Homework 2 - DS GA 1003

Selected Solutions

Statistical Learning Theory

Q1. Rexpressing the training Error

$\hat{b} = (X^T X)X^T y$
Train Error $= \frac{1}{N}||X\hat{b} - y||_2^2 = \frac{1}{N}||X(X^T X)^{-1}X^T y - y||_2^2 = \frac{1}{N}||(X(X^T X)^{-1}X^T - I)y||_2^2$
Let, $X(X^T X)^{-1}X^T = A, \epsilon = \sigma 1_n$
Train Error $= \frac{1}{N}||(A - I)y||_2^2 = \frac{1}{N}||(A - I)(Xb + \epsilon)||_2^2$
$= \frac{1}{N}||(A - I)\epsilon + (AXb - Xb)||_2^2 = \frac{1}{N}||(A - I)\epsilon||_2^2$

Q2. Expectation of train error

$E[\frac{1}{N}||(X(X^T X)^{-1}X^T - I)\epsilon||_2^2]$
$= \frac{1}{N}E[||(X(X^T X)^{-1}X^T - I)\epsilon||_2^2]$
$= \frac{1}{N}E[||(A - I)\epsilon||_2^2]$ (Same definition of$A$)
$= \frac{1}{N}E[\epsilon^T (A - I)^T (A - I)\epsilon]$
$= \frac{1}{N}E[\epsilon^T (I - A)\epsilon]$ (Simplifying: $(A - I)^T (A - I) = A^T A + I - 2A = I - A$)
$= \frac{1}{N}E[\epsilon^T \epsilon + \epsilon^T A\epsilon]]$

$$E[\epsilon^T \epsilon] = E[\sum_{i=1}^{N} \epsilon_i \epsilon_i] = \sum_{i=1}^{N} E[\epsilon_i^2] = \sum_{i=1}^{N} (Var[\epsilon_i] - E[\epsilon_i]^2) = \sum_{i=1}^{N} (\sigma^2 - 0) = N\sigma^2$$

$E[\epsilon^T A\epsilon] = E[\sum_{i,j} \epsilon_i a_{ij} \epsilon_j] = \sum_{i,j} a_{ij} E[\epsilon_i \epsilon_j]$
$= \sum_i a_{ii} E[\epsilon_i \epsilon_i]$ (Because, $E[\epsilon_i \epsilon_j] = 0$ if $i \neq j$)
$= \sum_i a_{ii}\sigma^2 = \sum_{i=1}^{N} d_{ii} p_i^T p_i \sigma^2$ (Taking Eigendecomposition of $A = PDP^T$)
$= \sum_{i=1}^{d} d_{ii} p_i^T p_i \sigma^2$ (Eigenvalues are 0 or 1. Exactly d eigenvalues are 1 because the rank of A is d)

$= d\sigma^2$

Hence train error: $E[\frac{1}{N}||(X(X^TX)^{-1}X^T - I)\epsilon||_2^2] = \frac{1}{N}(N-d)\sigma^2$

Q3. From this result, give a reason as to why the training error is very low when d is close to N i.e. when we overfit the data.

If $d \sim N$ then the middle term is very small and expected train error is almost zero i.e. overfitting occurs

Gradient Descent for Ridge(less) Regression

Q4. Modify function feature normalization to normalize all the features to [0, 1]. Can you use numpy's broadcasting here? Often broadcasting can help to simplify and/or speed up your code. Note that a feature with constant value cannot be normalized in this way. Your function should discard features that are constant in the training set.

The following is the updated function. In this particular instance, none of the features were removed because of being constant.

```
def feature_normalization(train, test):
    remove = []
    for i in range(train.shape[1]):
        if len(set(train[:, i])) == 1:
            remove.append(i)

    for i in remove:
        np.delete(train, i, axis=1)
        np.delete(test, i, axis=1)

    #print(remove)

    train_normalized = np.array(train.shape)
    test_normalized = np.array(test.shape)

    train_max_vals = np.max(train, axis=0)
    train_min_vals = np.min(train, axis=0)

    #print(train_max_vals, train_min_vals)

    train_normalized = (train - train_min_vals) / (train_max_vals - train_min_vals)
    test_normalized = (test - train_min_vals) / (train_max_vals - train_min_vals)

    return train_normalized, test_normalized
```

Q5. Let $X \in \mathbb{R}^{m \times (d+1)}$ be the design matrix, where the i'th row of X is x_i. Let $y_i = (y_1, ..., y_m)^T \in \mathbb{R}^{m \times 1}$ be the response. Write the objective function $J(\theta)$ as a matrix/vector expression, without using an explicit summation sign.

