Homework 2: Gradient Descent & Regularization

Due: Wednesday, February 16, 2022 at 11:59PM

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better. The last application is optional.

This second homework features 3 problem sets and explores gradient descent algorithms, loss functions (both topics of week 2), regularization (topic of week 3) and statistical learning theory (week 1 + week 2). Following the instructions in the homework you should be able to solve a lot of questions before the lecture of week 3. Additionally this homework has an optional problem set. Optional questions will be graded but the points do not count towards this assignment. They instead contribute towards the extra credit you can earn over the entire course (maximum 2%) which can be used to improve the final grade by half a letter (e.g., A- to A).

1 Statistical Learning Theory

In the last HW, we used training error to determine whether our models have converged. It is crucial to understand what is the source of this training error. We specifically want to understand how it is connected to the noise in the data. In this question, we will compute the expected training error when we use least squares loss to fit a linear function.

Consider a full rank $N \times d$ data matrix X ($N > d$) where the training labels are generated as $y_i = b\,x_i + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ is noise. From HW 1, we know the formula for the ERM, $\hat{b} = (X^TX)^{-1}X^Ty$.

1. Show that:

$$\text{Training Error} = \frac{1}{N}\left|\left|\Big(X(X^TX)^{-1}X^T - I\Big)\epsilon\right|\right|_2^2$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2 I_n)$ and training error is defined as $\frac{1}{N}||X\hat{b} - y||_2^2$.

Let $A = X(X^TX)^{-1}X^T$, and $y = Xb + \epsilon$ (note that this is b and not $\hat{b}$. Also note the following property:

$$\begin{aligned} A &= A^2 \\ X(X^TX)^{-1}X^T &= X(X^TX)^{-1}X^TX(X^TX)^{-1}X^T \\ X(X^TX)^{-1}X^T &= X(X^TX)^{-1}X^T \end{aligned} \tag{1}$$

Which shows that A is symmetric with eigenvalues strictly equal to 1 or 0.

To show the equivalency we begin by doing the following manipulations:

$$\begin{aligned}
\frac{1}{N}\bigg|\bigg|\Big(X(X^TX)^{-1}X^T - I\Big)\epsilon\bigg|\bigg|_2^2 &= \frac{1}{N}||X\hat{b} - y||_2^2 \\
&= \frac{1}{N}||Ay - y||_2^2 \\
&= \frac{1}{N}||(A - I)y||_2^2 \\
&= \frac{1}{N}||(A - I)(Xb + \epsilon)||_2^2 \\
&= \frac{1}{N}||(A - I)(A + \epsilon)||_2^2 \\
&= \frac{1}{N}||A^2 - A + A\epsilon - I\epsilon)||_2^2 \\
&= \frac{1}{N}||A\epsilon - I\epsilon||_2^2 \\
&= \frac{1}{N}||(A - I)\epsilon||_2^2 \\
&= \frac{1}{N}\bigg|\bigg|\Big(X(X^TX)^{-1}X^T - I\Big)\epsilon\bigg|\bigg|_2^2
\end{aligned} \tag{2}$$

We firstly substituted our definition for A, then expanded the definition of the Euclidean norm via dot product, foiled the inside terms and manipulated the expression accordingly, resulting in the desired equivalency.

2. Show that the expectation of the training error can be expressed solely in terms of **only** N, d, σ as:

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

Hints:

- Consider $A = X(X^TX)^{-1}X^T$. What is A^TA? Is A symmetric? What is A^2?
- For a symmetric matrix A satisfying $A^2 = A$, what are its eigenvalues?
- If X is full rank, then what is the rank of A? What is the eigenmatrix of A?

We have shown that:

$$\begin{aligned}
\text{Training Error} &= \frac{1}{N}\bigg|\bigg|\Big(A - I\Big)\epsilon\bigg|\bigg|_2^2 \\
&= \frac{1}{N}\langle(A - I), (A - I))\rangle \\
&= \frac{1}{N}(A^2 - 2A + I) * \epsilon^2 \\
&= \frac{1}{N}\epsilon^T * (I_N - A) * \epsilon
\end{aligned} \tag{3}$$

Therefore we can take the expectation and manipulate as such:

$$\begin{aligned} E(\text{Training Error}) &= \frac{1}{N}E(\epsilon^2 * I_N) - E(\epsilon^T * A * \epsilon) \\ &= \frac{1}{N}[E(\Sigma_{i=1}^{n}\epsilon^2) - E(\epsilon^T * A * \epsilon)] \\ &= \frac{1}{N}[(\Sigma_{i=1}^{n}E(\epsilon^2)) - E(\epsilon^T * A * \epsilon)] \\ &= \frac{1}{N}[n\sigma^2 - E(\epsilon^T * A * \epsilon) \\ &= \frac{1}{N}[n\sigma^2 - (\Sigma_{i=1}^{d}E(\epsilon^2) - (\Sigma_{i=d+1}^{n}E(0))] \\ &= \frac{(N-d)\sigma^2}{N} \end{aligned} \tag{4}$$

For this exercise, we begun by applying the second hint that $A^2 = A$, then we utilized the expectation to find the expected training error. Then, using linearity of expectations, we can take the sum of expectations. Finally, using the fact that A is square symmetric, and due to the Spectral Theorem can be expressed as $A = PDP^T$, we utilize the fact that the eigenvalues, can be only 1 or 0, and since the rank of A is equal to d, then the eigenmatrix, D will have 1s in the diagonal from i equals 1 to d, and eigenvalues of 0 from d+1 to n diagonals. Using the properties of the quadratic form, we know that the expression $\epsilon^T A\epsilon$ is the same as $Tr(A) \times \epsilon^2$.

3. 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.

   As we increase the number of features we include in our model, we are essentially increasing the amount of terms in our prediction function. With more terms, we can fit the noise present in our ground truth function better. This overfitting process reduces training error (and without a doubt would increase test error).

   In other words, as d approaches N the numerator in the equation we proved in part 2 ($\frac{(N-d)\sigma}{N}$) approaches 0. When that occurs the total training error becomes 0.

2 Gradient descent for ridge(less) linear regression

Dataset

We have provided you with a file called `ridge_regression_dataset.csv`. Columns `x0` through `x47` correspond to the input and column `y` corresponds to the output. We are trying to fit the data using a linear model and gradient based methods. Please also check the supporting code in `skeleton_code.py`. Throughout this problem, we refer to particular blocks of code to help you step by step.

Feature normalization When feature values differ greatly, we can get much slower rates of convergence of gradient-based algorithms. Furthermore, when we start using regularization, features with larger values are treated as "more important", which is not usually desired.

One common approach to feature normalization is perform an affine transformation (i.e. shift and rescale) on each feature so that all feature values in the training set are in $[0, 1]$. Each feature gets its own transformation. We then apply the same transformations to each feature on the validation set or test set. Importantly, the transformation is "learned" on the training set, and then applied to the test set. It is possible that some transformed test set values will lie outside the $[0, 1]$ interval.

4. 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.

At the end of the skeleton code, the function `load_data` loads, split into a training and test set, and normalize the data using your `feature_normalization`.

Linear regression

In linear regression, we consider the hypothesis space of linear functions $h_\theta : \mathbb{R}^d \to \mathbb{R}$, where

$$h_\theta(x) = \theta^T x,$$

for $\theta, \boldsymbol{x} \in \mathbb{R}^d$, and we choose θ that minimizes the following "average square loss" objective function:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(\boldsymbol{x}_i) - y_i\right)^2,$$

where $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m) \in \mathbb{R}^d \times \mathbb{R}$ is our training data.

While this formulation of linear regression is very convenient, it's more standard to use a hypothesis space of affine functions:

$$h_{\theta,b}(x) = \theta^T \boldsymbol{x} + b,$$

which allows a nonzero intercept term b – sometimes called a "bias" term. The standard way to achieve this, while still maintaining the convenience of the first representation, is to add an extra dimension to $\boldsymbol{x}$ that is always a fixed value, such as 1, and use $\theta, x \in \mathbb{R}^{d+1}$. Convince yourself that this is equivalent. We will assume this representation.

5. Let $X \in \mathbb{R}^{m\times(d+1)}$ be the *design matrix*, where the i'th row of X is $\boldsymbol{x}_i$. Let $y = (y_1, \ldots, y_m)^T \in \mathbb{R}^{m\times 1}$ be the *response*. Write the objective function $J(\theta)$ as a ma-

trix/vector expression, without using an explicit summation sign. [1]

This is a relatively easy task, as our design matrix X can be multiplied by a vector of weights, θ to calculate our predicted $\hat{y}$ values, where $\hat{y} \in \mathbb{R}^{m\times 1}$. Therefore, our loss function can be re-expressed as the following simple linear algebra expression:

$$J(\theta) = \frac{1}{m}||X\theta - y||_2^2$$

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

   The gradient of our loss function is defined as the vector of partial derivatives with respect to θ. It can be expressed as follows:

$$\begin{aligned}\nabla J(\theta) &= \frac{\partial J}{\partial \theta}\frac{1}{m}||X\theta - y||_2^2 \\ &= \frac{\partial J}{\partial \theta}(\frac{1}{m}(\theta^T X^T X\theta + y^T y - 2\theta^T X^T y)) \\ &= \frac{2}{m}(X^T X\theta - X^T y)\end{aligned} \tag{5}$$

   Therefore:

$$\nabla J(\theta) = \frac{\partial J}{\partial \theta} = \frac{2}{m}(X^T X\theta - X^T y)$$

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

   Our algorithm to update θ is rather simple. To calculate the next theta, iteration $i+1$, we take the θ from iteration i and subtract the gradient that we calculated above, scaled for some step size η:

$$\theta^{i+1} = \theta^i - \eta\nabla J(\theta^i) = \theta^i - \eta(\frac{2}{m}(X^T X\theta^i - X^T y)))$$

8. 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.

9. 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.

Gradient checker

We can numerically check the gradient calculation. If $J : \mathbb{R}^d \to \mathbb{R}$ is differentiable, then for any vector $h \in \mathbb{R}^d$, the directional derivative of J at θ in the direction h is given by

$$\lim_{\epsilon\to 0}\frac{J(\theta + \epsilon h) - J(\theta - \epsilon h)}{2\epsilon}.$$

[1] Being able to write expressions as matrix/vector expressions without summations is crucial to making implementations that are useful in practice, since you can use numpy (or more generally, an efficient numerical linear algebra library) to implement these matrix/vector operations orders of magnitude faster than naively implementing with loops in Python.

