Solutions to HW-1

- Arjun Parasuram Prasad

1: Polynomial Regression as linear least squares

1

$$b = [b_0, b_1, b_2 \dots b_d]^T \quad (b \in \mathbb{R}^{d+1})$$

$$X = \begin{bmatrix} 1 & x_1^1 & x_1^2 & \dots & x_1^d \\ 1 & x_2^1 & x_2^2 & \dots & x_2^d \\ \vdots & & & & \\ 1 & x_m^1 & x_m^2 & \dots & x_m^d \end{bmatrix} \quad (X \in \mathbb{R}^{m \times (d+1)})$$

$$y = [y_1, y_2 \dots, y_m]^T \quad (y \in \mathbb{R}^m)$$

please turn over

1.1

$\boxed{\text{Part 1}}$

<u>To show:</u>

Empirical Risk minimizer $\hat{b}$ is given by

$$\hat{b} = \arg\min_{b} \|Xb - y\|^2$$

please turn over

Proof:

Hypothesis Space

$H_d = \{ \mathbb{R}^d \rightarrow \mathbb{R} \mid f_b(x) = xb,\ b \in \mathbb{R}^{d+1},\ x \in \mathbb{R}^{1\times(d+1)} \}$

Risk

$R(f_b) = E\,\ell(f_b(x), y)$

taking the loss function as sq. distance

$R(f_b) = E(xb - y)^2 \quad \{x \in \mathbb{R}^{1\times(d+1)}, y \in \mathbb{R}\}$

$\Rightarrow$ for m samples, the empirical risk can be written as:

$$\hat{R}(f_b) = \frac{1}{m} \sum_{i=1}^{m} (x_i b - y_i)^2$$

$\Rightarrow$ The empirical risk minimizer is defined as:

$$\hat{f}_n \in \arg\min_{f \in \mathcal{F}} \hat{R}(f)$$

please turn over

$$\Rightarrow \quad \hat{f}_n \in \arg\min_{f \in \mathbb{F}} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i)$$

$\therefore$ for our case

$$\hat{f}_{b_m} \in \arg\min_{f_b \in \mathcal{H}_d} \frac{1}{m} \sum_{i=1}^{m} \ell(f(x_i), y_i)$$

$\Rightarrow$ as f_b is a function in $\mathcal{H}_d$ parametrized by $b = [b_0, \dots, b_d]^T$
we can rewrite the above expression as:

$$\hat{b} \in \arg\min_{b} \frac{1}{m} \sum_{i=1}^{m} (x_i b - y_i)^2$$

rewriting this in matrix form we get

$$\hat{b} = \arg\min_{b} \left(\frac{1}{m} \| Xb - y \|_2^2 \right)$$

$$\Rightarrow \quad \boxed{\hat{b} = \arg\min_{b} \| Xb - y \|_2^2}$$

[where $X \in \mathbb{R}^{m \times (d+1)}$, $b \in \mathbb{R}^{d+1}$, $y \in \mathbb{R}^{m}$]

1.2

$\boxed{\text{Part 2}}$

Given:

$N > d$, X is full rank

To prove:

$$\hat{b} = (X^T X)^{-1} X^T y$$

Proof:

$$\hat{b} = \arg\min_{b} \underbrace{\|Xb - y\|_2^2}_{\text{loss } f^n\ \ell}$$

To get $\hat{b}$, we shall differentiate ℓ wrt b and equate it to 0

please turn over

$$\Rightarrow \nabla_b(\ell) = 0 = \nabla_b\left(\frac{1}{m}\|Xb - y\|_2^2\right)$$

$$\Rightarrow 0 = \frac{1}{m}\nabla_b\left((Xb-y)^T(Xb-y)\right)$$

$$\Rightarrow 0 = \nabla_b\left(b^TX^TX - 2y^TXb + y^Ty\right)$$

$$0 = \nabla_b\, b^TX^TX - \nabla_b\, 2y^TXb + \nabla_b\, y^Ty$$

$$= 2X^TX\hat{b} - 2X^Ty$$

$$\Rightarrow \cancel{2}X^TX\hat{b} = \cancel{2}X^Ty$$

$$\Rightarrow \boxed{\hat{b} = (X^TX)^{-1}X^Ty}$$

why should $N > d$?

$\Rightarrow$ if $N \lesssim d$ we might have ∞ solutions or no solutions at all.

[as our system will have more variables than equations]

please turn over

why should X be full rank?

$\Rightarrow$ when $N > d$, X is full rank when its rows or columns are linearly independent.

$\therefore$ multicollinearity will not occur.

$\Rightarrow$ $\therefore$ full rank of X ensures that $X^T X$ is invertible, allowing us to get a unique sol. for $\hat{b}$.

please turn over

2: Gradient descent for ridge/linear regression

1

```
###########################################
### 1. Feature Normalization
def feature_normalization(train, test):
    """Rescale the data so that each feature in the training set is in
    the interval [0,1], and apply the same transformations to the test
    set, using the statistics computed on the training set.

    Args:
        train - training set, a 2D numpy array of size(num_instances,
num_features)
        test - test set, a 2D numpy array of size(num_instances,
num_features)

    Returns:
        train_normalized - training set after normalization
        test_normalized - test set after normalization
    """
    # for every feature the affine transformation will be
    # subract the mean and divide by the range
    min_vector = np.min(train, axis=0)
    range_vector = np.max(train, axis=0) - np.min(train, axis=0)
    range_vector[range_vector == 0] = 1 # this handles the case where the
min = max,
    # i.e. the features are constant and range = 0. This way we avoid 0/0
division when the features are constant.
    # train - min_vector  = 0 as well and the feature is thus ignored.
    train_normalized = (train - min_vector)*(1/(range_vector))
    test_normalized = (test - min_vector)*(1/(range_vector))
    return train_normalized, test_normalized
```