Given function $h_\theta(x) = \theta^T x + b$ mapping from $\mathbb{R}^{d+1} \to \mathbb{R}$ and $J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2$

The same can be written as the matrix/vector expression: $\frac{1}{m}(X\theta - y)^T(X\theta - y)$
where matrix $X \in \mathbb{R}^{m \times (d+1)}$ multiplied by weight vector $\theta \in \mathbb{R}^{d+1}$ gives a vector of same dimensionality as y i.e. R^m

Q6. Write down an expression for the gradient of J without using an explicit summation sign

The gradient of function J w.r.t. θ can be written as follows: $\nabla_\theta J(\theta) = \frac{2}{m}(X^T)(X\theta - y)$. Here $X^T \in \mathbb{R}^{d+1 \times m}$ is multiplied by a term with shape same as y i.e. $R^{m \times 1}$ to yield a gradient with shape R^{d+1} i.e. the same as the vector θ itself.

Q7. Write down the expression for updating θ in the gradient descent algorithm for a step size η

$\theta_{\text{new}} = \theta - \eta \nabla_\theta J(\theta) = \theta - \eta \frac{2}{m}(X^T)(X\theta - y)$

Here we take the gradient at the position of the current θ value so it varies after each gradient update.

Q8. Modify the function compute square loss, to compute $J(\theta)$ for a given θ. You might want to create a small dataset for which you can compute $J(\theta)$ by hand, and verify that your compute square loss function returns the correct value.

Below is the modified function:

```
def compute_square_loss(X, y, theta):
    assert X.shape[1] == theta.shape[0], print("Dimensions don't match for X and
        theta - ", X.shape, theta.shape)
    preds = X @ theta
    m = len(y)
    assert len(preds) == len(y), print("Dimensions don't match for
        preds and y - ", preds.shape, y.shape)
    loss = (1.0/m) * ((preds-y).T @ (preds - y)) #* np.sum((preds - y)**2)
    return loss
```

Q9. Modify the function compute square loss gradient, to compute $\nabla_\theta J(\theta)$. You may again want to use a small dataset to verify that your compute square loss gradient function

returns the correct value.

Here's the modified function:

```
def compute_square_loss_gradient(X, y, theta):
    assert X.shape[1] == theta.shape[0], print("Dimensions don't match for X
        and theta - ", X.shape, theta.shape)
    preds = X @ theta
    m = len(y)
    grad = (2.0/m) * ((preds - y) @ X)
    return grad
```

Q10. Complete the function grad_checker according to the documentation of the function given in the skeleton code.py. Alternatively, you may complete the function generic_grad_checker so which can work for any objective function.

```
def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):
    fn_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.zeros(num_features) for i in range(num_features)]
    for i in range(num_features):
        e[i][i] = 1
    approx = []
    for ei in e:
        upper = compute_square_loss(X, y, theta + epsilon*ei)
        lower = compute_square_loss(X, y, theta - epsilon*ei)
        approx.append((upper - lower)/(2.0 * epsilon))

    approx = np.asarray(approx)
    if np.linalg.norm(fn_gradient - approx) > tolerance:
        return False
    return True
```

Q11. Complete 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

    stopped = False
```

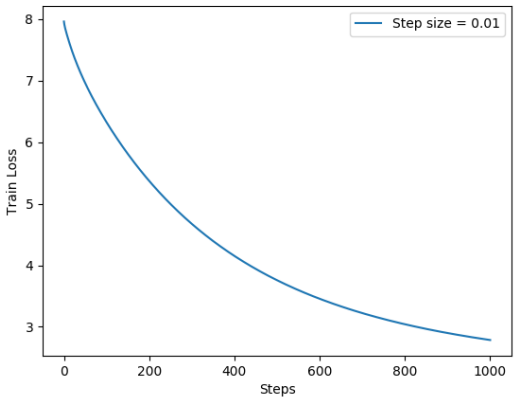
```
    for i in range(num_step+1):
        theta_hist[i] = theta
        loss_hist[i] = compute_square_loss(X, y, theta)
        if i == num_step:
            break
        grad = compute_square_loss_gradient(X, y, theta)
        if grad_check == True and not grad_checker(X, y, theta):
            print("Error, gradient check failed", i)
            stopped = True
            break
        theta = theta - alpha*grad