It is also given by the more standard definition of directional derivative,

$$\lim_{\epsilon \to 0} \frac{1}{\epsilon} \left[J(\theta + \epsilon h) - J(\theta) \right] \ .$$

The former form gives a better approximation to the derivative when we are using small (but not infinitesimally small) ϵ. We can approximate this directional derivative by choosing a small value of $\epsilon > 0$ and evaluating the quotient above. We can get an approximation to the gradient by approximating the directional derivatives in each coordinate direction and putting them together into a vector. In other words, take $h = (1, 0, 0, \ldots, 0)$ to get the first component of the gradient. Then take $h = (0, 1, 0, \ldots, 0)$ to get the second component, and so on.

10. 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.

You should be able to check that the gradients you computed above remain correct throughout the learning below.

Batch gradient descent

We will now finish the job of running regression on the training set.

11. Complete `batch_gradient_descent`. Note the phrase *batch* gradient descent distinguishes between *stochastic* gradient descent or more generally *minibatch* gradient descent.

12. 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.

13. 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.

Ridge Regression

We will add ℓ_2 regularization to linear regression. When we have a large number of features compared to instances, regularization can help control overfitting. *Ridge regression* is linear regression with ℓ_2 regularization. The regularization term is sometimes called a penalty term. The objective function for ridge regression is

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

where λ is the regularization parameter, which controls the degree of regularization. Note that the bias term (which we included as an extra dimension in θ) is being regularized as well as the other parameters. Sometimes it is preferable to treat this term separately.

14. 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)

    Gradient of computing $J_\lambda(\theta)$: To begin, lets derive the gradient of ridge regression using linear algebra, similarly with how we did to problem 6.

$$\begin{aligned} J_\lambda(\theta) &= \frac{1}{m}||X\theta - y||_2^2 + \lambda\theta^T\theta \\ \nabla J_\lambda(\theta) &= \frac{\partial J}{\partial\theta}(\frac{1}{m}(\theta^T X^T X\theta + y^T y - 2\theta^T X^T y) + \lambda\theta^T\theta) \\ \nabla J_\lambda(\theta) &= \frac{2}{m}(X^T X\theta - X^T y) + 2\lambda\theta \end{aligned} \tag{6}$$

    Updating the gradient descent algorithm with our new gradient:

$$\theta^{i+1} = \theta^i - \eta\nabla J(\theta^i) = \theta^i - \eta(\frac{2}{m}(X^T X\theta^i - X^T y) + 2\lambda\theta^i)))$$

15. Implement `compute_regularized_square_loss_gradient`.

16. Implement `regularized_grad_descent`.

Our goal is to find λ that gives the minimum average square loss on the test set. So you should start your search very broadly, looking over several orders of magnitude. For example, $\lambda \in \left\{10^{-7}, 10^{-5}, 10^{-3}, 10^{-1}, 1, 10, 100\right\}$. Then you can zoom in on the best range. Follow the steps below to proceed.

17. 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?

18. 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 ?

19. 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?

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

    In practice, it is best to select the θ that minimizes testing error the most out of all θ considered. The reason for this is that we want our model to generalize the best to real world data, and our testing split is the best indicator during an experiment of how the model might perform once deployed. In our example, it seems that the value of 10^{-2} minimized the testing error the most, so that is the one we would choose to use.

Stochastic Gradient Descent (SGD) (optional)

When the training data set is very large, evaluating the gradient of the objective function can take a long time, since it requires looking at each training example to take a single gradient step. In SGD, rather than taking $-\nabla_\theta J(\theta)$ as our step direction to minimize

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} f_i(\theta),$$

we take $-\nabla_\theta f_i(\theta)$ for some i chosen uniformly at random from $\{1,\ldots,m\}$. The approximation is poor, but we will show it is unbiased.

In machine learning applications, each $f_i(\theta)$ would be the loss on the ith example. In practical implementations for ML, the data points are randomly shuffled, and then we sweep through the whole training set one by one, and perform an update for each training example individually. One pass through the data is called an *epoch*. Note that each epoch of SGD touches as much data as a single step of batch gradient descent. You can use the same ordering for each epoch, though optionally you could investigate whether reshuffling after each epoch affects the convergence speed.

21. Show that the objective function

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

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.

This is can be simply shown by substituting our definition of our loss function, and calculating its risk. Our Ridge Regression Least squares loss function for a single point, (x_i, y_i) is defined as:

$$f_i(\theta) = (h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta$$

Taking a batch of size m (m denoting the number of data points used in SGD) we have:

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

And thus, showing the equivalency.

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

Firstly, we start with the definition of the Ridge Regression gradient:

$$\nabla f_i(\theta) = \frac{2}{m}(X_i^T X_i\theta - X_i^T y_i) + 2\lambda\theta$$

We can show that the SGD is an unbiased estimator of the full batch gradient by first illustrating what happens when we plug in only 1 data point into our SGD gradient calculator. Note that the probability of selecting any one individual data point is $P(x_i = i) = \frac{1}{m}$.

$$\nabla f_i(\theta) = 2(X_i^T X_i\theta - X_i^T y_i) + 2\lambda\theta$$

We can then apply the expected value to this function. Adding in the probability of selecting any one of our m data points for each m datapoint we can show the equivalency:

$$\begin{aligned}\mathbb{E}[\nabla f_i(\theta)] &= E[2(X_i^T X_i\theta - X_i^T y_i) + 2\lambda\theta] \\ &= 2 \times \frac{1}{m} \times \sum_{i=1}^{m}(X_i^T X_i\theta - X_i^T y_i) * P(X_i = i) + 2\lambda\theta \\ &= \frac{2}{m}(X^T X\theta - X^T y) + 2\lambda\theta\end{aligned} \tag{7}$$

This proves the equivalency as:

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

Which we accomplished by noting that the probability of selecting any one data point m is:

$$2(X_i^T X_i\theta - X_i^T y_i) * P(X_i = i)$$

After all that we have:

$$\mathbb{E}[\nabla f_i(\theta)] = \nabla J_\lambda(\theta)$$

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

    We simply rewrite the way we would normally update the theta during ridge regression, but instead of using the whole design matrix X and labels, y, we use the subset selected for SGD: the j^{th} row of X and the j^{th} observation in our target vector, y. For each iteration i of our stochastic gradient descent, we update θ as follows:

$$\theta^{i+1} = \theta^i - \eta\nabla f_j(\theta^i) = \theta^i - \eta(\frac{2}{m}(X_j^T X_j\theta^i - X_j^T y_j) + 2\lambda\theta^i)))$$

24. Implement `stochastic_grad_descent`.

25. Use SGD to find θ_λ^* that minimizes the ridge regression objective for the λ you selected in the previous problem. (If you could not solve the previous problem, choose $\lambda = 10^{-2}$). Try a few fixed step sizes (at least try $\eta_t \in \{0.05, .005\}$). Note that SGD may not converge with fixed step size. Simply note your results. Next try step sizes that decrease with the step number according to the following schedules: $\eta_t = \frac{C}{t}$ and $\eta_t = \frac{C}{\sqrt{t}}$, $C \leq 1$. Please include $C = 0.1$ in your submissions. You are encouraged to try different values of C (see notes below for details). For each step size rule, plot the value of the objective function (or the log of the objective function if that is more clear) as a function of epoch (or step number, if you prefer). How do the results compare?

A few remarks about the question above:

- In this case we are investigating the convergence rate of the optimization algorithm with different step size schedules, thus we're interested in the value of the objective function, which includes the regularization term.

- Sometimes the initial step size (C for C/t and $C/\sqrt{t}$) is too aggressive and will get you into a part of parameter space from which you can't recover. Try reducing C to counter this problem.

- SGD convergence is much slower than GD once we get close to the minimizer (remember, the SGD step directions are very noisy versions of the GD step direction). If you look at the objective function values on a logarithmic scale, it may look like SGD will never find objective values that are as low as GD gets. In statistical learning theory terminology, GD has much smaller *optimization* error than SGD. However, this difference in optimization error is usually dominated by other sources of error (estimation error and approximation error). Moreover, for very large datasets, SGD (or minibatch GD) is much faster (by wall-clock time) than GD to reach a point close enough to the minimizer.

Acknowledgement: This problem set is based on assignments developed by David Rosenberg of NYU and Bloomberg.

3 Image classification with regularized logistic regression

Dataset

In this second problem set we will examine a classification problem. To do so we will use the MNIST dataset[2] which is one of the traditional image benchmark for machine learning algorithms. We will only load the data from the 0 and 1 class, and try to predict the class from the image. You will find the support code for this problem in `mnist_classification_source_code.py`. Before starting, take a little time to inspect the data. Load `X_train, y_train, X_test, y_test` with `pre_process_mnist_01()`. Using the function `plt.imshow` from `matplotlib` visualize some data points from `X_train` by reshaping the 764 dimensional vectors into 28×28 arrays. Note how the class labels '0' and '1' have been encoded in `y_train`. No need to report these steps in your submission.

Logistic regression

We will use here again a linear model, meaning that we will fit an affine function,

$$h_{\theta,b}(\boldsymbol{x}) = \theta^T \boldsymbol{x} + b,$$

with $\boldsymbol{x} \in \mathbb{R}^{764}$, $\boldsymbol{\theta} \in \mathbb{R}^{764}$ and $b \in \mathbb{R}$. This time we will use the logistic loss instead of the squared loss. Instead of coding everything from scratch, we will also use the package `scikit learn` and study the effects of ℓ_1 regularization. You may want to check that you have a version of the package up to date (0.24.1).

26. Recall the definition of the logistic loss between target y and a prediction $h_{\theta,b}(\boldsymbol{x})$ as a function of the margin $m = yh_{\theta,b}(\boldsymbol{x})$. Show that given that we chose the convention $y_i \in \{-1, 1\}$, our objective function over the training data $\{\boldsymbol{x}_i, y_i\}_{i=1}^m$ can be re-written as

$$L(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(1 + y_i)\log(1 + e^{-h_{\theta,b}(\boldsymbol{x}_i)}) + (1 - y)\log(1 + e^{h_{\theta,b}(\boldsymbol{x}_i)}).$$

We know from Lecture 2 that the definition of the logistic log function is as follows:

$$L(\theta) = \frac{1}{n}\sum_{i}^{n} ln(1 + e^{-m})$$

where n is the number of data points and $m = y_i h_{\theta,b}(x_i)$. Using our definition above, and substituting in our margin for m we have the following statement:

$$L(\theta) = \frac{1}{m}\sum_{i}^{m} ln(1 + e^{-y_i h_{\theta,b}(x_i)}) \text{ where } y \in -1, 1$$

Please note: now the variable m denotes the number of datapoints, and we no longer use the indicator variable n.

