}{\partial b}\Big[\tfrac{1}{n}\sum_{i=1}^{n}\ell_i(w,b)\Big] = -\,\frac{1}{n}\,\sum_{i=1}^{n} y_i\,\big[\,1-\mu_i(w,b)\big].$$

Since $\lambda\,\|w\|_2^2$ does not involve b,

$$\nabla_b J(w,b) = -\,\frac{1}{n}\,\sum_{i=1}^{n} y_i\,\big[\,1-\mu_i(w,b)\big].$$

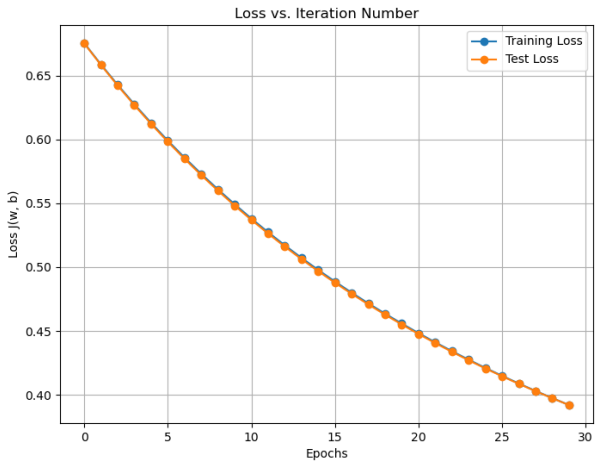
Moving the negative sign inside the summation,

$$\boxed{\nabla_b J(w,b) = \frac{1}{n}\,\sum_{i=1}^{n}\big(\mu_i(w,b)-1\big)y_i.}$$

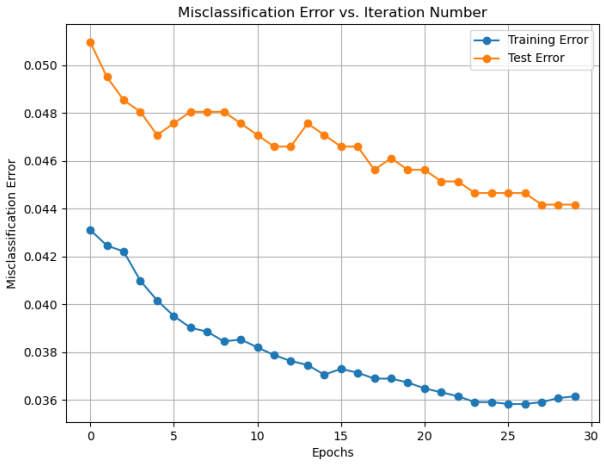
■

A6(b)

(i) Loss vs. Iteration Number

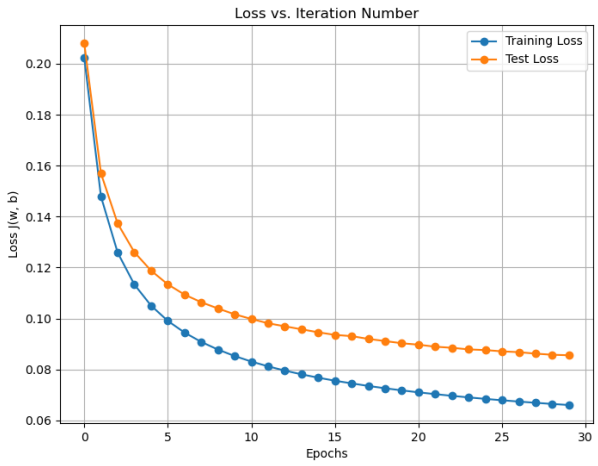


(ii) Misclassification Error vs. Iteration Number

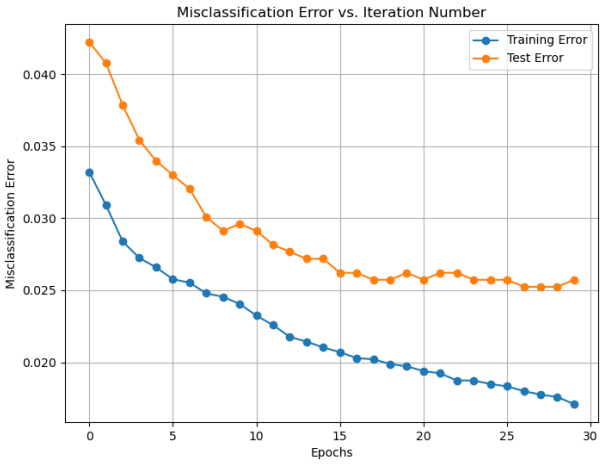


A6(c)

(i) Loss vs. Iteration Number



(ii) Misclassification Error vs. Iteration Number



A6 Code Implementation

```
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")
```

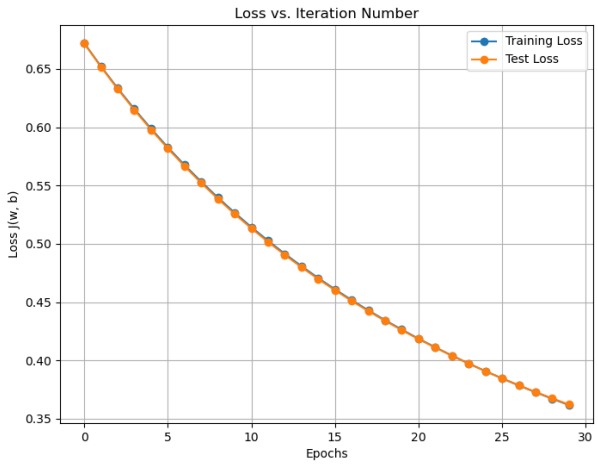
```
    def predict(self, X: np.ndarray) -> np.ndarray:
        """Given X, weight and bias predict values of y.

        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]
```

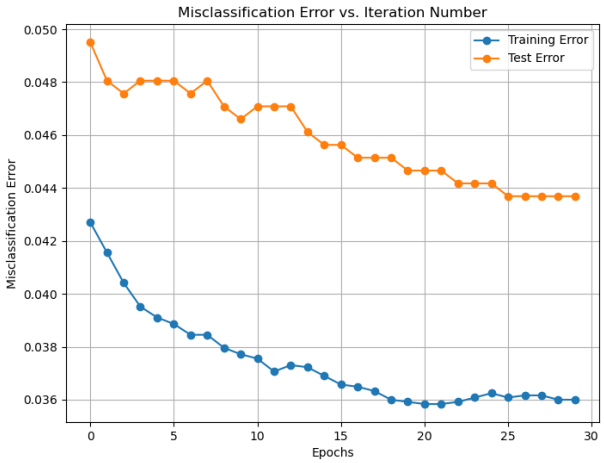
```
                y_train_shuffled = y_train[indices]
                for i in range(num_batches):
                    start = i * batch_size
                    end = min(start + batch_size, n)
                    X_batch = X_train_shuffled[start:end]
                    y_batch = y_train_shuffled[start:end]
                    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()
```

A6(d)

(i) Loss vs. Iteration Number



(ii) Misclassification Error vs. Iteration Number



A7

I spent about 25 hours on this homework.