    if stopped:
        return loss_hist[:i+1], theta_hist[:i+1]
    else:
        return loss_hist, theta_hist
```

Q12. Now let's experiment with the step size. Note that if the step size is too large, gradient descent may not converge. Starting with a step-size of 0.1, try various different fixed step sizes to see which converges most quickly and/or which diverge. As a minimum, try step sizes 0.5, 0.1, .05, and .01. Plot the average square loss on the training set as a function of the number of steps for each step size. Briefly summarize your findings.

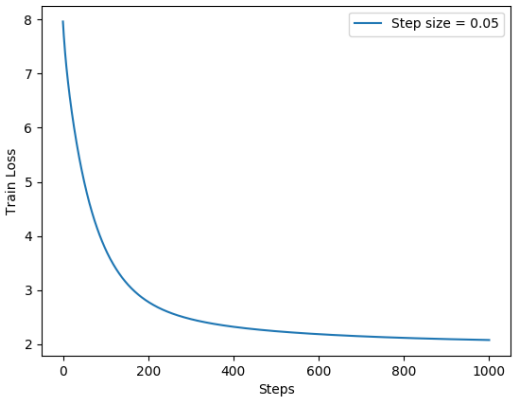
Fig. 1 has the plots for the 4 different fixed step sizes. We can see that the training loss converges for the two smaller values of step size i.e. 0.01 and 0.05. It explodes for step size 0.1 and 0.5 (Note the exponent at the top of the y-axis in plots Fig. 1 (c) and 1 (d)). Additionally we see a faster convergence for step size 0.05 than 0.01, which is to be expected. For the 0.01 case the model still seems to be making reasonable changes to the loss value even after 1000 steps. We select this as the value for Q13. Also note from the x-axis values that Fig. 1 (c) and 1 (d) were truncated early when the gradient check failed because the value exploded.

Q13. For the learning rate you selected above, plot the average test loss as a function of the iterations. You should observe overfitting: the test error first decreases and then increases.

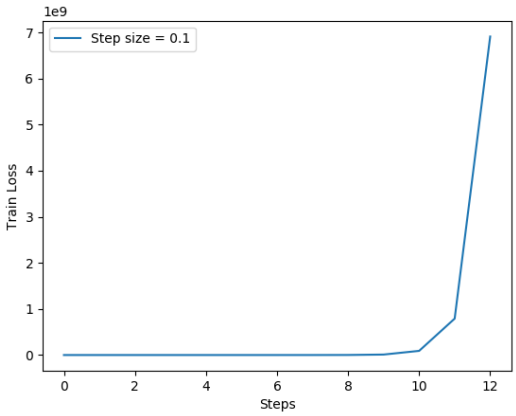
From Q12, we select the value of step size 0.05. Fig. 2 is the required figure. We observe overfitting as test loss first decreases to a minimum after around 200 train steps and then then begins to increase again.



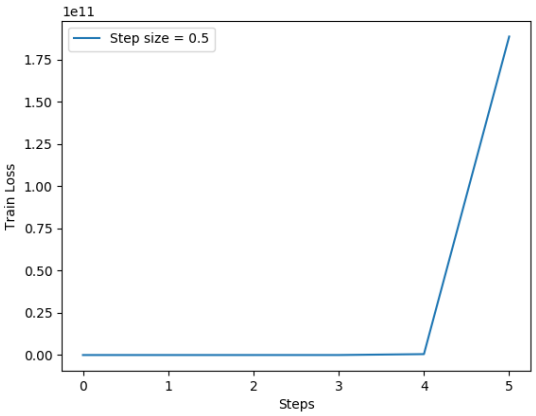
(a)



(b)



(c)



(d)

Figure 1: Plot of train loss as number of steps increase for different fixed step sizes.

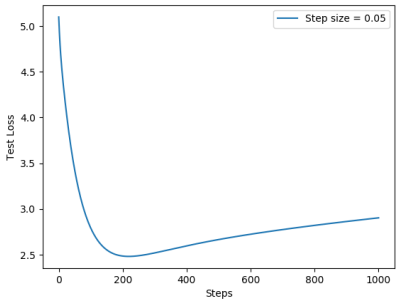


Figure 2: Plot of test loss as number of train steps increase for step size 0.05. We observe overfitting as test loss first decreases then begins to increase again.

Q14. Compute the gradient of $J_\lambda(\theta)$ and write down the expression for updating θ in the gradient descent algorithm. (Matrix/vector expression, without explicit summation)

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

$$\nabla_\theta J(\theta) = \frac{2}{m}(X^T)(X\theta - y) + 2\lambda\theta$$

$$\theta_{\text{new}} = \theta - \eta\nabla_\theta J(\theta) = \theta - \eta\left(\frac{2}{m}(X^T)(X\theta - y) + 2\lambda\theta\right)$$

Q15. Implement compute_regularized_square_loss_gradient.