We can prove the equivalency by doing a few manipulations. Firstly, lets multiply our statement by $\frac{2}{2}$ to get the following:

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

[2]http://yann.lecun.com/exdb/mnist/

Now lets explore what happens to our equation when y_i takes it's two possible values, (when $y_i \in [-1, 1]$).

Firstly, lets try substituting y_i = -1 to illustrate the equivalency we need to prove

$$\begin{aligned}
L(\theta) &= \frac{1}{2m}\sum_{i}^{m}(1+-1)*ln(1+e^{h_{\theta,b}(x_i)}) + (1+1)*ln(1+e^{h_{\theta,b}(x_i)}) \\
&= \frac{1}{2m}\sum_{i}^{m}(0)*ln(1+e^{h_{\theta,b}(x_i)}) + 2*ln(1+e^{h_{\theta,b}(x_i)}) \\
&= \frac{1}{2m}\sum_{i}^{m}2*ln(1+e^{h_{\theta,b}(x_i)}) \\
&= \frac{1}{m}\sum_{i}^{m}ln(1+e^{h_{\theta,b}(x_i)})
\end{aligned} \tag{8}$$

After doing the above manipulations, we arrive at the original logistic loss function, thus shoving the equivalency between the two when $y = -1$

Now for our other case, when $y_i = 1$:

$$\begin{aligned}
L(\theta) &= \frac{1}{2m}\sum_{i}^{m}(1+1)*ln(1+e^{-h_{\theta,b}(x_i)}) + (1-1)*ln(1+e^{h_{\theta,b}(x_i)}) \\
&= \frac{1}{2m}\sum_{i}^{m}2*ln(1+e^{-h_{\theta,b}(x_i)}) + 0*ln(1+e^{h_{\theta,b}(x_i)}) \\
&= \frac{1}{2m}\sum_{i}^{m}2*ln(1+e^{-h_{\theta,b}(x_i)}) \\
&= \frac{1}{m}\sum_{i}^{m}ln(1+e^{-h_{\theta,b}(x_i)})
\end{aligned} \tag{9}$$

And again, we arrive at our original logistic loss function, thus showing equivalency between the two functions when $y = 1$. Since the only two cases we can have are $y = 1, y = -1$, we have proven the equivalency between the two functions.

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

    If we regularize the loss function similarly to how we have done in previos problems like Ridge Regression, our Logistic Loss Function would change accordignly:

$$L(\theta) = \frac{1}{n}\sum_{i}^{n}ln(1+e^{-m}) + \alpha||\theta||_1$$

    Where $||\theta||_1$ is the $\ell 1$ norm of θ with regularization parameter α.

We are going to use the module `SGDClassifier` from scikit learn. In the code provided we have set a little example of its usage. By checking the online documentation[3], make sure you

[3] `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html`

understand the meaning of all the keyword arguments that were specified. We will keep the learning rate schedule and the maximum number of iterations fixed to the given values for all the problem. Note that scikit learn is actually implementing a fancy version of SGD to deal with the ℓ_1 penalty which is not differentiable everywhere, but we will not enter these details here.

28. 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)`.

To speed up computations we will subsample the data. Using the function `sub_sample`, restrict `X_train` and `y_train` to `N_train = 100`.

29. 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.

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

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

32. 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.

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

In [1]:
```
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
# ########################################
# ### Generic gradient checker
# def generic_gradient_checker(X, y, theta, objective_func, gradient_f
#                              epsilon=0.01, tolerance=1e-4):
#     """
#     The functions takes objective_func and gradient_func as paramete
#     And check whether gradient_func(X, y, theta) returned the true
#     gradient for objective_func(X, y, theta).
#     Eg: In LSR, the objective_func = compute_square_loss, and gradie
#     """
#     #TODO

def load_data():
    #Loading the dataset
    print('loading the dataset')

    df = pd.read_csv('ridge_regression_dataset.csv', delimiter=',')
    X = df.values[:,:-1]
    y = df.values[:,-1]

    print('Split into Train and Test')
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_siz
    print("Scaling all to [0, 1]")
    X_train, X_test = feature_normalization(X_train, X_test)
    X_train = np.hstack((X_train, np.ones((X_train.shape[0], 1))))  #
    X_test = np.hstack((X_test, np.ones((X_test.shape[0], 1))))
    return X_train, y_train, X_test, y_test
```

Problem 4

In [3]:
```
########################################
### 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,
        test - test set, a 2D numpy array of size(num_instances, num_f

    Returns:
        train_normalized - training set after normalization
        test_normalized - test set after normalization
    """
    #Initialize return variables
    train_normalized, test_normalized = [] , []
    #Iterate over 2D array
    for d in range(train.shape[1]):
        #Grab subsets
        train_temp = train[:,d]
        test_temp = test[:,d]
        train_min = train_temp.min()
        train_max = train_temp.max()
        #If the column is not filled with constants, we want to incluc
        if len(np.unique(train_temp)) > 1:
            #Grab the columns, transform, and append to list (implicit
            train_normalized.append((train_temp - train_min) / (train_
            test_normalized.append((test_temp - train_min) / (train_ma

    #Reformat data type to np.array and Transpose so rows become colum
    train_normalized, test_normalized = np.array(train_normalized).T,
    #Return values
    return train_normalized, test_normalized
```

In [4]:
```
x_train, y_train, x_test, y_test = load_data()
```

```
loading the dataset
Split into Train and Test
Scaling all to [0, 1]
```

Problem 8

In [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 pr

    Args:
        X - the feature vector, 2D numpy array of size(num_instances,
        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
    """
    #Count number of rows
    n = X.shape[0]
    #Calculate y hat
    y_hat = X @ theta
    #Calculate sum of squares
    sum_of_squares = sum((y_hat - y)**2)
    #Return our calculated loss
    loss = sum_of_squares/n
    return loss
```

Problem 9

In [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 comp

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

    Returns:
        grad - gradient vector, 1D numpy array of size(num_features)
    """
    #Calculate number of observations
    m = X.shape[0]
    #Calculate gradient using closed form solution from problem 6
    gradient = ((X.T @ X @ theta) - (X.T @ y)) * 2/m
    #Return gradient
    return gradient
```

In [7]:
```
compute_square_loss_gradient(x_train, y_train, np.random.rand(x_train.
```

Out[7]:
```
array([22.73131311, 22.12803126, 21.50621043, 21.15094816, 20.8970598
5,
       20.465192  , 19.65398635, 19.65398635, 18.67988746, 17.0252665
7,
       15.5707931 , 14.81789844, 12.56434133, 10.8761827 ,  6.7358557
9,
        5.92556579,  4.17655825,  0.73183347, 19.94527049, 19.9452704
9,
       19.94527049, 18.46507502, 18.46507502, 18.46507502, 17.0787958
3,
       17.07879583, 17.07879583, 16.43568024, 16.43568024, 16.4356802
4,
       16.07860351, 16.07860351, 16.07860351, 10.08809468, 10.0880946
8,
       10.08809468, 12.27488263, 12.27488263, 12.27488263, 13.5354083
8,
       13.53540838, 13.53540838, 14.10817529, 14.10817529, 14.1081752
9,
       14.42257155, 14.42257155, 14.42257155, 22.91995948])
```

Problem 10

In [8]:
```
#######################################
### Gradient checker
#Getting the gradient calculation correct is often the trickiest part
#of any gradient-based optimization algorithm. Fortunately, it's very
#easy to check that the gradient calculation is correct using the
#definition of gradient.
#See http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_ac
def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):
    """Implement Gradient Checker
    Check that the function compute_square_loss_gradient returns the
    correct gradient for the given X, y, and theta.

    Let d be the number of features. Here we numerically estimate the
    gradient by approximating the directional derivative in each of
    the d coordinate directions:
(e_1 =(1,0,0,...,0), e_2 =(0,1,0,...,0), ..., e_d =(0,...,0,1))

    The approximation for the directional derivative of J at the point
    theta in the direction e_i is given by:
(J(theta + epsilon * e_i) - J(theta - epsilon * e_i)) /(2*epsilon).

    We then look at the Euclidean distance between the gradient
    computed using this approximation and the gradient computed by
    compute_square_loss_gradient(X, y, theta).  If the Euclidean
    distance exceeds tolerance, we say the gradient is incorrect.

    Args:
        X - the feature vector, 2D numpy array of size(num_instances,
        y - the label vector, 1D numpy array of size(num_instances)
        theta - the parameter vector, 1D numpy array of size(num_featu
        epsilon - the epsilon used in approximation
        tolerance - the tolerance error

    Return:
        A boolean value indicating whether the gradient is correct or
    """
    ### Given Code:
    true_gradient = compute_square_loss_gradient(X, y, theta) #The tru
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features) #Initialize the gradient we a

    ### My Code:
    #Generate helper variable, a vector filled with epsilons
    epsilon_vec = np.array([epsilon]*num_features)

    #Iterate d times, where d is the number of features
    for i in range(num_features):
        #Generate array of 0s
        e_i = np.zeros(num_features)
        #Set the ith entry to 1
        e_i[i] = 1
        #Compute Partial Derivative for ith entry
        partial = (compute_square_loss(X,y,theta+(e_i * epsilon)) - (c
        #Write to our approx_grad array
        approx_grad[i] = partial
    #Calculate the Euclidean Norm of the difference in gradients
    difference = (sum((approx_grad - true_gradient)**2)) ** .5
    #Check if our approximation exceeded the tolerance level
    if difference > tolerance:
        return False #If the difference exceeds tolerance, return Fals
    else:
        return True #Otherwise our gradient works well, return True
```

