Assignment 2: More Regressions - Machine Learning 6316

Jacob Dineen / JD5ED

September 28, 2019

Questions and programming address topics covered from weeks 2:4 of Machine Learning 6316 (Fall). *Note that I finished up most of the programming before the updated template was uploaded to canvas - The code file for Ridge Regression has two separate methods for Normal Equation and SGD.

1 Polynomial Regression

1.1 Polynomial Regression Data Generation and Model Fitting

We begin with an underlying polynomial distribution, using DataGeneration.poly, in which the values of x are uniformly sampled n times (where n is fixed to 300), and the values of y are generated by adding some gaussian noise to the underlying distribution, as shown in Figure 1. Note that the stochastic nature of the sampling procedure will ensure that future plots generated from the random distribution will not match exactly, but will be close. I've added some method parameters that allow for saving, but have toggled them off before submission. The data, containing a single independent variable and a response variable, is stored in dataPoly.txt, which we'll split into respective arrays upon read in. After reading the data in, we begin to fill in the method templates, as provided to us. Going in order, we have a method that takes the input array, which is x (with a single feature) and creates a reformulation of the input space based on polynomial basis expansion. E.g., if the degree of the desired input space is 4, we will have a mapping from the input space, $nx1$, to $nx5$, where the basis is represented by $[1, x, x^2, x^3, x^4]$. Transforming the input space to n dimensions allows us to generate a weight matrix that can fit a polynomial/nonlinear regression through the underlying distribution. The gradient descent code, which we generated last week for linear regression, remains the same. *Note that there was an update to the homework assignment which helps explain why I was suffering from exploding gradients. As such, tasks 1,2 and 4 will use the normal equation, while task 3 will use full batch GD. As we covered in the LR homework, full batch gradient descent generally gives the best approximation of the gradient and allows for less noisy updates. Load data, Get Loss and Predict remain the same here, as we're still dealing with a continuous response and likely won't need to tinker until we're dealing with classification.

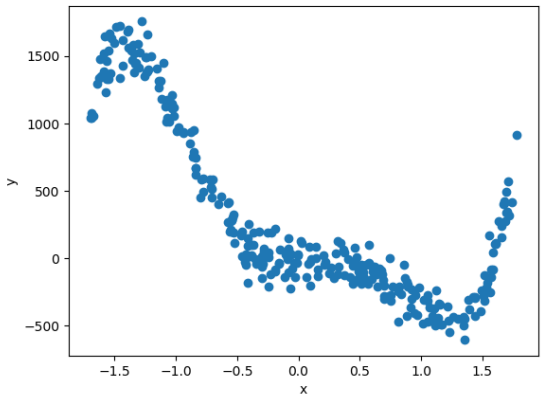


Figure 1: Scatter Plot Showing Generated Polynomial Distribution

Task 1 was to plot the training and validation loss per degree of the polynomial order (figure 1)- We do this to see which order polynomial converges to the most accurate solution, per MSE. Again, this was utilizing the normal equation solver, as overflow became an issue with batch GD updates.

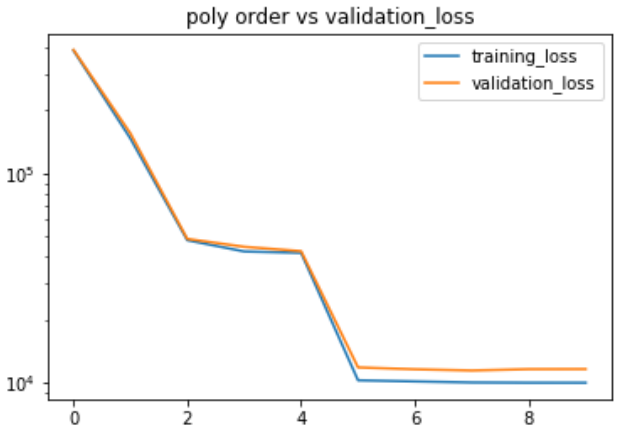


Figure 2: Poly Order VS Loss

We can see that as we increase the complexity of our model (parameters increase), our training and validation loss decreases - There does not appear much to gain by increasing the size of the poly order past 5 (the underlying distribution), but I decided to move forward with d = 8 because the argmin validation loss from the stored list returned such..
min train loss(Batch GD): 10000.54
min validation loss (Batch GD): 11440.67