```
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
    assert X.shape[1] == theta.shape[0], print("Dimensions don't match for X
        and theta - ", X.shape, theta.shape)
    preds = X @ theta
    m = len(y)
    grad = ((2.0/m) * ((preds - y) @ X)) + (2 * lambda_reg * theta)
    return grad
```

Q16. Implement regularized_grad_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(num_step+1):
        theta_hist[i] = theta
        loss_hist[i] = compute_square_loss(X, y, theta)
        if i == num_step:
            break
        grad = compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)
        theta = theta - alpha*grad

    return loss_hist, theta_hist
```

Q17. Choosing a reasonable step-size, plot training average square loss and the test average square loss (just the average square loss part, without the regularization, in each case) as a function of the training iterations for various values of λ. What do you notice in terms of overfitting?

Fig. 3 and 4 are the train and test loss as a function of train steps for a variety of λ values. Note that we truncated the values for $\lambda = 1, 10, 100$ because the loss values were very large. We notice that the models overfit for the very small values of $\lambda < 0.01$ because while train loss stays really low, test loss achieves a minimum and then climbs back up (Fig. 4). For $\lambda = 0.01$ the model test loss converges and stays at that value. When there is too much regularization i.e. $\lambda = 0.1$, the test loss never reaches the same level as the others. (Note that we used the same step size of 0.05 in all of the above)

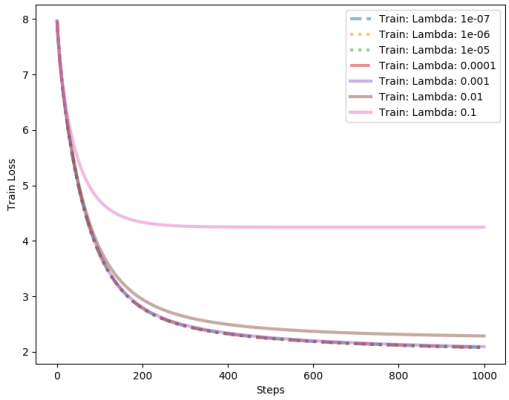


Figure 3: Train loss as a function of train steps for various values of λ

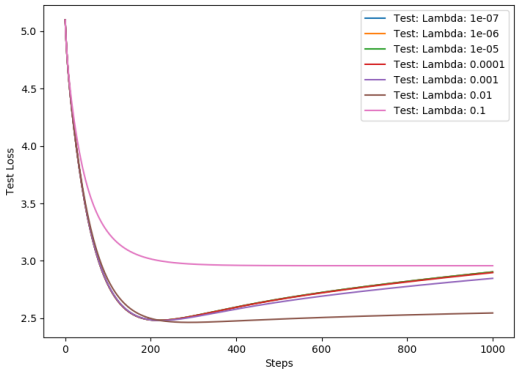


Figure 4: Test loss as a function of train steps for various values of λ

Q18. Plot the training average square loss and the test average square loss at the end of training as a function of λ. You may want to have $log(\lambda)$ on the x-axis rather than λ. Which value of λ would you choose ?

Fig. 5 is the required plot. Note that we stop at $\lambda = 10^{-1}$ because the other values tend to explode. We would select the value of $\lambda = 0.01(10^{-2})$ because it has the lowest test loss

value. (Note that here we select and use step size = 0.05)

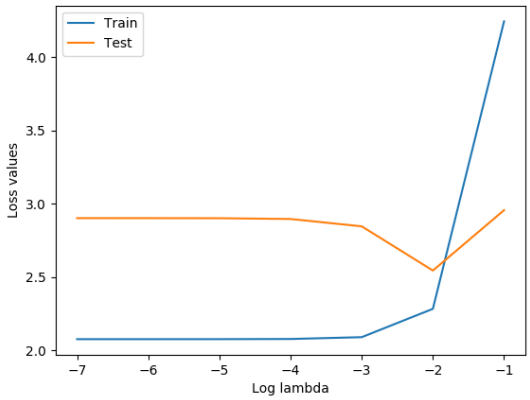


Figure 5: Train and Test loss at the end of training as a function of $\log \lambda$

Q19. Another heuristic of regularization is to early-stop the training when the test error reaches a minimum. Add to the last plot the minimum of the test average square loss along training as a function of λ. Is the value λ you would select with early stopping the same as before?