In [9]:
```
grad_checker(x_train, y_train,np.random.rand(x_train.shape[1]))
```

Out[9]:
```
True
```

Problem 11

In [10]:
```
########################################
### Batch gradient descent
def batch_grad_descent(X, y, alpha=0.1, num_step=1000, grad_check=Fals
    """
    In this question you will implement batch gradient descent to
    minimize the average square loss objective.

    Args:
        X - the feature vector, 2D numpy array of size(num_instances,
        y - the label vector, 1D numpy array of size(num_instances)
        alpha - step size in gradient descent
        num_step - number of steps to run
        grad_check - a boolean value indicating whether checking the g

    Returns:
        theta_hist - the history of parameter vector, 2D numpy array o
                     for instance, theta in step 0 should be theta_his
        loss_hist - the history of average square loss on the data, 1D
    """
    ### Given Code:
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_step + 1, num_features))  #Initialize t
    loss_hist = np.zeros(num_step + 1)  #Initialize loss_hist
    theta = np.zeros(num_features)  #Initialize theta
    ### My Code:
    #theta = np.random.rand(num_features)
    #Iterate over gradient descent steps
    for i in range(0,num_step+1):
        #Write theta
        theta_hist[i,:] = theta
        #Compute Gradient and Calculate and Append Loss
        gradient = compute_square_loss_gradient(X,y,theta)
        loss_hist[i] = compute_square_loss(X,y,theta)
        #Calculate and write new theta
        theta = theta - (alpha * gradient)
    #Return theta_hist, loss_hist
    return theta_hist, loss_hist
```

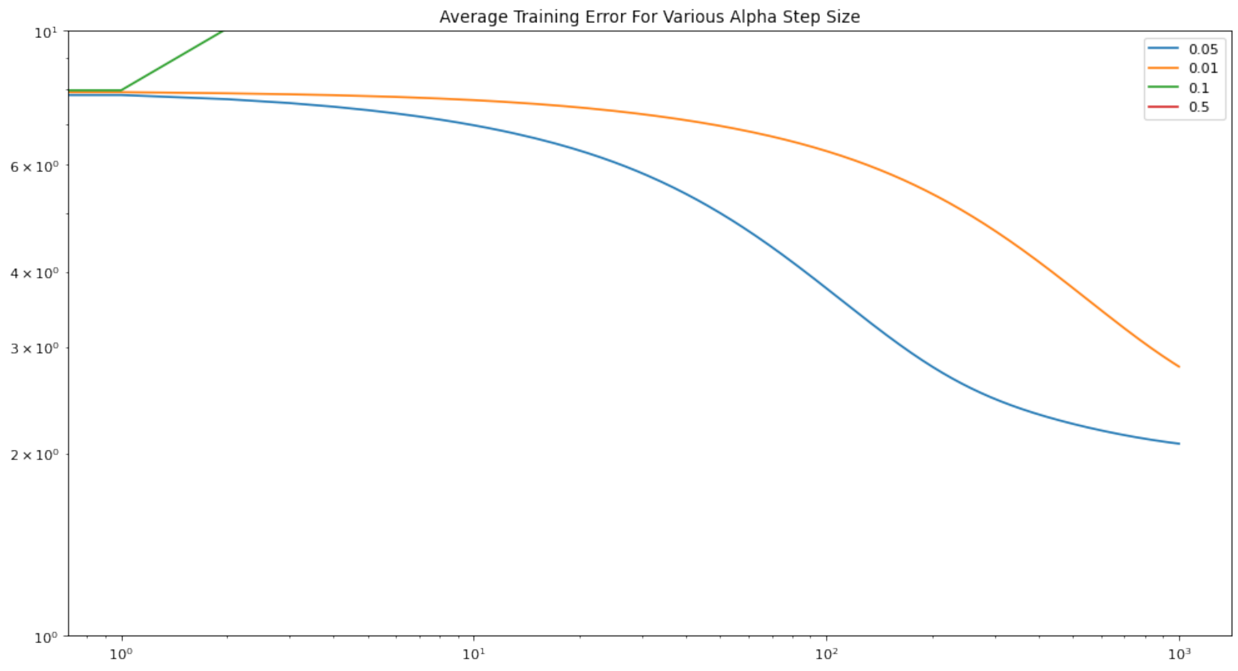
Problem 12

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.

In [11]:
```
alphas = [.05,.01,.1,.5]
alpha_theta_hist_dict, alpha_loss_hist_dict = {}, {}
for a in alphas:
    alpha_theta_hist_dict[a] = []
    alpha_loss_hist_dict[a] = []

for a in alphas:
    alpha_theta_hist_dict[a], alpha_loss_hist_dict[a] = batch_grad_des
```

In [12]:
```
#Generate helper variable for x-axis
x = list(range(1001))
#Begin plotting
plt.figure(figsize=(15, 8), dpi=80)
for alpha in alphas:
    plt.plot(x,alpha_loss_hist_dict[alpha], label=alpha)
plt.yscale('log')
plt.xscale('log')
plt.legend()
plt.title("Average Training Error For Various Alpha Step Size")
```

Out[12]:
```
Text(0.5, 1.0, 'Average Training Error For Various Alpha Step Size')
```



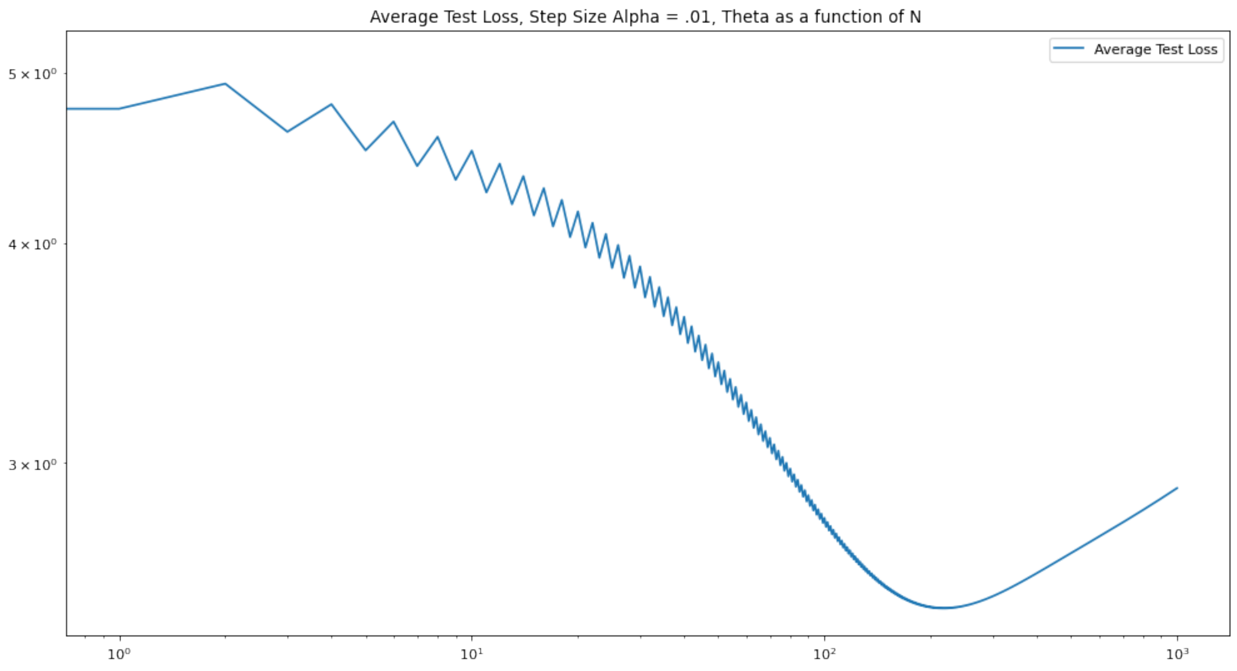
It appears that step sizes .5 and .01 diverge to infinity, while step sizes .05 and .01 converge to ~2 and ~2.7 respectively.

Problem 13

In [13]:
```
###Using step size alpha = .05 calculate average test loss as a functi

#Initialize helper variables
N = 1000
average_test_loss = np.zeros(N)

#Iterate over 1000 steps
for step in range(N):
    #Calculate average test loss
    average_test_loss[step] = compute_square_loss(x_test,y_test, alpha
```

In [14]:
```
#Generate helper variable for x-axis
x = list(range(1000))
#Begin plotting
plt.figure(figsize=(15, 8), dpi=80)
plt.plot(x,average_test_loss, label='Average Test Loss')
plt.yscale('log')
plt.xscale('log')
plt.legend()
plt.title("Average Test Loss, Step Size Alpha = .01, Theta as a functi
```

Out[14]:
```
Text(0.5, 1.0, 'Average Test Loss, Step Size Alpha = .01, Theta as a
function of N ')
```



Question 15

In [15]:
```
########################################
### The gradient of regularized batch gradient descent
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
    """
    Compute the gradient of L2-regularized average square loss functic

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

    Returns:
        grad - gradient vector, 1D numpy array of size(num_features)
    """
    #Calculate number of observations
    m = X.shape[0]
    #Calculate gradient using closed form solution from problem 14
    grad = (((X.T @ X @ theta) - (X.T @ y)) * 2/m) + (2*lambda_reg*the
    #Return gradient
    return grad
```