Task 2 relates to finding the best 'degree' hyperparameter given a list of potential candidates. This controls the degree of the polynomial expansion of the input space (which we scale down to avoid exploding MSE / gradients). The main method here enlists a series of helper functions

concerned with finding the optimal degree parameter, plotting various losses associated with the training and validation set, and plotting the best fit learned nonlinear curve to the original data. As the homework mentions, we want to holdout the test set for evaluating our ability to generalize, so the method to find the optimal poly degree order is fed in the train sets generated from train-test split only. We split this train set into 2 sets to generate a validation set and leave the test set alone here. Eventually, we use the train set and the validation set, concatenated, to retrain our model, given the appropriate hyperparameters, and evaluate performance on the test set (figure 2).

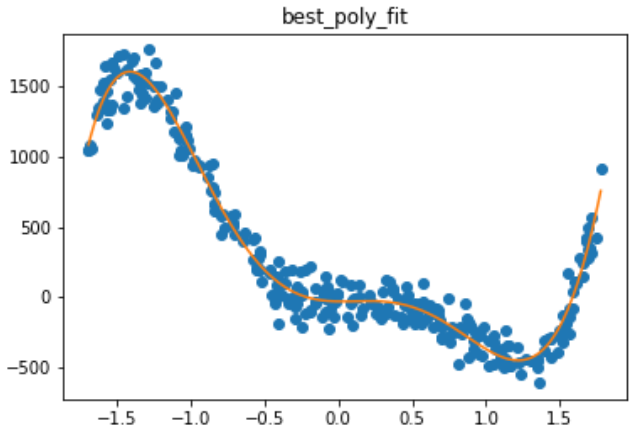


Figure 3: Poly Order - Degree 8 fit

Figure 3 shows the best fit curve of 200 uniformly sampled data points using the numpy linespace call superposed on top of the original data distribution, per request. The model (normal equation thetas) was fit on the concatenated train/validation splits and inference was performed on the sampled data (transformed to desired input space), which resulted in the lowest MSE loss per Figure 2. Explicitly running the holdout (test) data through our model, we end up with an MSE test error of 8,756, or an average residual of 93.57.

best theta via normal equation, degree 8:
$[-29.003, -13.426, 359.135, -1102.023, 37.415, 434.143, -28.833, -25.776, 1.422]$

best theta via batch GD, degree 8, max epochs = 2000, alpha = .0001: [20.26, -48.209, 25.03, -51.86, 29.75, -47.74, 28.75, 1.63, 1.488]

The underlying data distribution, when originally plotted, looked to have about 4 or 5 changes in concavity, so intuition led me to believe that we'd need at least 4 degrees to adequately reduce the distance from predictions to the response. Going back to the data generation script, we see that y was created by using a degree 5 polynomial, with gaussian noise added.

Task 3 also involved the method used in task 2, but this time we used a gradient-based algorithm to fit the data (full batch GD). Our requirement was to fit a model on the train set (80 percent) and use that model on the test set, noting losses for both. I used a learning rate of 0.0001 and a t_{max} = 2000. I should also note that the gradient updates were divided by the

length of the train set, and the inputs were scaled down using a min-max scaler before training/testing*. This helped to reduce some of the numerical overflow that I was struggling with early on. The chart shows both types of losses decreasing at a decreasing rate, although continually training past the epoch threshold appears to be intractable, and the closed-form solution was more applicable to this particular case.

*The min-max scaler was removed after an update to the assignment that scales the data on generation.