Fig. 6 is the required plot. Here again the best value we find is again $\lambda = 0.01$ but the difference between that and some other smaller values of λ is much less than if we examine test loss at the end of training. This is because when we take a smaller value of λ there is much less regularization and during training the model might find a good set of parameters that optimize both train error and generalize to test loss too but later begin to overfit the train loss and hence no longer are at the optimal test loss value.

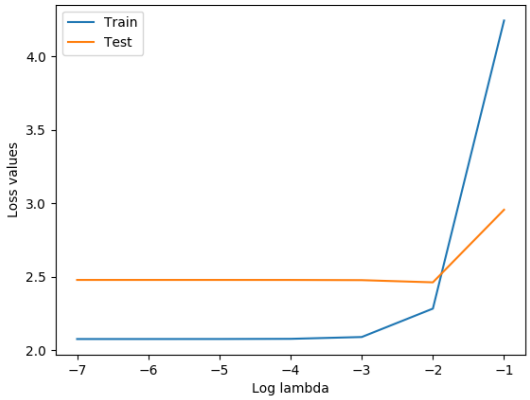


Figure 6: Train and Test loss as a function of $\log \lambda$, but with early stopping where we select the iteration of minimum test loss and not final test loss

Q20. What θ would you select in practice and why?

In practise you would select the parameters θ using validation data. So during the training process maintain a record of the best achieved validation loss up to that point, and at multiple times during each epoch you would examine the validation loss to see if it does better than the best. And every time you find a new best you update the best achieved loss and store that set of parameter values. While λ often shows a trend of overfitting or underfitting, θ can be more noisy so using this process you get the benefit of storing the best loss model (as seen in early stopping case) while also continuously checking if the model manages to do better.

Q21. Show that the objective function $J_\lambda(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta$ can be written in the form $J_\lambda(\theta) = \frac{1}{m}\sum_{i=1}^{m} f_i(\theta)$ by giving an expression for $f_i(\theta)$ that makes the two expressions equivalent.

For a single example (x_i, y_i), choose $f_i(\theta) = (\theta^T x_i - y_i)^2 + \lambda\theta^T\theta$ which represents the loss on a single example.

Q22. Show that the stochastic gradient $\nabla_\theta f_i(\theta)$, for i chosen uniformly at random from $1, ..., m$, is an unbiased estimator of $\nabla_\theta J_\lambda(\theta)$. In other words, show that $\mathbb{E}[\nabla f_i(\theta)] = \nabla J_\lambda(\theta)$ for any θ. It will be easier to prove this for a general $J(\theta) = \sum_{i=1}^{m} f_i(\theta)$, rather than the specific case of ridge regression. You can start by writing down an expression for $\mathbb{E}[\nabla f_i(\theta)]$

If there are m examples:
$\mathbb{E}[\nabla f_i(\theta)] = \frac{1}{m}\sum_{i=1}^{m}\nabla f_i(\theta)$

Consider the general case where, $J(\theta) = \frac{1}{m}\sum_{i=1}^{m} f_i(\theta)$. So $\nabla J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\nabla f_i(\theta)$

$\mathbb{E}[\nabla f_i(\theta)] = \frac{1}{m}\sum_{i=1}^{m}\nabla f_i(\theta)$
(linearity of expectation, each example has equal probability of being selected)
$= \nabla J(\theta)$
So $\nabla_\theta f_i(\theta)$, for i chosen uniformly at random from $1, ..., m$, is an unbiased estimator of $\nabla_\theta J_\lambda(\theta)$

Q23. Write down the update rule for θ in SGD for the ridge regression objective function.

The update rule can be written as: $\theta_{\text{new}} = \theta - \eta[2x_i(\theta^T x_i - y_i) + 2\lambda\theta]$

Q24. Implement stochastic grad descent.

```
def stochastic_grad_descent(X, y, alpha=0.1, lambda_reg=10**-2,
    num_epoch=1000, eta0=False, C=0.1):
    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.ones(num_features) #Initialize theta

    theta_hist = np.zeros((num_epoch, num_instances,
```

```
    num_features)) #Initialize theta_hist
loss_hist = np.zeros((num_epoch, num_instances))

count_updates = 1
for i in range(num_epoch):
    order = np.random.choice(range(num_instances), num_instances, replace=False)
    for idx in order:
        theta_hist[i][idx] = theta
        loss_hist[i][idx] = compute_square_loss(X, y, theta)
        grad = 2 * (X[idx] @ theta.T - y[idx]) * X[idx] +
            2*lambda_reg*theta
        step_size = None
        if isinstance(alpha, float):
            step_size = alpha
        elif alpha == "1/sqrt(t)":
            step_size = (C*1.0)/np.sqrt(count_updates)
            count_updates+=1
        elif alpha == "1/t":
            step_size = (C*1.0)/count_updates
            count_updates+=1
        else:
            print("Invalid alpha")
            return
        theta = theta - step_size*grad
return loss_hist, theta_hist
```