2

2

$\boxed{2}$

$$J(\theta) = \frac{1}{m}\left[(X\theta - y)^T(X\theta - y)\right]$$

please turn over

3

$\boxed{3}$ $$\nabla_\theta(J(\theta)) = \nabla_\theta\left[\frac{1}{m}(X\theta-y)^T(X\theta-y)\right]$$

$$\Rightarrow \nabla_\theta(J(\theta)) = \frac{1}{m}\left[2X^TX\theta - 2X^Ty\right]$$

$$\boxed{\nabla_\theta(J(\theta)) = \frac{2}{m}\left[X^TX\theta - X^Ty\right]}$$

4

$\boxed{4}$ $$\theta := \theta - \eta\nabla_\theta(J(\theta))$$

$$\rightarrow \boxed{\theta := \theta - \eta\left[\frac{2}{m}(X^TX\theta - X^Ty)\right]}$$

please turn over

5

```
########################################
### 5. The square loss function
def compute_square_loss(X, y, theta):
    """
    Given a set of X, y, theta, compute the average square loss for
predicting y with X*theta.

    Args:
        X - the feature vector, 2D numpy array of size(num_instances,
num_features)
        y - the label vector, 1D numpy array of size(num_instances)
        theta - the parameter vector, 1D array of size(num_features)

    Returns:
        loss - the average square loss, scalar
    """
    # number of training samples
    m = len(y)
    J = (1/m)*(((X @ theta) - y).T @ ((X @ theta) - y))
    # convert 1x1 numpy matrix to a scalar value
    return J.item()
```

6

```
########################################
### 6. The gradient of the square loss function
def compute_square_loss_gradient(X, y, theta):
    """
    Compute the gradient of the average square loss(as defined in
compute_square_loss), at the point theta.

    Args:
        X - the feature vector, 2D numpy array of size(num_instances,
num_features)
        y - the label vector, 1D numpy array of size(num_instances)
        theta - the parameter vector, 1D numpy array of size(num_features)

    Returns:
        grad - gradient vector, 1D numpy array of size(num_features)
    """
    m = len(y)
    grad_J = (2/m) * (((X.T @ X) @ theta) - (X.T @ y))
    return grad_J
```

```
please turn over
```

7

```
########################################
### 7. Gradient checker
def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):
    true_gradient = compute_square_loss_gradient(X, y, theta) #The true 
gradient
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features) #Initialize the gradient we 
approximate


    E = np.eye(num_features)

    theta_plus_epsilon_e = theta + (epsilon*E)
    theta_minus_epsilon_e = theta - (epsilon*E)

    # trying out list comprehension for more optimized compute
    approx_grad = (1/(2*epsilon)) * 
(np.array([compute_square_loss_gradient(X, y, i) \
        for i in theta_plus_epsilon_e]) - 
np.array([compute_square_loss_gradient(X, y, i) \
            for i in theta_minus_epsilon_e]))
    return tolerance >= np.linalg.norm(approx_grad - true_gradient)
```

please turn over

8

```
########################################
### 8. Batch gradient descent
def batch_grad_descent(X, y, alpha=0.1, num_step=1000, grad_check=False):
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_step + 1, num_features))  #Initialize
theta_hist
    loss_hist = np.zeros(num_step + 1)  #Initialize loss_hist
    theta = np.zeros(num_features)  #Initialize theta
    loss_hist[0] = 1e64
    # theta gets updated as theta := theta - alpha*grad(Jtheta)
    for i in range(1, num_step + 1):
        if (grad_check == True):
            error = np.sum(grad_checker(X, y, theta))
            if error == True:
                print("at step i = " + str(i) + " we find that gradient has
failed to compute successfully. \
                    exiting program")
                return
        theta = theta - alpha*(compute_square_loss_gradient(X, y, theta))
        theta_hist[i:] = theta
        loss_hist[i] = compute_square_loss(X, y, theta)
        # print("loss: " + str(loss_hist[i]))
    return theta_hist, loss_hist
```