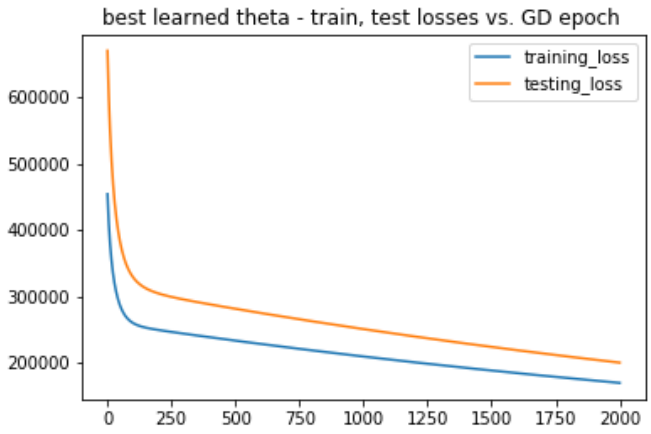


Figure 4: GD lossplot

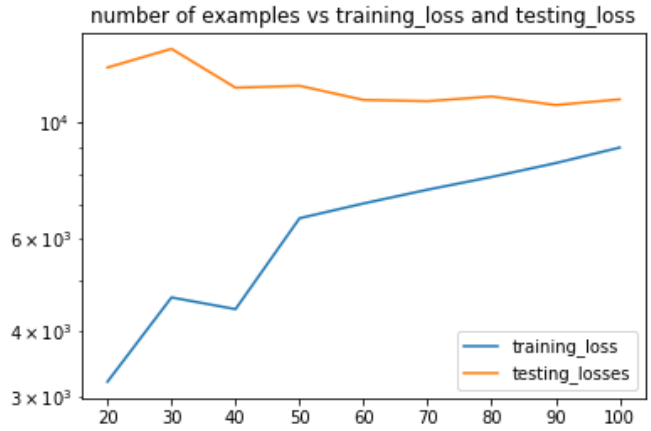


Figure 5: Task 4

Lastly, we circle back to one of the last questions on homework 1, dealing with the impact of train set size on loss. The task is to split the data into 50/50 train/test splits and iterate through a list of scalars containing the portion of the train set that we should use to train our model. Figure 5 shows the number of examples used to train the model on the x-axis, and the resulting losses (MSE) in orange and blue, in relation to the y-axis. As we increase the train size, we see the training loss increase (as we begin to fit to more noise), but our ability to generalize to the

test set continually improves, albeit modestly. This closure of the distance between the two costs can be explored with the bias-variance tradeoff concept which was covered in lecture slides 7. A low training loss and high test loss means that we are overfitting to the training data. When the gap between the training and test loss shrinks or crosses, we begin to underfit the underlying distribution. This task utilized the normal equation on the rescaled underlying distribution, per update. Note that a fellow student's Piazza question led me to redo this part. I originally trained on each range of the example number list, and reported test accuracy on the full test set. The instructor's update on the post: *getlosspertrnumexamples*() explanation led me to believe that this was not the intention, and instead, if the number of examples per the list of examples in *examplenum* is 10, we should train on five data points and test on five. Doing so results in the following plot:

Figure 6: Task 4 Revised

All submitted code should work.

2 Ridge Regression

2.1 Ridge Regression Programming

Code Submitted in RidgeRegression.py

As discussed in class/lecture notes, Ridge regression, or L2 regression, constrains the weights to fall within a circle in the n-dimensional loss surface. If lambda, the tunable parameter, is set to zero, we get out OLS estimator. If lambda approaches infinity, our weights approach zero. We use regularization to inject bias into our model and decay the weights - This oftentimes, particularly with lasso regression (sparse beta vector), leads to more interpretable models. Figure 6 shows the underlying data distribution, generated via the given python file. We plot the single independent variable, x, against y, the response generated via a linear function with

some Gaussian noise. 100 other features are generated, denoted meaningless features, that we hope our estimator will learn to silence with the impending regularization technique.