Q25. SGD Results

Fig. 7, 8 and 9 contain the plots for SGD with the three schemes for varying step size - fixed to 0.05, varying according to $C/\sqrt{t}$ and varying according to C/t, $C = 0.1$. We can see that the fixed step size doesn't seem to converge whereas the C/t seems to do best, outperforming $C/\sqrt{t}$ because it reaches a slightly better test loss value faster than the latter. These plots can vary.

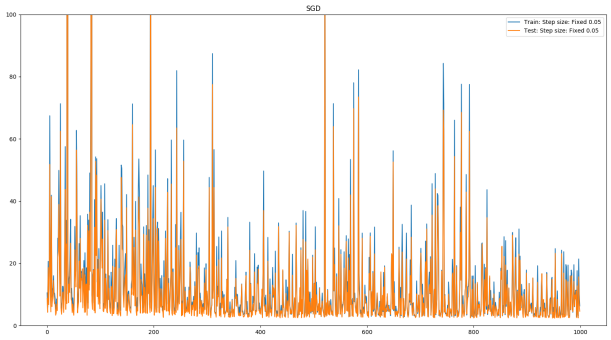


Figure 7: Test loss vs training steps. SGD with fixed step size

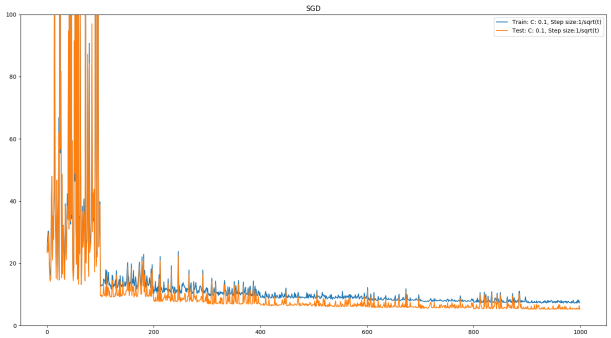


Figure 8: Test loss vs training steps. SGD with step size varying by square root of step count

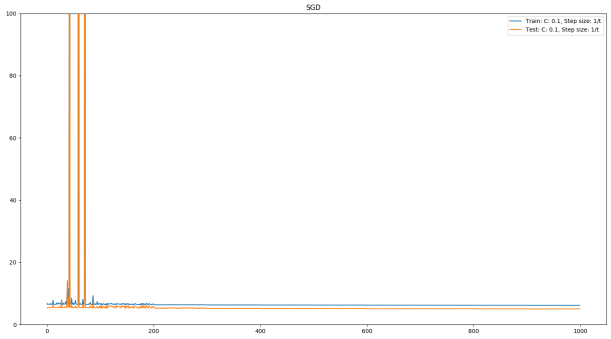


Figure 9: Test loss vs training steps. SGD with step size varying inversely with step count.

Q26. Rewrite the expression for logistic loss as a function of the margin to match the expression in the question.

Margin, $m = yh_{\theta,b}(x)$

Logistic Loss, $l = \log(1 + e^{-m})$

For a set of m examples, logistic loss, $l = \frac{1}{m}\sum_{i=1}^{m}\log(1 + e^{-m_i}) = \frac{1}{m}\sum_{i=1}^{m}\log(1 + e^{-y_i h_{\theta,b}(x_i)})$

In the convention that we apply, $y_i \in \{-1, 1\}$. So the term inside the summation becomes $\log(1 + e^{-h_{\theta,b}(x_i)})$ if $y_i = 1$ and $\log(1 + e^{h_{\theta,b}(x_i)})$ if $y_i = -1$. Additionally these are the only two expressions the summation term can take.