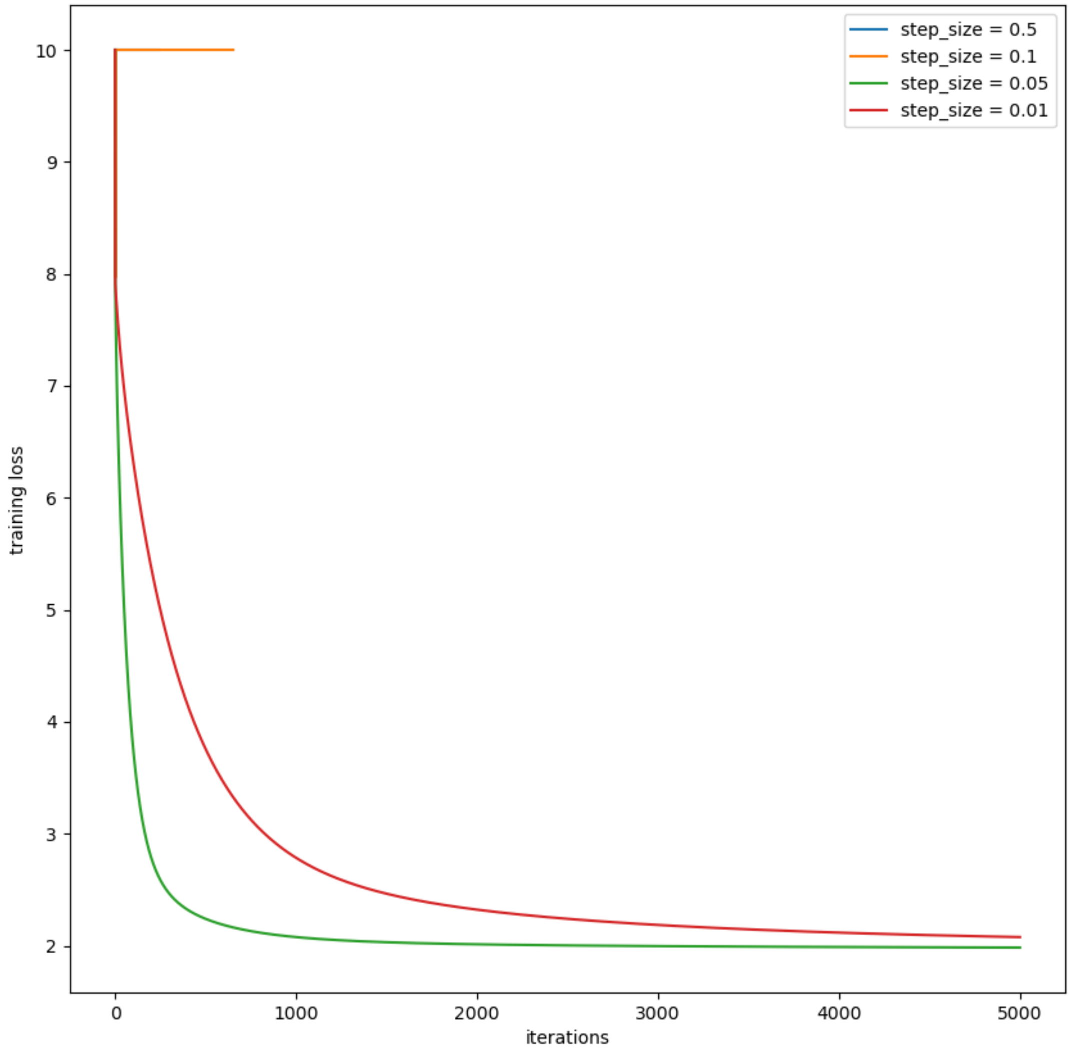
please turn over

9

```
def plot_training_loss_vs_iterations_for_multiple_step_sizes(X_train,
y_train, step_sizes = [0.5, 0.1, 0.05, 0.01],\
    max_iterations = 1000, loss_cap = 20):
    plt.figure(figsize=(10, 10))
    for step_size in step_sizes:
        theta_hist, loss_hist = batch_grad_descent(X_train, y_train, alpha=
step_size, num_step= max_iterations, \
            grad_check= False)
        loss_hist[loss_hist > loss_cap] = loss_cap
        plt.plot((range(0, len(loss_hist))), loss_hist, label = "step_size
= " + str(step_size))
    plt.xlabel("iterations")
    plt.ylabel("training loss")
    plt.legend()
    plt.show()
```

please turn over

- output



- findings
  - Step size = 0.05, converges the fastest.
  - Step size = 0.01 converges slightly slower than step size = 0.05
  - Step sizes = [0.1, 0.5] diverge
- NOTE:
  - I have capped the max loss in the plot to 10 to showcase values more clearly