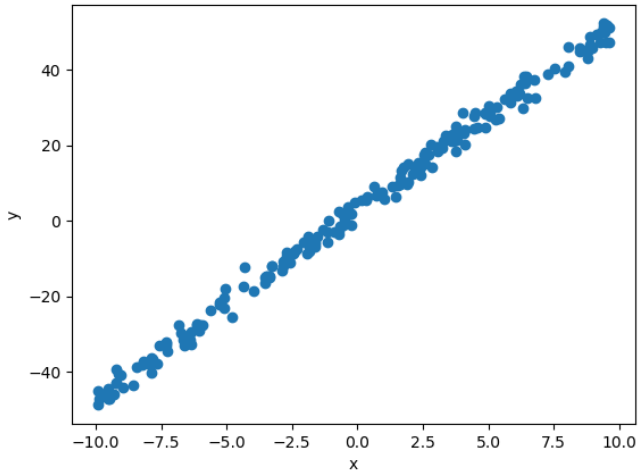


Figure 7: Underlying Distribution - Ridge Generation

Much of this task, side for some extra credit, will use the augmented normal equation. Many of the methods used in the previous section can be reused, side from the OLS normal equation and a function that performs k fold (k fixed to 4) cross-validation. The cross-validation function split the data into 4 even (if possible) arrays, selects 3/4 of that data to train the model using the normal equation, and performs inference on the remaining data. We then take the average loss per iteration (once all folds have been utilized) for both the training and validation sets. Additionally, we want to run this through multiple lambdas (hyperparameter) to see which results in the lowest validation loss. I added some error handling (w/ some assistance from sklearn source code) to try to eliminate some of the errors that would arise if you didn't get an even split, e.g. if you had a remainder after splitting 201 samples. The last sample would be thrown into the last set of splits as additional data to train on.

Figure 7 shows the train and validation loss after performing Kfold CV. The code template calculates the argmin of the validation loss and returns '4' as the optimal lambda which minimizes the val loss. We'll use this to construct our best model, utilizing the full 80 percent of the data to build our model this time.

First, let's look at some of the statistics generated from running the normal equation with various lambdas, remembering that our best lambda was equal to 4.

L2 norm of normal beta: 30.269
L2 norm of best beta: 6.640
L2 norm of large lambda beta: 4.651
Average testing loss for normal beta: 11.031
Average testing loss for best beta: 4.636

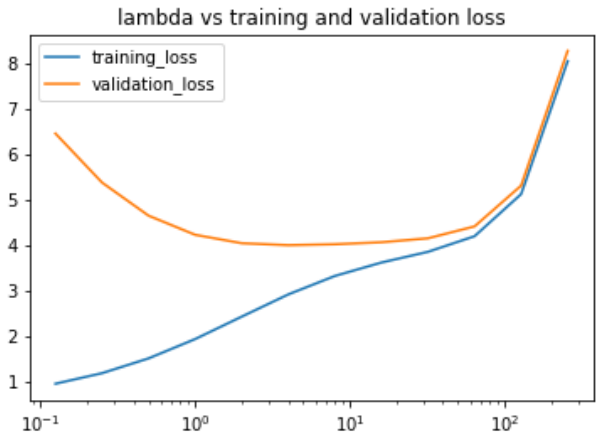


Figure 8: Lambda Vs Loss on Kfold CV

Average testing loss for large lambda beta: 12.126

The normal beta gives us our OLS solution (we end up multiplying the L2 Norm by the lambda param), so with the gaussian noise added to the underlying distribution, it's not particularly surprising that we overfit to the data and generate the highest loss. The best beta was found via the hyperparameter tuning step and resulted in the lowest validation error. Looking at the beta matrices, we can see the effect of lambda on the size of our weights through each of the 3 variants above. In the normal beta, the variance of weights is high, and each feature impacts the model in some way. In the best beta, we see the impact is being driven by the first two features, and in the large beta calculation, we see the second feature driving most of the prediction power, squashing the intercept term down quite a bit and leading to increased val loss.

We now retrain our model (normal equation) using the concatenated train/validation arrays, expanding our train size, and perform inference on the untouched test set, resulting in:

Training Loss w/ best beta: 1.807
Test Loss w/ best beta : 2.039

Figure 9, below, helps to show the feature weighting with our best learned β vector.