Therefore, the expression becomes:

$l = \frac{1}{m}\sum_{i=1}^{m}\log(1 + e^{-m_i}) = \frac{1}{m}\sum_{i=1}^{m}\mathbb{1}_{y_i=1}\log(1 + e^{-h_{\theta,b}(x_i)}) + \mathbb{1}_{y_i=-1}\log(1 + e^{h_{\theta,b}(x_i)})$

$= \frac{1}{2m}\sum_{i=1}^{m}\mathbb{1}_{y_i=1} \times 2 \times \log(1 + e^{-h_{\theta,b}(x_i)}) + \mathbb{1}_{y_i=-1} \times 2 \times \log(1 + e^{h_{\theta,b}(x_i)})$

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

Q27. What will become the loss function if we regularize the coefficients of θ with an l_1 penalty using a regularization parameter α

Logistic loss, $l = \frac{1}{2m}\Big(\sum_{i=1}^{m}(1 + y_i)\log(1 + e^{-h_{\theta,b}(x_i)}) + (1 - y_i)\log(1 + e^{h_{\theta,b}(x_i)})\Big) + \alpha||\theta||_1$

Q28. To evaluate the quality of our model we will use the classification error, which corresponds to the fraction of incorrectly labeled examples. For a given sample, the classification error is 1 if no example was labeled correctly and 0 if all examples were perfectly labeled. Using the method clf.predict() from the classifier write a function that takes as input an SGDClassifier which we will call clf, a design matrix X and a target vector y and returns the classification error. You should check that your function returns the same value as 1 - clf.score(X, y)

The following function code does the same. Note we add an assertion in case that the error we calculate does not match up with that from clf.score.

```
def classification_error(clf, X, y):
    preds = clf.predict(X)
    #print(preds[:5], y[:5])
    error = 1 - (1.0*sum(preds == y))/len(X)
    skerror = 1 - clf.score(X, y)
    assert error == skerror, print("Errors don't match up - ", error, skerror)
    return error
```

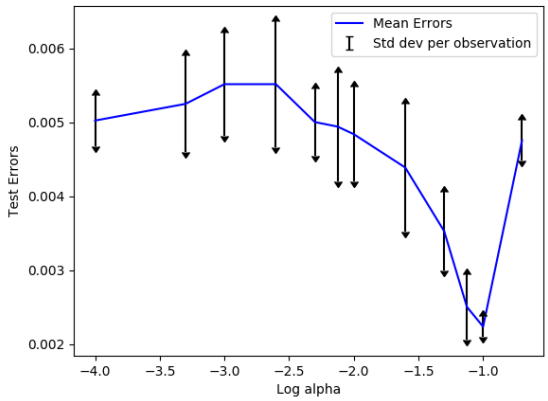


Figure 10: Plot of test error vs $\log \alpha$ repeated 10 times. The points on the line indicate mean error and the arrows on the errorbar indicate standard deviation

Q29. Report the test classification error achieved by the logistic regression as a function of the regularization parameters α (taking 10 values between 10^{-4} and 10^{-1}). You should make a plot with α as the x-axis in log scale. For each value of α, you should repeat the experiment 10 times so has to finally report the mean value and the standard deviation. You should use plt.errorbar to plot the standard deviation as error bars.

Fig. 10 is the required plot. We can see that the best alpha we obtain is at $\alpha = 0.1$. The code to obtain this plot follows the question block. The error values are quite small ($\sim 10^{-3}$) but we see that as we increase α we finally achieve an optimal test error before a sharp increase beyond that. This is good evidence of initial overfitting being overcome with regularization before eventually giving way to too much regularization leading to suboptimal learning.

Q30. Which source(s) of randomness are we averaging over by repeating the experiment?

The sources of randomness I see are that the SGDClassifier has set parameter shuffle=True so the order of training points is shuffled on each epoch which can change the training results by nature of SGD (since updates depend on order of examples). So when we average over 10 runs, we are trying to account for this variability. One point to note is that I always set the seed at the start of the script. So each different run of the script as a whole would yield the same output. Also the subsample function given in the sample code doesn't introduce any randomness.

Q31. What is the optimal value of the parameter α among the values you tested?

The optimal α is 0.1 with lowest mean test error. Some other points nearby have a minimum on par with it but they come with a higher standard deviation so this seems like a good choice in the context.

Q32. Finally, for one run of the fit for each value of α plot the value of the fitted θ. You can access it via clf.coef , and should reshape the 764 dimensional vector to a 28×28 arrray to visualize it with plt.imshow. Defining scale = np.abs(clf.coef).max(), you can use the following keyword arguments (cmap=plt.cm.RdBu, vmax=scale, vmin=-scale) which will set the colors nicely in the plot. You should also use a plt.colorbar() to visualize the values associated with the colors.

Fig. 11 is the required image. We plot it for 12 different α values and from (a) to (l) these are shown in increasing order i.e. more regularization.

Q33. What can you note about the pattern in θ? What can you note about the effect of the regularization?