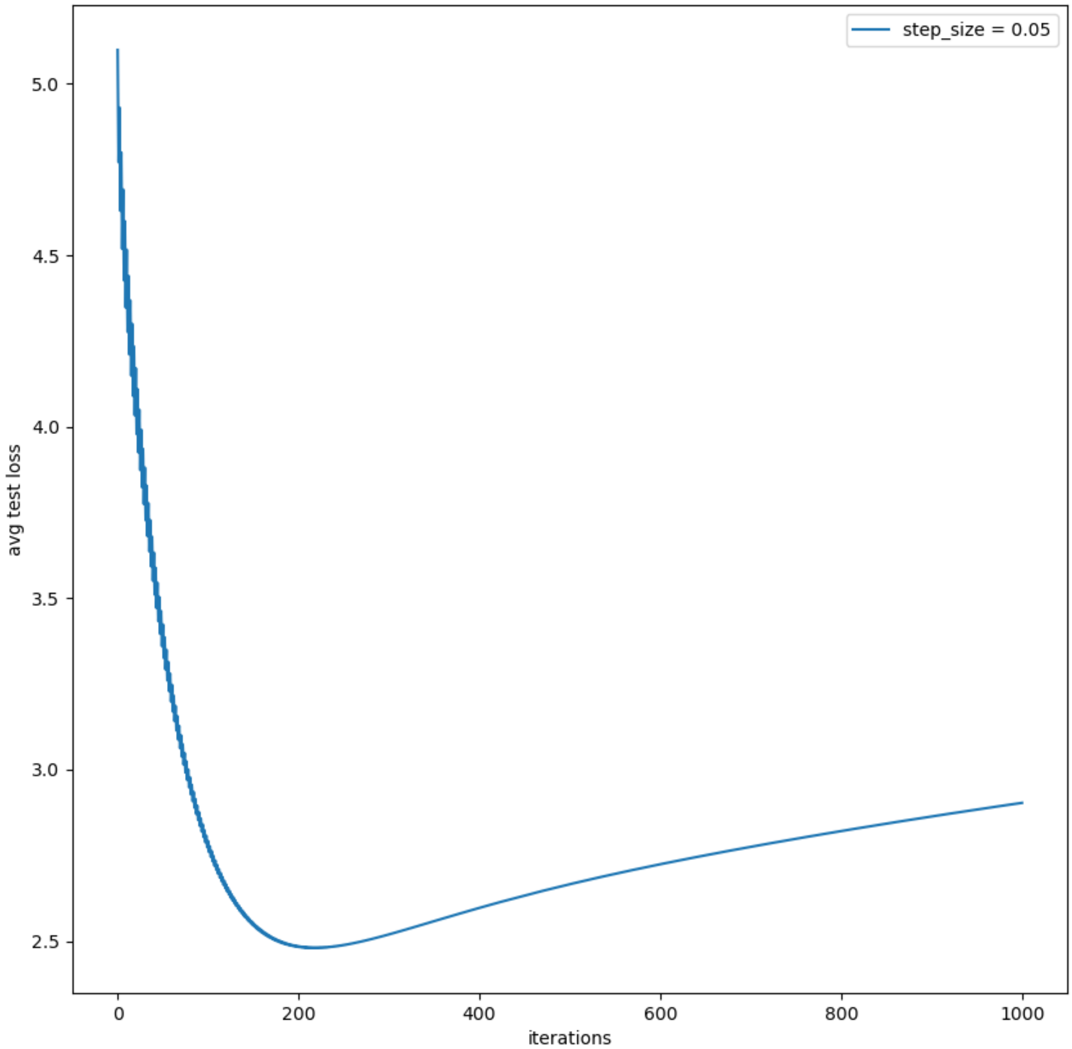
`please turn over`

10

```
#######################################
def plot_test_loss_vs_iterations(X_train, X_test, y_train, y_test,
step_size = 0.05, \
    max_iterations = 1000, loss_cap = 20):
    plt.figure(figsize=(10, 10))
    theta_hist, training_loss_hist = batch_grad_descent(X_train, y_train,
alpha= step_size, \
        num_step= max_iterations, grad_check= False)
    testing_loss_hist = np.zeros(max_iterations + 1)
    for i in range(max_iterations + 1):
        testing_loss_hist[i] = compute_square_loss(X_test, y_test, theta=
theta_hist[i])
    testing_loss_hist[testing_loss_hist > loss_cap] = loss_cap
    plt.plot((range(0, max_iterations + 1)), testing_loss_hist, label =
"step_size = " + str(step_size))
    plt.xlabel("iterations")
    plt.ylabel("avg test loss")
    plt.legend()
    plt.show()
```

please turn over

- output



please turn over

11

$\boxed{11}$

$$J_\lambda(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x_i) - y_i\right)^2 + \lambda\theta^T\theta$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x_i) - y_i\right)^2 + \lambda\|\theta\|^2$$

$\nabla_\theta(J(\theta)) = \frac{2}{m}\sum_{i=1}^{m}\left(h_\theta(x_i) - y_i\right) + 2\lambda\sum_{i=0}^{d}\theta_i$, which can be rewritten as

$$\boxed{\nabla_\theta(J_\lambda(\theta)) = \frac{2}{m}\left[X^TX\theta - X^Ty\right] + 2\lambda\theta}$$

12

```
########################################
### 12. The gradient of regularized batch gradient descent
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
    m = len(y)
    grad_J = (2/m) * (((X.T @ X) @ theta) - (X.T @ y)) + 2*lambda_reg*theta
    return grad_J
```

please turn over

13

```
########################################
### 13. Regularized batch gradient descent
def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=10**-2,
num_step=1000):
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.zeros(num_features) #Initialize theta
    theta_hist = np.zeros((num_step+1, num_features)) #Initialize
theta_hist
    loss_hist = np.zeros(num_step+1) #Initialize loss_hist
    for i in range(1, num_step + 1):
        theta = theta - alpha*(compute_regularized_square_loss_gradient(X,
y, theta, lambda_reg))
        theta_hist[i:] = theta
        loss_hist[i] = compute_square_loss(X, y, theta) + lambda_reg *
((theta.T @ theta).item())
        # print("loss: " + str(loss_hist[i]))
    return theta_hist, loss_hist
```