The last task involving Ridge Regression was to try to perform the above using a gradient-based learning algorithm and compare the results to the augmented normal equation. I was able to get build something that resembled the best lambda normal equation solution but wasn't quite able to get to the optimal intercept in 25k iterations of batch GD (learning rate of 0.001). I ended up scaling this down so you wouldn't have to wait a minute plus while running the code. I think this is because I performed all weight updates over the beta vector, and I believe that the intercept term needs to be excluded/treated as a special case. The only real change from GD in linear regression to Ridge regression is the addition of the regularization term, which is the beta vector squared, and the lambda. Differentiating β^2 gives 2β, thus our

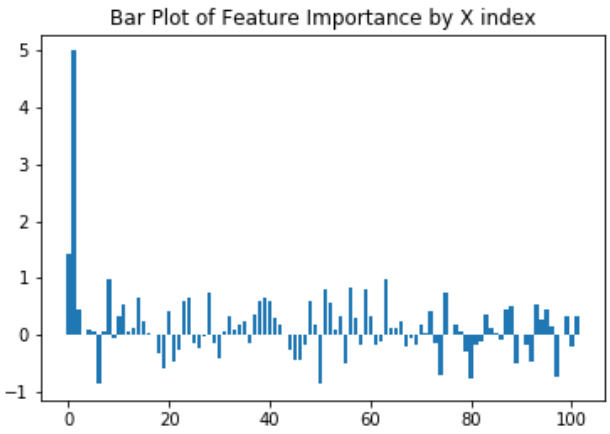


Figure 9: Feature Importance (weight) vs feature index

weight update at each step looks something like: $\beta_{t+1} = \beta_t - \frac{\alpha}{size}(y - ypred)(x) + 2\lambda\beta$. This new model was actually able to decrease loss from the closed form solution provided above, or approach it. See Figure 10.

Training Loss w/ best beta: 2.95
Test Loss w/ best beta : 4.63
gradient descent, betas: [0.622 5.01 0.195 0.159 0.018]
gradient descent test loss : 4.618
normal equation, betas: [1.409 5.002 0.441 0.008 0.097]

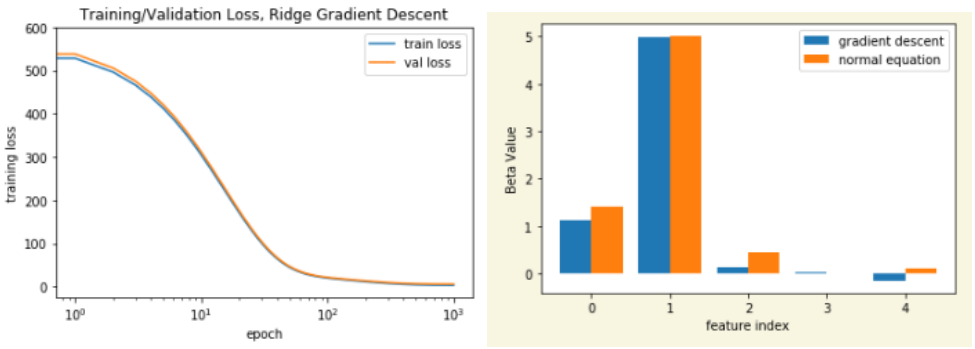


Figure 10: Gradient Descent Behvaior + Feature Importance (weight) vs feature index (200k epochs on GD)

2.2 Ridge Regression QA: 3 Questions

2.2.1

Please provide the math derivation procedure for ridge regression (shown in Figure 1)

Figure 1 : Ridge Regression / Solution Derivation /1.1
If not invertible, a solution is to add a small element
to diagonal $\hat{Y} = \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_p x_p$ Basic Model,

$$\beta^* = \left(X^T X + \lambda I\right)^{-1} X^T \overline{y}$$

The ridge estimator is solution from

$$\hat{\beta}^{rdge} = \text{argmin}(y - X\beta)^T (y - X\beta) + \lambda \beta^T \beta$$

Let us first derive the normal equation for Ordinary Least Squares, as many of the steps remain the same when generalizing to Ridge Regression.