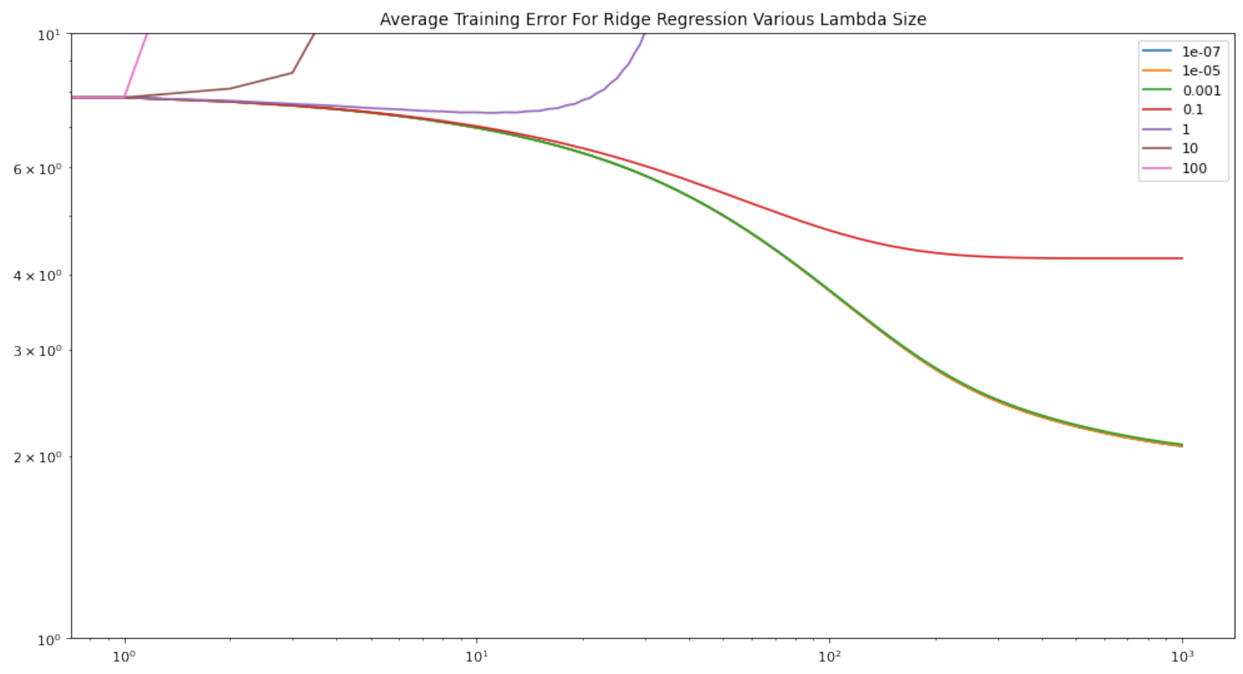
Question 16

In [16]:
```
#######################################
### Regularized batch gradient descent
def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=10**-2, num_
    """
    Args:
        X - the feature vector, 2D numpy array of size(num_instances,
        y - the label vector, 1D numpy array of size(num_instances)
        alpha - step size in gradient descent
        lambda_reg - the regularization coefficient
        num_step - number of steps to run

    Returns:
        theta_hist - the history of parameter vector, 2D numpy array o
                     for instance, theta in step 0 should be theta_his
        loss hist - the history of average square loss function withou
    """
    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 thet
    loss_hist = np.zeros(num_step+1) #Initialize loss_hist
    #TODO
    #Iterate over gradient descent steps
    for i in range(0,num_step+1):
        #Write theta
        theta_hist[i,:] = theta
        #Compute Gradient and Calculate and Append Loss
        loss_hist[i] = compute_square_loss(X,y,theta)
        #Calculate gradient
        gradient = compute_regularized_square_loss_gradient(X,y,theta,
        #Calculate and write new theta
        theta = theta - (alpha * gradient)
    #Return theta_hist, loss_hist
    return theta_hist, loss_hist
```

Question 17

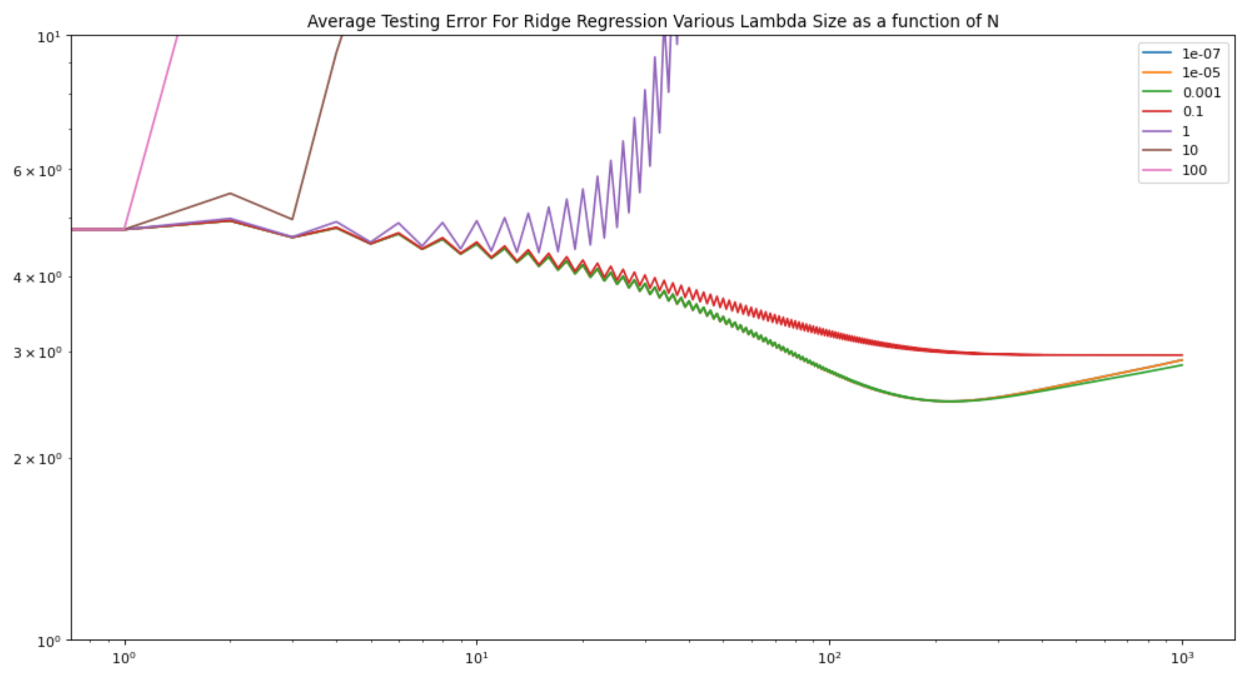
In [17]:
```
#Define helper variables
lambdas = [10**-7, 10**-5, 10**-3, 10**-1,1, 10,100]
step_size = .05
train_loss_dict = {}
hist_theta_dict = {}
test_loss_dict = {}
hist_theta_dict = {}
#Iterate over lambdas
for l in lambdas:
     hist_theta_dict[l], train_loss_dict[l] = regularized_grad_descent
#Generate helper variable for x-axis
x = list(range(1001))
#Begin plotting
plt.figure(figsize=(15, 8), dpi=80)
for l in lambdas:
    plt.plot(x,train_loss_dict[l], label=''.join(str(l)))
plt.yscale('log')
plt.xscale('log')
plt.legend()
plt.title("Average Training Error For Ridge Regression Various Lambda
```

Out[17]:
```
Text(0.5, 1.0, 'Average Training Error For Ridge Regression Various L
ambda Size')
```



In [18]:
```
###Using step size alpha = .05 calculate average test loss as a functi

#Initialize helper variables
N = 1001
average_test_loss = {}
for l in lambdas:
    average_test_loss[l] = []
#Iterate over 1000 steps
for l in lambdas:
    for step in range(N):
        #Calculate average test loss
        average_test_loss[l].append(compute_square_loss(x_test,y_test,
    average_test_loss[l] = np.array(average_test_loss[l]) #Make np arr
#Generate helper variable for x-axis
x = list(range(1001))
#Begin plotting
plt.figure(figsize=(15, 8), dpi=80)
for l in lambdas:
    plt.plot(x,average_test_loss[l], label=''.join(str(l)))
plt.yscale('log')
plt.xscale('log')
plt.legend()
plt.title("Average Testing Error For Ridge Regression Various Lambda S
```

Out[18]:
```
Text(0.5, 1.0, 'Average Testing Error For Ridge Regression Various La
mbda Size as a function of N')
```



While some lambdas diverge, other lambdas such as $L \in [10^{-7}, 10^{-5}, 10^{-3}, 10^{-1}]$ begin to reduce testing error and show the benefeits of regularization, as our model begins generalizing better and better to the testing data. This is not without fault: as for all of the lines that performed better begin to overfit the data as the step size increases above 200. This is clearly illustrated on the graph above as the curved lines change from sloping downwards to upwards.

Question 18

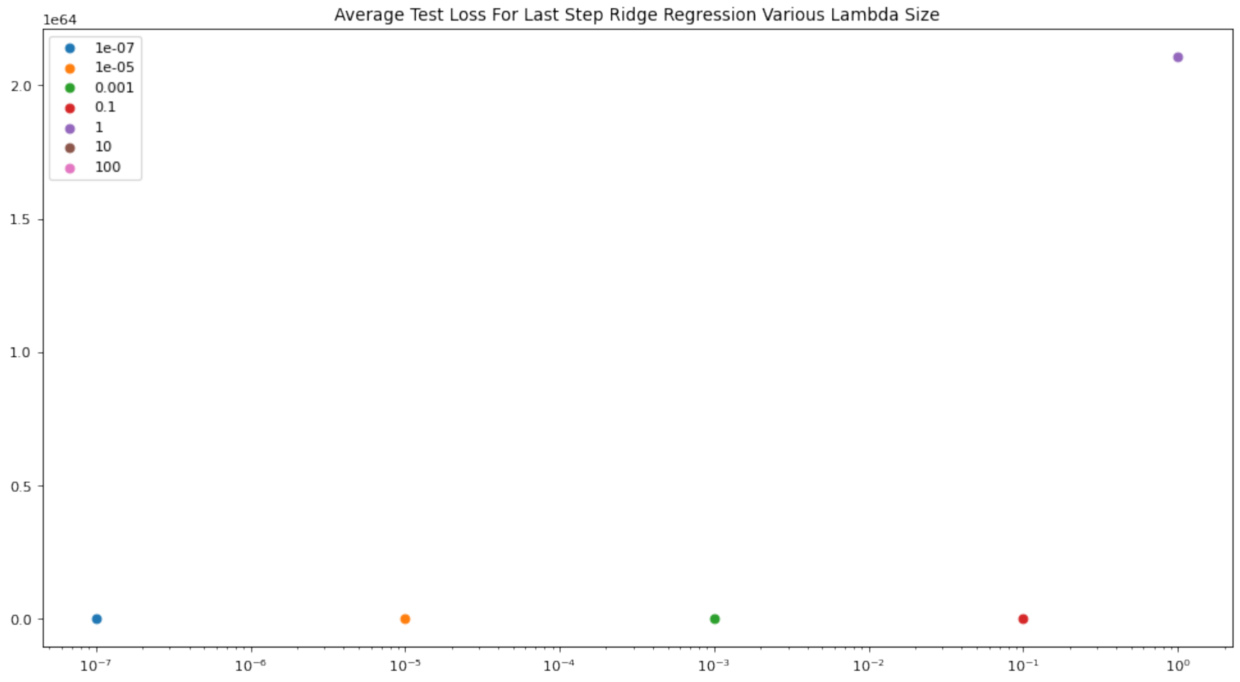
When $\lambda = 10^{-3}$ the testing error is reduced as much as possible at 2.84, while still having great training error only .014 more than the minimum training error.