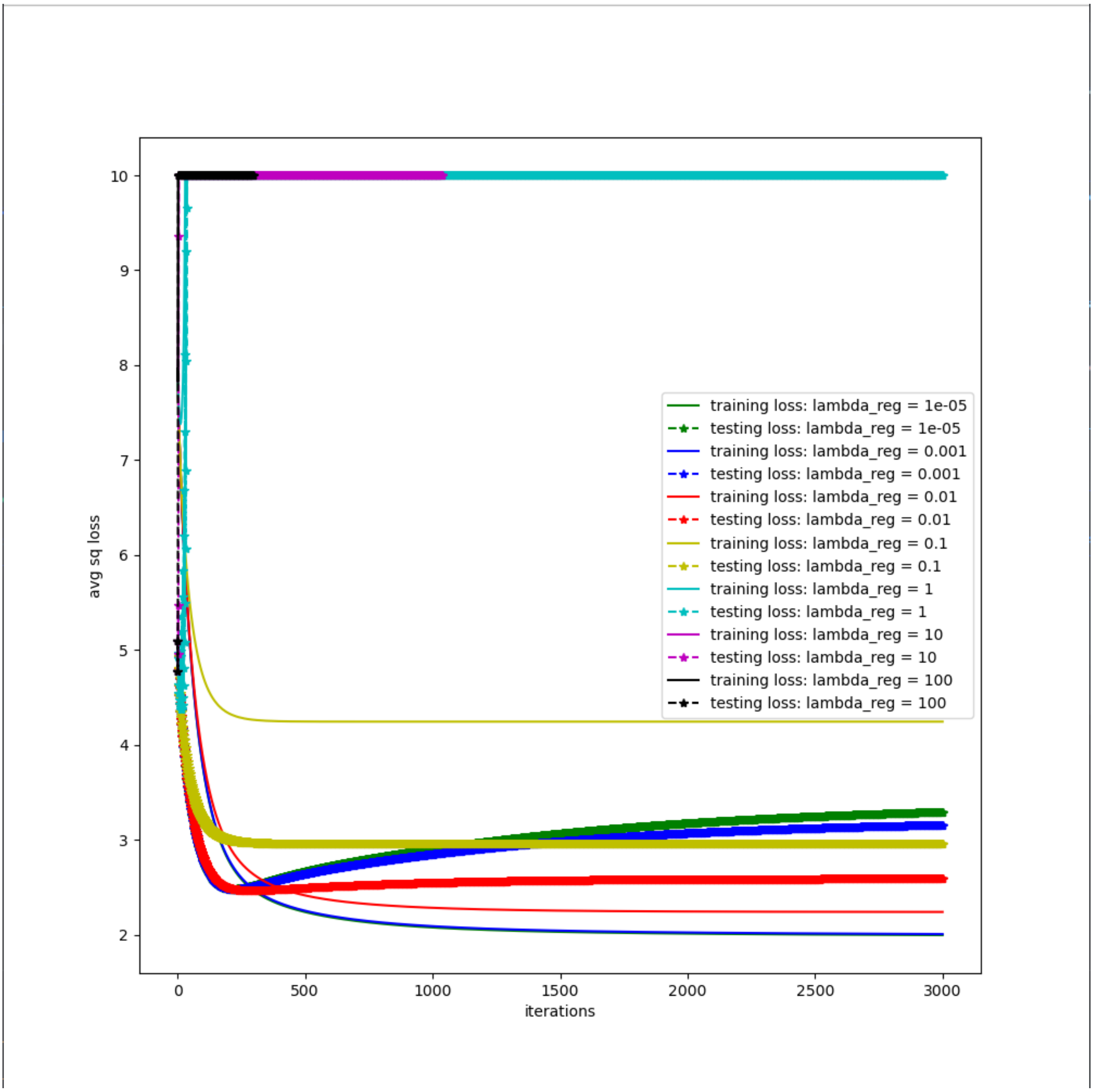
`please turn over`

14

```
def plot_loss_vs_iterations_for_multiple_lambda_reg(X_train, X_test,
y_train, y_test, step_size = 0.05, \
    lambda_regs = [1e-7, 1e-5, 1e-3, 1e-1, 1, 10, 100], max_iterations =
1000, loss_cap = 20):
    plt.figure(figsize=(10, 10))
    colors = ['g', 'b', 'r', 'y', 'c', 'm', 'k']
    color_itr = 0
    for lambda_reg in lambda_regs:
        theta_hist, training_loss_hist = regularized_grad_descent(X_train,
y_train, alpha= step_size, \
            num_step= max_iterations, lambda_reg=lambda_reg)

        train_avg_sq_loss = np.array([compute_square_loss(X_train, y_train,
theta) for theta in theta_hist])
        train_avg_sq_loss[train_avg_sq_loss > loss_cap] = loss_cap

        test_avg_sq_loss = np.array([compute_square_loss(X_test, y_test,
theta) for theta in theta_hist])
        test_avg_sq_loss[test_avg_sq_loss > loss_cap] = loss_cap

        plt.plot(range(0, max_iterations + 1), train_avg_sq_loss, label =
"training loss: lambda_reg = " + \
            str(lambda_reg), color = colors[color_itr%(len(colors))])
        plt.plot(range(0, max_iterations + 1), test_avg_sq_loss, label =
"testing loss: lambda_reg = " + \
            str(lambda_reg), marker = '*', color = colors[color_itr%
(len(colors))], linestyle = '--')
        color_itr += 1
    plt.xlabel("iterations")
    plt.ylabel("avg sq loss")
    plt.legend()
    plt.show()
```

please turn over

- output



- findings
  - It is observed that λ = [1e-2, 1e-3, 1e-5] exhibit certain degree of overfitting
  - The best test set performance is when we keep λ = 1e-2
  - λ = 1e-1 does not exhibit any overfitting and in fact has test error lower than the training error Its test error also doesn't increase even on increasing the number of iterations upto 3000
  - It is only at about 1500 iterations do we see that the error due overfitting (for models having λ = [1e-5, 1e-3]) exceed the model with 1e-1 = 1e-1 (which never overfits)