OLS Regression Derivation Procedure:

$$J(\theta) = (X\theta - y)^2$$

$$= (X\theta - y)^T (X\theta - y)$$

$$= (\theta^T X^T - y^T)(X\theta - y)$$

$$= \theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^T y$$

The middle two terms are equivalent due to matrix transpose rules: $\theta^T X^T y = y^T X\theta$, thus:

$$= \theta^T X^T X\theta - 2\theta^T X^T y + y^T y$$

Now we wish to take the gradient of our loss function, J, given θ:

$$\nabla_\theta J = \nabla_\theta \theta^T X^T X\theta - 2\nabla_\theta \theta^T X^T y + \nabla_\theta y^T y$$

$$\nabla_\theta J = 2X^T X\theta - 2X^T y + 0$$

We then set the gradient of J to 0, shift the second term, divide to remove the constant, and isolate/solve for θ

$$2X^T X\theta = 2X^T y$$

$$X^T X\theta = X^T y$$

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

Ridge Regression Derivation Procedure:
We replace our θ terms with β here to avoid confusion, noting that this still refers to our weight matrix.

$$J(\beta) = (X\beta - y)^2 + \lambda \beta^T \beta$$

$$= (\beta^T X^T - y^T)(X\beta - y) + \lambda\beta^T\beta$$
$$= \beta^T X^T X\beta - \beta^T X^T y - y^T X\beta + y^T y + \lambda\beta^T\beta$$
$$= \beta^T X^T X\beta - 2\beta^T X^T y + y^T y + \lambda\beta^T\beta$$

Here, we can combine the first and the fourth terms as such:

$$\beta^T X^T X\beta + \lambda\beta^T\beta = \beta^T(X^T X + \lambda I)\beta$$

$$= \beta^T(X^T X + \lambda I)\beta - 2\beta^T X^T y + y^T y$$

$$\nabla_\beta J = \nabla_\beta \beta^T(X^T X + \lambda I)\beta - 2\nabla_\beta \beta^T X^T y + \nabla_\beta y^T y$$
$$\nabla_\beta J = 2(X^T X + \lambda I)\beta - 2X^T y$$

We then set the gradient of J to 0, shift the second term, divide to remove the constant, and isolate/solve for β

$$(X^T X + \lambda I)\beta = X^T y$$
$$\beta = (X^T X + \lambda I)^{-1} X^T y$$

There are a few non-intuitive matrix differentiation rules, particularly regarding $\nabla_\beta J = 2(X^T X + \lambda I)\beta$ and $\nabla_\theta \theta^T X^T y$ for which the Matrix Cookbook [1] proved valuable.

2.2.2

Suppose $X = \begin{bmatrix} 1 & 2 \\ 3 & 6 \\ 5 & 10 \end{bmatrix}$ and $Y = [1, 2, 3]^T$, could this problem be solved through linear regression?
Please provide your reasons.
(Hint: just use the normal equation to explain the reason)

When we try to solve the following through the OLS normal equation, we see that $X^T X$, a 2x2 matrix, produces a zero determinant, making it a singular matrix, thus a non-invertible matrix (The resulting matrix is not linearly independent, as -2*R2 = R1).

$$\det(x^\top x) = \begin{bmatrix} 140 & -70 \\ -70 & 35 \end{bmatrix} = AD - BC = 4900 - 4900 = 0$$

As we've seen with previous examples in class, λ is our parameter of interest when trying to invert singular, non-invertible matrices for Regression - The goal is to augment the diagonals of the $X^T X$ matrices with a tiny positive element to force rank. As λ approaches 0, we converge to our OLS solution (assuming full rank), and as λ approaches infinity, our weight matrix will inevitably converge to 0. lambda is a tunable parameter, which requires extensive cross validation to measure against the bias-variance trade-off - The more we force our weights

closer to zero, the more bias we are injecting into our model. For the sake of this question, let's assume we are fitting a ridge estimator to our data, and that our λ is set to 0.1.

$$\beta = (X^T X + \lambda I)^{-1} X^T y$$

$$\lambda = 0.1$$

$$\lambda I = \begin{bmatrix} 0.1 & 0.0 \\ 0.0 & 0.1 \end{bmatrix}$$

$$(x^\top x + \lambda I) = \begin{bmatrix} 35 & 70 \\ 70 & 140 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.0 \\ 0.0 & 0.1 \end{bmatrix} = \begin{bmatrix} 35.1 & 70 \\ 70 & 140.1 \end{bmatrix}$$

$$(x^r x + \lambda I)^{-1} \approx \begin{bmatrix} -8 & -4 \\ -4 & -2 \end{bmatrix}$$

$$\beta = (x^\top x + \lambda I)^{-1} x^\top y = \begin{bmatrix} 0.123 \\ 0.25 \end{bmatrix}$$