In [19]:
```
for l in lambdas:
    print('Lambda = ', l, 'Avg Train Loss ', train_loss_dict[l][-1], '
```

```
Lambda =  1e-07 Avg Train Loss  2.077700614426301 Avg Test Loss  2.90
34175759537235
Lambda =  1e-05 Avg Train Loss  2.0778239254952293 Avg Test Loss  2.9
02834530224271
Lambda =  0.001 Avg Train Loss  2.0913500007595984 Avg Test Loss  2.8
47542512245568
Lambda =  0.1 Avg Train Loss  4.245811582850459 Avg Test Loss  2.9577
10971076267
Lambda =  1 Avg Train Loss  2.4907665006408443e+64 Avg Test Loss  2.1
05795019591247e+64
Lambda =  10 Avg Train Loss  inf Avg Test Loss  inf
Lambda =  100 Avg Train Loss  nan Avg Test Loss  nan
```

In [20]:
```
#Begin plotting
plt.figure(figsize=(15, 8), dpi=80)
for l in lambdas:
    plt.scatter(l,average_test_loss[l][-1], label=''.join(str(l)))
#plt.yscale('log')
plt.xscale('log')
plt.legend()
plt.title("Average Test Loss For Last Step Ridge Regression Various La
```

Out[20]:
```
Text(0.5, 1.0, 'Average Test Loss For Last Step Ridge Regression Vari
ous Lambda Size')
```



Question 19

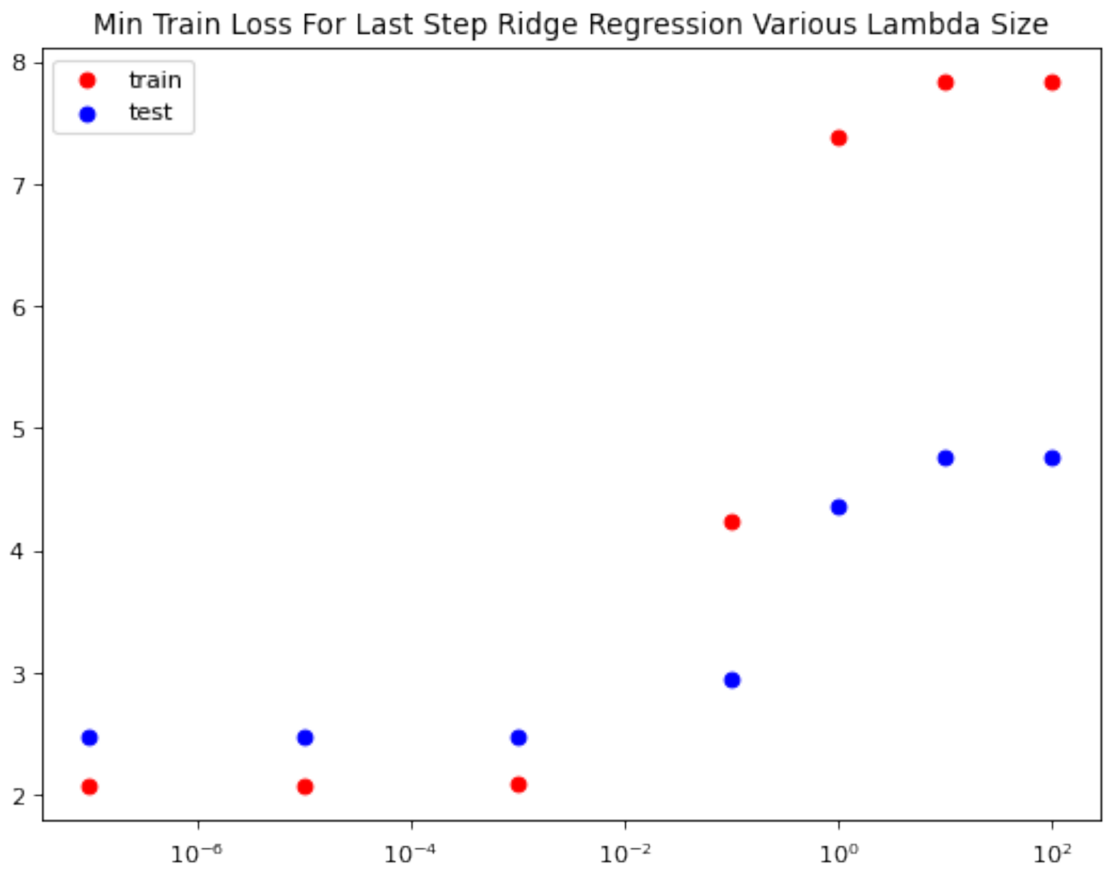
I would still pick $\lambda = 10^{-3}$ as it minimizes both train and test loss the most effectively.

In [21]:
```
for l in lambdas:
    print('Lambda = ', l, 'Min Train Loss ', min((train_loss_dict[l]))
```

```
Lambda =  1e-07 Min Train Loss  2.077700614426301 Min Test Loss  2.47
99523634844363
Lambda =  1e-05 Min Train Loss  2.0778239254952293 Min Test Loss  2.4
799379219810977
Lambda =  0.001 Min Train Loss  2.0913500007595984 Min Test Loss  2.4
784998174321498
Lambda =  0.1 Min Train Loss  4.245811582850459 Min Test Loss  2.9576
980675377906
Lambda =  1 Min Train Loss  7.380047560534102 Min Test Loss  4.371101
634599719
Lambda =  10 Min Train Loss  7.826519529702243 Min Test Loss  4.77149
33968428515
Lambda =  100 Min Train Loss  7.826519529702243 Min Test Loss  4.7714
933968428515
```

In [22]:
```
#Begin plotting
plt.figure(figsize=(8, 6), dpi=80)
for l in lambdas:
    plt.scatter(l,min(train_loss_dict[l]),label='Train',color='r')
    plt.scatter(l,min(average_test_loss[l]), label='Test',color='b')
#plt.yscale('log')
plt.xscale('log')
plt.legend(labels=['train','test'])
plt.title("Min Train Loss For Last Step Ridge Regression Various Lambc
```

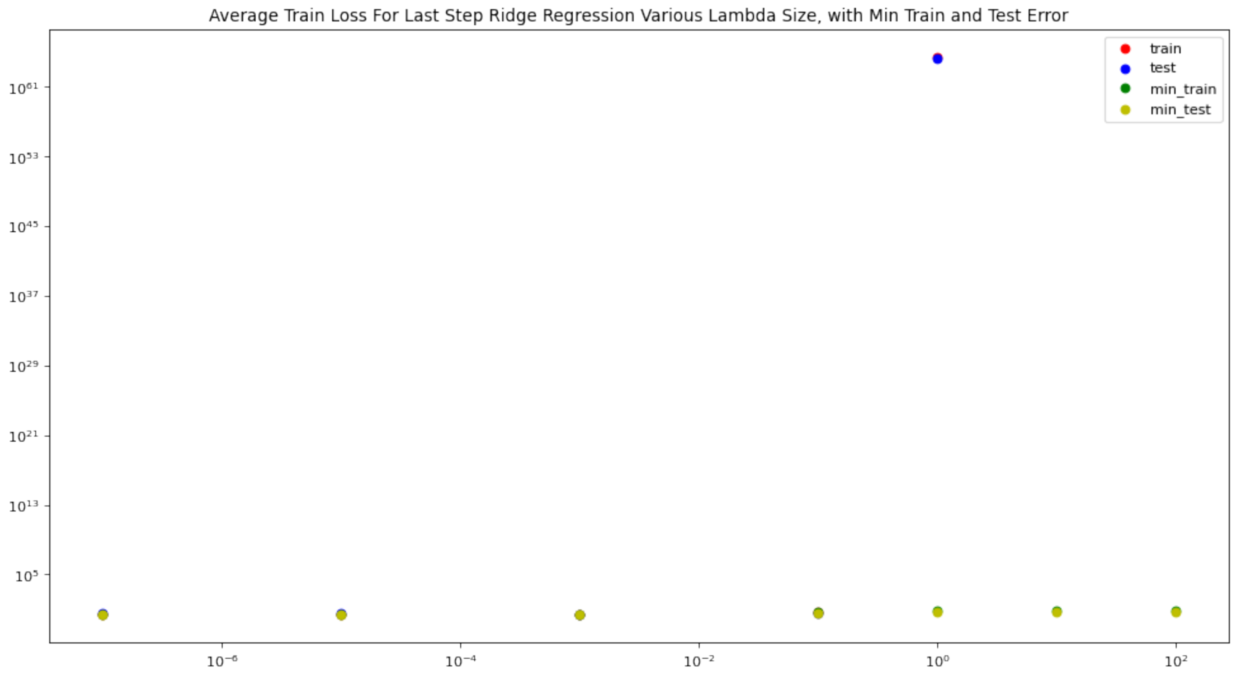
Out[22]:
```
Text(0.5, 1.0, 'Min Train Loss For Last Step Ridge Regression Various
Lambda Size')
```



In [23]:
```
#Begin plotting
plt.figure(figsize=(15, 8), dpi=80)
for l in lambdas:
    plt.scatter(l,train_loss_dict[l][-1], color='r')
    plt.scatter(l,average_test_loss[l][-1], color='b')
    plt.scatter(l,min(train_loss_dict[l]),color='g')
    plt.scatter(l,min(average_test_loss[l]), color='y')
    print('Lambda = ', l, 'Avg Train Loss ', train_loss_dict[l][-1], '
    #print('Lambda = ', l, 'Min Train Loss ', min((train_loss_dict[l])
plt.yscale('log')
plt.xscale('log')
plt.legend(labels=['train','test','min_train','min_test'])
plt.title("Average Train Loss For Last Step Ridge Regression Various L
```

```
Lambda =  1e-07 Avg Train Loss  2.077700614426301 Avg Test Loss  2.90
34175759537235
Lambda =  1e-05 Avg Train Loss  2.0778239254952293 Avg Test Loss  2.9
02834530224271
Lambda =  0.001 Avg Train Loss  2.0913500007595984 Avg Test Loss  2.8
47542512245568
Lambda =  0.1 Avg Train Loss  4.245811582850459 Avg Test Loss  2.9577
10971076267
Lambda =  1 Avg Train Loss  2.4907665006408443e+64 Avg Test Loss  2.1
05795019591247e+64
Lambda =  10 Avg Train Loss  inf Avg Test Loss  inf
Lambda =  100 Avg Train Loss  nan Avg Test Loss  nan
```