- NOTE
  - loss has been capped to 10 to showcase plot value in a better manner

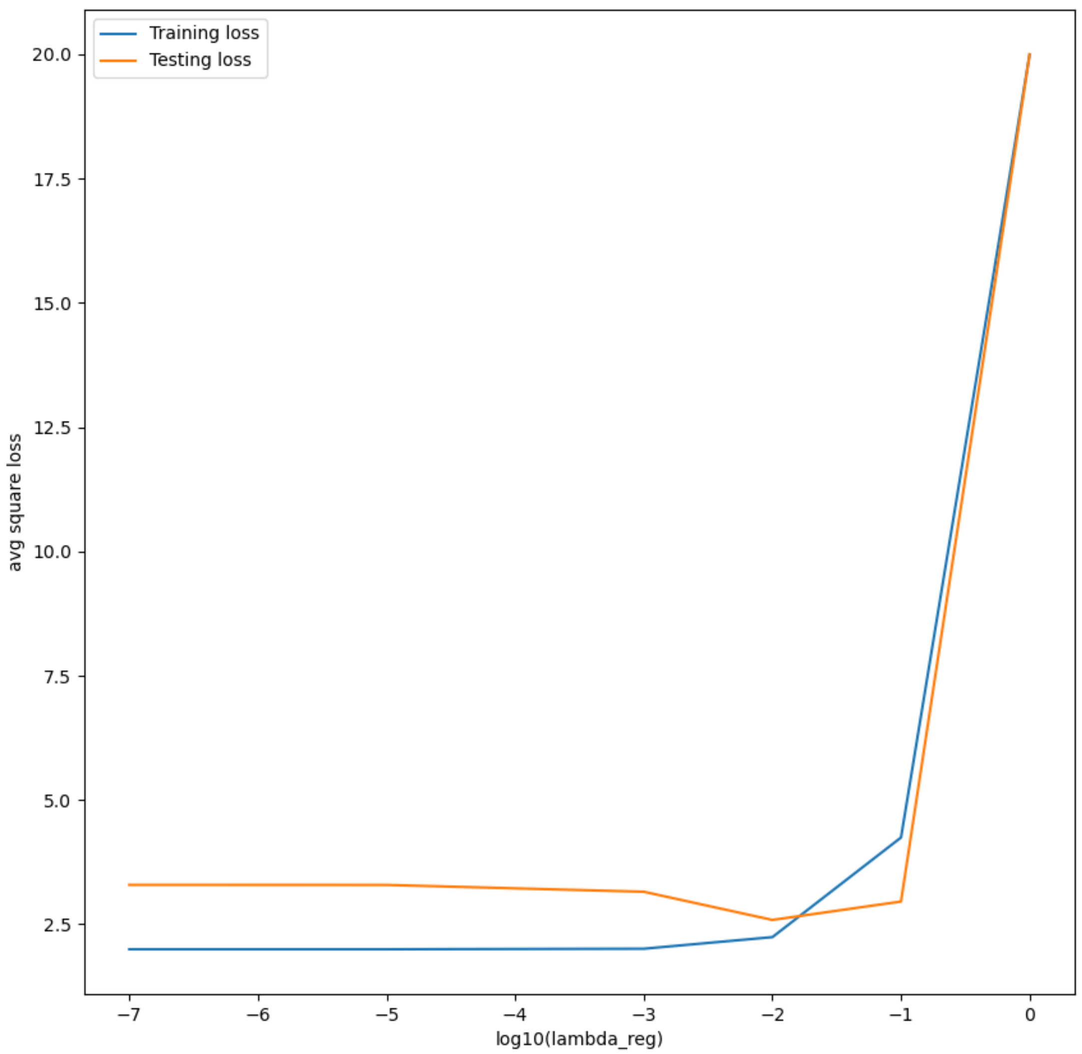
`please turn over`

15

```
###########################################
def plot_loss_vs_lambda_reg(X_train, X_test, y_train, y_test, \
    lambda_reg = [1e-7, 1e-5, 1e-3, 1e-2, 1e-1, 1e0, 10, 100, 0], step_size
= 0.05, iterations = 1000):
    training_loss = []
    testing_loss = []
    log_lambda_reg = []
    for i in lambda_reg:
        theta_hist, loss_hist = regularized_grad_descent(X_train, y_train,
step_size, i, iterations)
        training_loss.append(compute_square_loss(X_train, y_train,
theta_hist[-1]))
        if(training_loss[-1] > 20):
            training_loss[-1] = 20
        testing_loss.append(compute_square_loss(X_test, y_test,
theta_hist[-1]))
        if(testing_loss[-1] > 20):
            testing_loss[-1] = 20
        log_lambda_reg.append(np.log10(i))

    plt.figure(figsize=(10, 10))
    plt.plot(log_lambda_reg, training_loss, label = "Training loss")
    plt.plot(log_lambda_reg, testing_loss, label = "Testing loss")
    plt.ylabel("avg square loss")
    plt.xlabel("log10(lambda_reg)")
    plt.legend()
    plt.show()
```

please turn over

- output



- findings
    - I would choose λ = 1e-2 as it has lowest test error, and its degree of overfitting is minimal.

`please turn over`

16

$\boxed{16}$ $\sigma(z) = \dfrac{1}{1+e^{-\sigma}}$

loss for the data sample (x_i, y_i):

$$l_i(\theta) = \log\left(1+e^{-M_i}\right) \left\{ \begin{array}{l} \text{where} \\ M \text{ is margin} \\ \text{and} \\ M_i = \hat{y}_i y_i \end{array} \right\}$$

$$\Rightarrow l_i(\theta) = \begin{cases} \log\left(1+e^{-h_{\theta,b}(x_i)}\right), & y_i = 1 \\ \log\left(1+e^{+h_{\theta,b}(x_i)}\right), & y_i = -1 \end{cases}$$

This can be rewritten as

$$l_i(\theta) = \begin{cases} \frac{1}{2}\left[(1+y_i)\log\left(1+e^{-h_{\theta,b}(x_i)}\right)+(1-y_i)\log\left(1+e^{h_{\theta,b}(x_i)}\right)\right], & y_i = 1 \\ \frac{1}{2}\left[(1+y_i)\log\left(1+e^{-h_{\theta,b}(x_i)}\right)+(1-y_i)\log\left(1+e^{h_{\theta,b}(x_i)}\right)\right], & y_i = -1 \end{cases}$$

$$\Rightarrow l_i(\theta) = \frac{1}{2}\left[(1+y_i)\log\left(1+e^{-h_{\theta,b}(x_i)}\right) + (1-y_i)\log\left(1+e^{h_{\theta,b}(x_i)}\right)\right]$$