Firstly we see a really nice general pattern of predicting zero if there's a circle on the outside and one if there's a concentration at the center of the image, which is as you would expect. This is θ so if the input image has a higher concentration of pixels set to 1 either in the circle or the line down the middle, we make a correct prediction because θ_{ij} values in the matrix are set to positive values (blue) in the center and negative values (brown) around it so a zero input would give a more negative output score and one as input would be more positive as we desire. As we increase α we see 2 things - first that the magnitude of pixels seems to go down a notch (especially in negative direction) and also more sparsity as can be expected with more lasso regularization (i.e. higher α, more regularization, more sparsity).

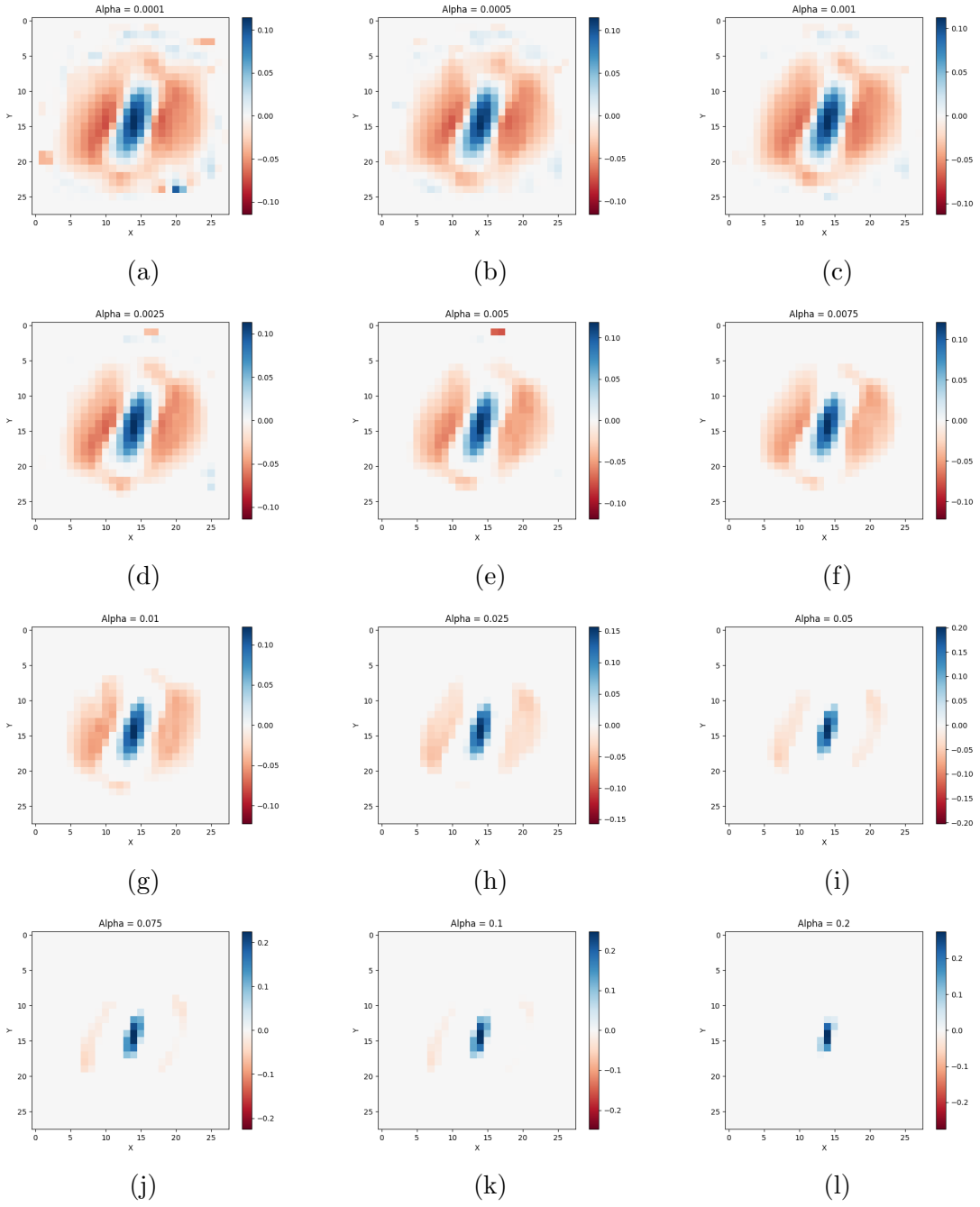


Figure 11: Plot of θ values reshaped to a 28×28 matrix for various values of α.