Out[23]:
```
Text(0.5, 1.0, 'Average Train Loss For Last Step Ridge Regression Var
ious Lambda Size, with Min Train and Test Error')
```



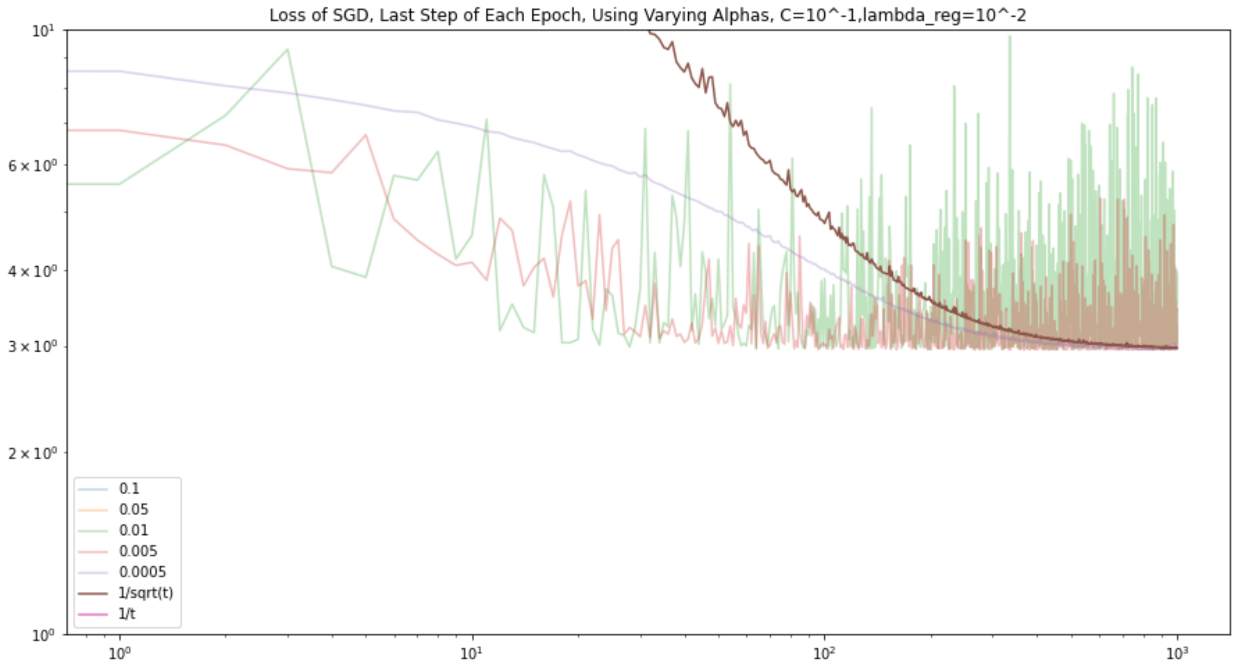
Question 24

```
In [24]: ########################################
         ### Stochastic gradient descent
         def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=10**-2, num_e
             """
             In this question you will implement stochastic gradient descent wi

             Args:
                 X - the feature vector, 2D numpy array of size(num_instances,
                 y - the label vector, 1D numpy array of size(num_instances)
                 alpha - string or float, step size in gradient descent
                         NOTE: In SGD, it's not a good idea to use a fixed step
                         if alpha is a float, then the step size in every step
                         if alpha == "1/sqrt(t)", alpha = 1/sqrt(t).
                         if alpha == "1/t", alpha = 1/t.
                 lambda_reg - the regularization coefficient
                 num_epoch - number of epochs to go through the whole training

             Returns:
                 theta_hist - the history of parameter vector, 3D numpy array o
                              for instance, theta in epoch 0 should be theta_hi
                 loss_hist - the history of loss function vector, 2D numpy arra
             """
             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)) #I
             loss_hist = np.zeros((num_epoch, num_instances)) #Initialize loss_
             #TODO

             #Save helper variable
             m = X.shape[0]

             t=1
             #Iterate over the number of epochs
             for epoch in range(num_epoch):

                 #Shuffle the data so we can sample randomly
                 randomized_data = np.random.permutation(num_instances)
                 for step in range(len(randomized_data)):

                     #Append thetas and loss
                     theta_hist[epoch,step] = theta
                     loss_hist[epoch,step] = compute_square_loss(X,y,theta) + l

                     #Helper variable
                     index = randomized_data[step]
```

```
            X_temp = np.array(X[index]).reshape((1,49))
            y_temp = np.array(y[index]).reshape((1,1))
            theta = theta.reshape((49,1))
            gradient = (2*((X_temp.T@X_temp@theta) - (X_temp.T@y_temp)
            gradient, theta = gradient.reshape((49,)), theta.reshape((

            #Check for alpha value
            if alpha == "1/sqrt(t)":
                #1/sqrt(t) step size
                theta = theta - (gradient * (.1/np.sqrt(t)))
                t += 1
            elif alpha == "1/t":
                #1/t step size
                theta = theta - (gradient * (.1/t))
                t += 1
            else:
                #Fixed step size
                theta = theta - (gradient * alpha)

    #Return values
    return theta_hist, loss_hist
```

Question 25

In [25]:
```python
#Initialize helper variables
lam = 10**-2
alphas = [.1,.05,.01,.005,.0005,'1/sqrt(t)','1/t']
loss_dict = {}
theta_dict = {}
for alpha in alphas:
    loss_dict[alpha] = []
    theta_dict[alpha] = []
#Caclulate Thetas and Loss Dictionaries
for a in alphas:
    theta_dict[a], loss_dict[a] = stochastic_grad_descent(x_train,y_tr

#Plot loss
plt.figure(figsize=(15,8))
for a in alphas:
    if type(a)==float:
        plt.plot(x[:-1],loss_dict[a][:,-1], label=a, alpha=.3)
    else:
        plt.plot(x[:-1],loss_dict[a][:,-1], label=a)
plt.xscale('log')
plt.yscale('log')
plt.title('Loss of SGD, Last Step of Each Epoch, Using Varying Alphas,
plt.legend()
plt.show()
```



In [ ]:

Results Analysis:

As expected, for the static step sizes most diverged, while other estimates were noisey and bounced around the minimum. The function that performed the best (as in was the smoothest) was the dynamic step size of $\frac{C}{\sqrt{t}}$, with $C = .1$. This function converged to the local minimum slower, but in much smoother fashion. This is due to the dynamic step size which dampens the noise and lets SGD avoid bouncing around the local or global minimum, and allows for easier convergence to said minimum. That being said, a static really small step size, in this case $\alpha = .005$, also performed very well, reaching the minimum faster than the dynamic step size while not bouncing around the minimum like its larger static step size peers.

In [26]:
```python
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler


def pre_process_mnist_01():
    """
    Load the mnist datasets, selects the classes 0 and 1
    and normalize the data.
    Args: none
    Outputs:
        X_train: np.array of size (n_training_samples, n_features)
        X_test: np.array of size (n_test_samples, n_features)
        y_train: np.array of size (n_training_samples)
        y_test: np.array of size (n_test_samples)
    """
    X_mnist, y_mnist = fetch_openml('mnist_784', version=1,
                                    return_X_y=True, as_frame=False)
    indicator_01 = (y_mnist == '0') + (y_mnist == '1')
    X_mnist_01 = X_mnist[indicator_01]
    y_mnist_01 = y_mnist[indicator_01]
    X_train, X_test, y_train, y_test = train_test_split(X_mnist_01, y_
                                                        test_size=0.33
                                                        shuffle=False)

    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    y_test = 2 * np.array([int(y) for y in y_test]) - 1
    y_train = 2 * np.array([int(y) for y in y_train]) - 1
    return X_train, X_test, y_train, y_test
```

```
def sub_sample(N_train, X_train, y_train):
    """
    Subsample the training data to keep only N first elements
    Args: none
    Outputs:
        X_train: np.array of size (n_training_samples, n_features)
        X_test: np.array of size (n_test_samples, n_features)
        y_train: np.array of size (n_training_samples)
        y_test: np.array of size (n_test_samples)
    """
    assert N_train <= X_train.shape[0]
    return X_train[:N_train, :], y_train[:N_train]



X_train, X_test, y_train, y_test = pre_process_mnist_01()

clf = SGDClassifier(loss='log', max_iter=1000,
                    tol=1e-3,
                    penalty='l1', alpha=0.01,
                    learning_rate='invscaling',
                    power_t=0.5,
                    eta0=0.01,
                    verbose=1)
clf.fit(X_train, y_train)

# test = classification_error(clf, X_test, y_test)
# train = classification_error(clf, X_train, y_train)
# print('train: ', train, end='\t')
# print('test: ', test)
```

```
-- Epoch 1
Norm: 0.69, NNZs: 289, Bias: 0.003645, T: 9902, Avg. loss: 0.041850
Total training time: 0.03 seconds.
-- Epoch 2
Norm: 0.78, NNZs: 258, Bias: 0.003422, T: 19804, Avg. loss: 0.031808
Total training time: 0.05 seconds.
-- Epoch 3
Norm: 0.84, NNZs: 241, Bias: 0.003509, T: 29706, Avg. loss: 0.030258
Total training time: 0.08 seconds.
-- Epoch 4
Norm: 0.89, NNZs: 233, Bias: 0.003690, T: 39608, Avg. loss: 0.029196
Total training time: 0.11 seconds.
-- Epoch 5
Norm: 0.92, NNZs: 227, Bias: 0.003947, T: 49510, Avg. loss: 0.028678
Total training time: 0.14 seconds.
-- Epoch 6
Norm: 0.96, NNZs: 219, Bias: 0.004242, T: 59412, Avg. loss: 0.028280
```

```
Total training time: 0.17 seconds.
-- Epoch 7
Norm: 0.99, NNZs: 214, Bias: 0.004562, T: 69314, Avg. loss: 0.027930
Total training time: 0.19 seconds.
-- Epoch 8
Norm: 1.01, NNZs: 212, Bias: 0.004903, T: 79216, Avg. loss: 0.027672
Total training time: 0.22 seconds.
-- Epoch 9
Norm: 1.04, NNZs: 208, Bias: 0.005261, T: 89118, Avg. loss: 0.027464
Total training time: 0.25 seconds.
Convergence after 9 epochs took 0.25 seconds
```

```
Out[26]: SGDClassifier(alpha=0.01, eta0=0.01, learning_rate='invscaling', loss
='log',
              penalty='l1', verbose=1)
```