$\therefore$ for m samples, average logistic loss, aka our objective function is

$$\boxed{l(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left[(1+y_i)\log\left(1+e^{-h_{\theta,b}(x_i)}\right) + (1-y_i)\log\left(1+e^{h_{\theta,b}(x_i)}\right)\right]}$$

17

(17) regularizing the loss function with l_1 pen and reg param "α"

$$\boxed{l(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left[(1+y_i)\log\left(1+e^{-h_{\theta,b}(x_i)}\right) + (1-y_i)\log\left(1+e^{h_{\theta,b}(x_i)}\right)\right] + \alpha\|\theta\|_1}$$

please turn over

18

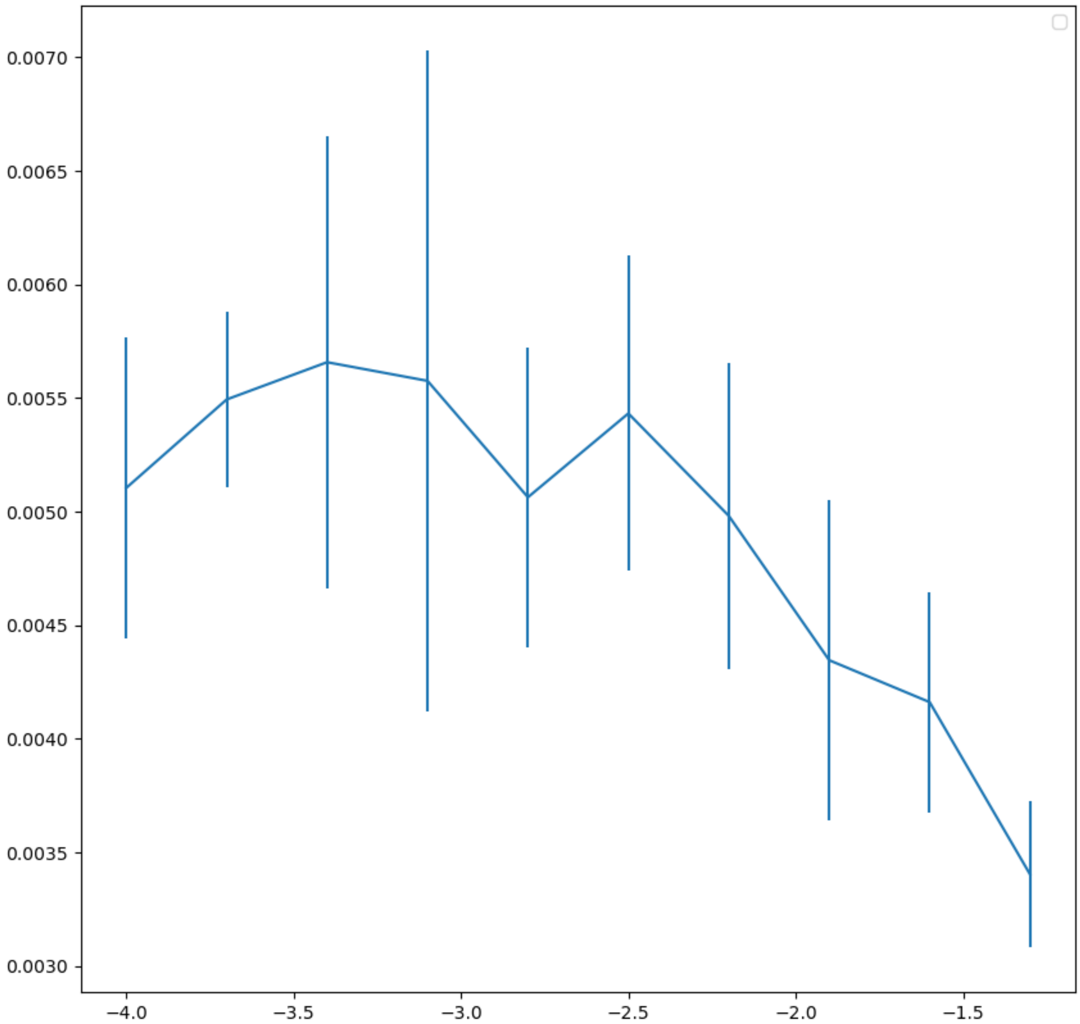
```
def classification_error(clf, X, y):
    return np.sum(clf.predict(X) != y)/X.shape[0]
```

19

```
def classification_error_vs_lambda_reg(X_train, y_train, lambda_reg_lo =
1e-4, lambda_reg_hi = 1e-1):
    log_lambda_reg = []
    mean_error_list = []
    std_dev_error_list = []
    epsilon = 1e-9
    for i in np.arange(np.log10(lambda_reg_lo) + epsilon,
np.log10(lambda_reg_hi) - epsilon, (np.log10(lambda_reg_hi) -
np.log10(lambda_reg_lo))/10): # getting exactly 10 values between the upper
and lower limit
        error_arr = np.zeros(10)
        for j in range(10):
            classifier = SGDClassifier(
                loss = 'log_loss',
                max_iter = 1000,
                tol = 1e-3,
                penalty = 'l1',
                alpha= np.power(10, i),
                learning_rate='invscaling',
                power_t = 0.5,
                verbose = 1,
                eta0 = 0.01
            )
            classifier.fit(X_train, y_train)

            error_arr[j] = classification_error(classifier, X_test, y_test)

        mean_error = np.mean(error_arr)
        std_dev = np.std(error_arr)
        log_lambda_reg.append(i)
        mean_error_list.append(mean_error)
        std_dev_error_list.append(std_dev)

    plt.figure(figsize=(10, 10))
    plt.errorbar(x = log_lambda_reg, y = mean_error_list,
yerr=std_dev_error_list)
    plt.legend()
    plt.show()
    return
```

please turn over

- output



20

- SGDClassifier has shuffles the training sample order when running classifier.fit() . Since stochastic gradient descent considers only one sample at a time, the order of the training samples will affect the manner in which theta gets updated. To account for this randomness, we take the mean loss after 10 iterations of the fitting the model to the training data.

21

- alpha = 1e(-1.3)
  - has the lowest mean error ~ 0.0034
  - has the std deviation ~ (0.0037 - 0.0031)*0.5 = 0.0003
- As the average loss is minimum for this value of alpha, coupled with minimal std. deviation, we can say that this choice of alpha is the best for our model.

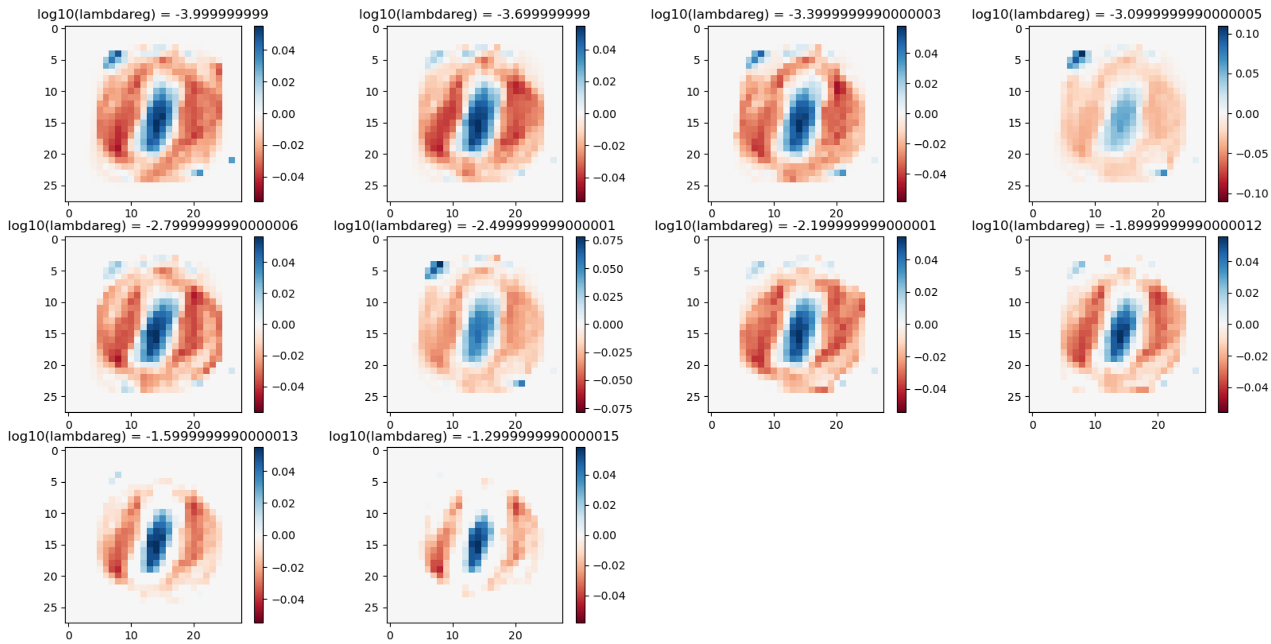
`please turn over`

22

```
def plot_theta_vs_lambda_reg(X_train, y_train, lambda_reg_lo = 1e-4,
lambda_reg_hi = 1e-1):
    plt.figure(figsize=(20, 20))
    epsilon = 1e-9
    idx = 0
    for i in np.arange(np.log10(lambda_reg_lo) + epsilon,
np.log10(lambda_reg_hi) - epsilon, \
        (np.log10(lambda_reg_hi) - np.log10(lambda_reg_lo))/10): \
            # getting exactly 10 values between the upper and lower limit
        classifier = SGDClassifier(
            loss = 'log_loss',
            max_iter = 1000,
            tol = 1e-3,
            penalty = 'l1',
            alpha= np.power(10, i),
            learning_rate='invscaling',
            power_t = 0.5,
            verbose = 0,
            eta0 = 0.01
        )
        classifier.fit(X_train, y_train)
        theta = classifier.coef_.reshape(28, 28)
        scale = np.abs(classifier.coef_).max()
        idx = idx+1
        plt.subplot(4, 4, idx)
        plt.title("log10(lambdareg) = " + str(i))
        plt.imshow(theta, cmap=plt.cm.RdBu, vmax=scale, vmin=-scale)
        plt.colorbar()
    plt.show()
```

please turn over

- output



23

- As the values of alpha (I have named them as lambda_reg in my code) keep increasing the weights get suppressed further, resulting in weights which were of little value (close to 0) got entirely suppresed to 0, For e.g. we note that for log10(lambdareg) = -1.2999999 there is considerably much more white color in the image in contrast to log10(lambdareg) = -3.99999999 (wherein the weights were allowed more freedom to have values and overfit the data)