Problem 28

```
In [27]: def classification_error(clf, X, y):
    """
    Input:
    clf: (object) trained classifier model
    X: (np.array) design matrix
    y: (np.array) target values

    Output:
    error: (float) Classification Error
    """
    two_n = 2*len(y)
    summed_error = sum(abs(clf.predict(X)-y))
    error = summed_error / two_n
    return error
```

```
In [28]: print("Classiciation Error:",classification_error(clf, X_test,y_test))
```

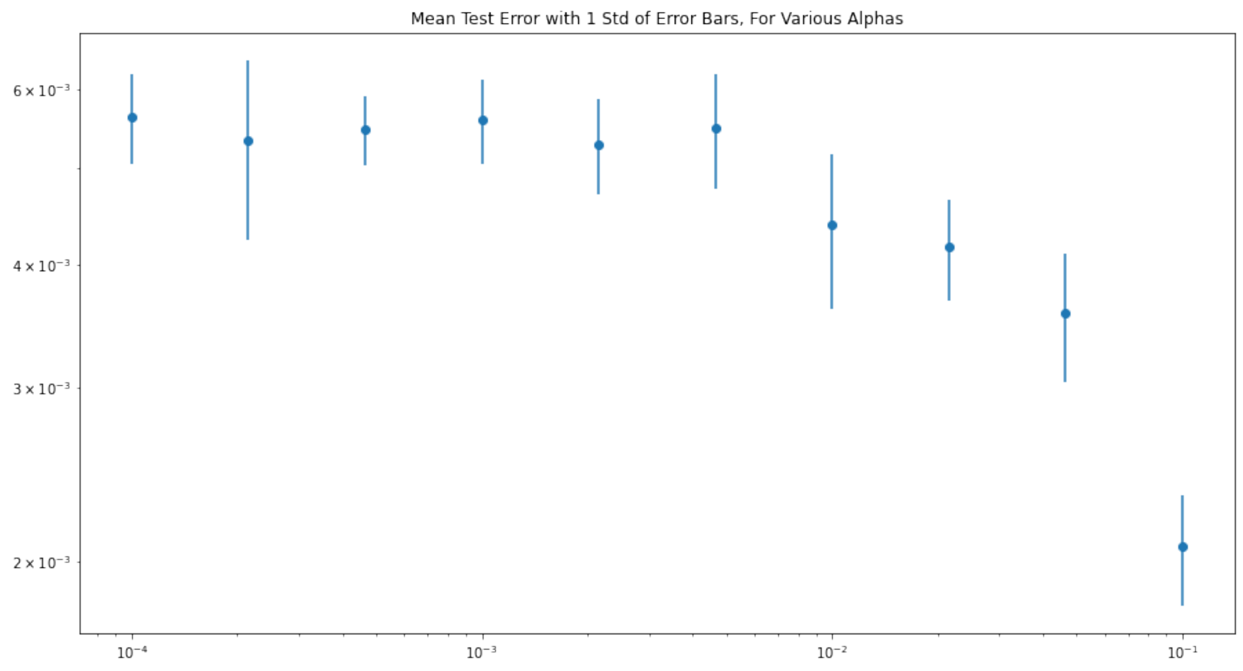
```
Classiciation Error: 0.001025010250102501
```

Problem 29

In [29]:
```
def create_error_by_alpha (X_train, y_train, X_test, y_test, alpha_arr

    #errArray = np.zeros(len(alpha_array)*repeat_times)
    average_arr, std_arr, coeffs_arr = [], [], []

    #Iterate over alpha_array
    for j in range(len(alpha_array)):

        #Create temporary Helper variable
        temp_arr = np.zeros(repeat_times)

        #Initialize classifier
        clf = SGDClassifier(loss='log', max_iter=1000,
                      tol=1e-3,
                      penalty='l1', alpha=alpha_array[j],
                      learning_rate='invscaling',
                      power_t=0.5,
                      eta0=0.01,
                      verbose=0)
        #Iterate for how over many times
        for i in range(repeat_times):
            #Select subsample with 100 points
            x_train, y_train = sub_sample(100, X_train, y_train)
            #Fit model
            clf.fit(x_train, y_train)
            #Get classification error
            temp_arr[i]=classification_error(clf, X_test, y_test)
            #Edge cases where i == 0
            if i ==0:
                coeffs_arr.append(clf.coef_)
        #Calculate mean, std, and append
        average, std = np.mean(temp_arr), np.std(temp_arr)
        #Append
        average_arr.append(average)
        std_arr.append(std)
    #Return the means and standard deviations
    return average_arr, std_arr, np.array(coeffs_arr)
```

In [30]:
```
#Set up optimal Spacing
alpha_array = np.logspace(-4,-1, num =10)
#Create subsamples and calculate average, std, coeffs
average_arr, std_arr, coeffs_arr = create_error_by_alpha(X_train, y_tr
#Plot the figure
plt.figure(figsize = (15,8))
plt.errorbar(alpha_array,average_arr,yerr = std_arr, fmt = 'o')
plt.yscale("log")
plt.xscale('log')
plt.title("Mean Test Error with 1 Std of Error Bars, For Various Alpha
```

Out[30]:
```
Text(0.5, 1.0, 'Mean Test Error with 1 Std of Error Bars, For Various
Alphas')
```



In [31]:
```
#Initialize helper variable
output_arr = []
#Iterate over the alpha values used for regularization
for alpha in range(coeffs_arr.shape[0]):
    #Reshape the array to 28x28 pixels
    reshaped_array = coeffs_arr[alpha][0].reshape(28,28)
    #Append reshaped array
    output_arr.append(reshaped_array)
    #Print alpha and L1 Norm Values
    print("For Alpha ",alpha_array[alpha]," the L1 norm is: ", sum(abs
```

```
For Alpha  0.0001  the L1 norm is:  8.089057040139913
For Alpha  0.00021544346900318845  the L1 norm is:  8.077843494873163
For Alpha  0.00046415888336127773  the L1 norm is:  7.917882542767075
For Alpha  0.001  the L1 norm is:  7.9837497213638065
For Alpha  0.002154434690031882  the L1 norm is:  7.709855371253047
For Alpha  0.004641588833612777  the L1 norm is:  7.0872641049912035
For Alpha  0.01  the L1 norm is:  6.19616177542646
For Alpha  0.021544346900318822  the L1 norm is:  5.089043082646317
For Alpha  0.046415888336127774  the L1 norm is:  3.5123758485032095
For Alpha  0.1  the L1 norm is:  2.1586612335879334
```

Problem 30

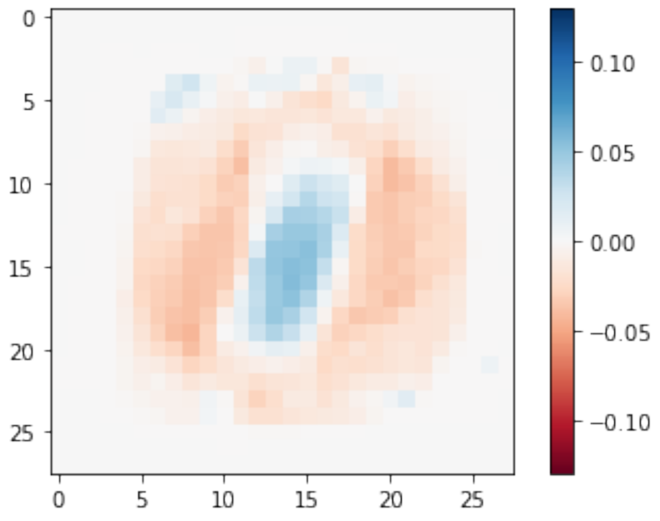
Since we repeated this experiment many times, and thus sampled many times, the main source of randomness we averaged over and eliminated was the randomness inherente in our method of selecting train and testing data.

Problem 31

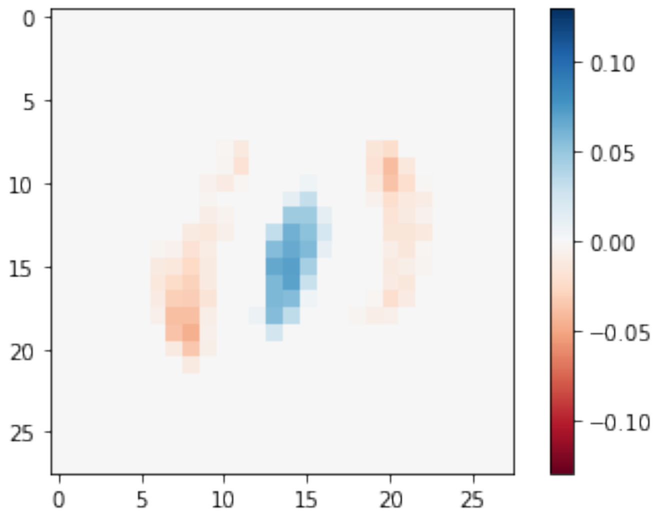
The optimal value for alpha was the one that lowered the testing error the most, which in our case was when $\alpha = 10^{-1}$.

Problem 32

In [32]:
```
#Plot the heat map for the first iteration (Least Regularization)
scale = np.abs(clf.coef_).max()
plt.imshow(output_arr[0], cmap=plt.cm.RdBu, vmax=scale, vmin=-scale )
plt.colorbar()
plt.show()
```



In [33]:
```
#Plot the heat map for the last iteration (Most Regularization)
plt.imshow(output_arr[-1], cmap=plt.cm.RdBu, vmax=scale, vmin=-scale )
plt.colorbar()
plt.show()
```



Problem 33

There is definitely a pattern occuring in θ. As we increase our regularization parameter, α, since we use the L1 norm, more and more coefficients get zeroed out, hence the large amount of gray values present in the heat maps above. Those gray values do not provide any information or predictive value on whether or not our 28×28 image of pixels is a 1 or a 0. In a very interesting and cool fashion, the coefficients in θ that have the smallest value, (the largest absolute value for a negative number), form a symmetrical O shape around the middle of our plotted θ. This makes perfect sense as negative θ coefficient values indicate that those pixels increase the likelihood that the image is a 0 rather than a 1. Conversely, the largely positive values of θ are centered around in a line that extends vertically in the center of the image, completely what we would expect given how most people draw their number 1s.