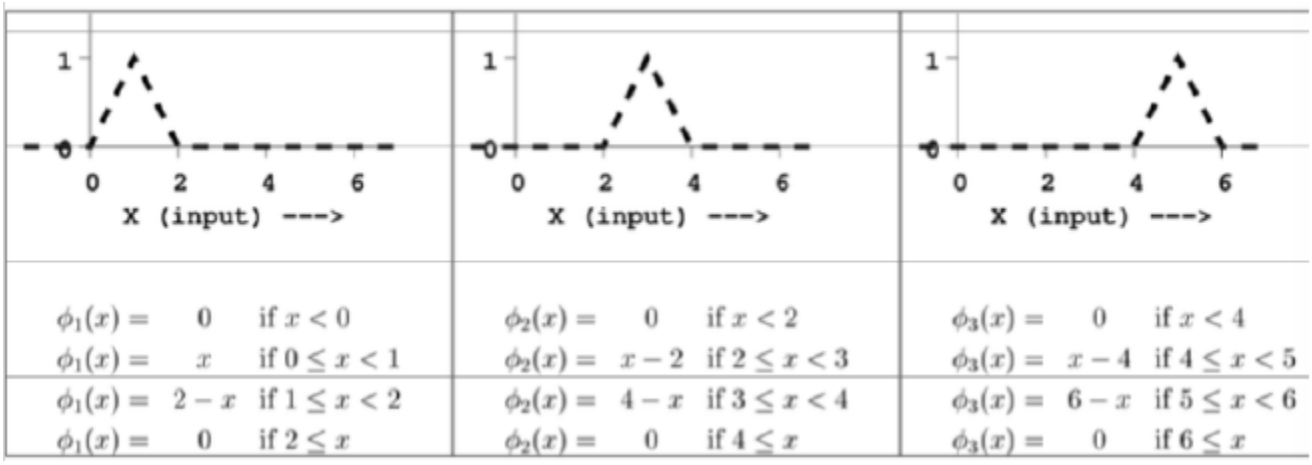
2.2.3

· If you have the prior knowledge that the coefficient β should be sparse, which regularized linear regression method should be chosen to use? (Hint: sparse vector)

As the slides and the lectures associated with regularization discussed, regularization is a technique to keep the learn-able parameters, in this notation β, more regular/normal and interpretative to model-builders. That being said, if we were looking to mitigate the impact of features that didn't quite lend their hand to intuitive explaining the functional mapping from input to output through the use of a sparse weight vector/matrix, we'd want to utilize the Lasso/Squared Loss+L1 Norm estimator. While both forms of regularization will decay the weights to smaller values, Lasso regression is subject to the constraint that the sum of L1 β norms is less than or equal to s. Where the constraint imposed by the L2/Ridge Regression was that the weights hand to fall within a circle, L1 forces the weights to fall within a diamond. The intersection, in some n-dimensional space, of this constraint and the loss surface of J(θ) is likely to occur on an axis (If tuning parameter is set small enough), thus imposing some weights will be zero. Lasso regression, per the slides, performs variable selection and shrinkage simultaneously.

3 Sample Questions

3.1 Question 1. Basis functions for regression



$\phi_1(x) = 0$ if $x < 0$
$\phi_1(x) = x$ if $0 \leq x < 1$
$\phi_1(x) = 2 - x$ if $1 \leq x < 2$
$\phi_1(x) = 0$ if $2 \leq x$

$\phi_2(x) = 0$ if $x < 2$
$\phi_2(x) = x - 2$ if $2 \leq x < 3$
$\phi_2(x) = 4 - x$ if $3 \leq x < 4$
$\phi_2(x) = 0$ if $4 \leq x$

$\phi_3(x) = 0$ if $x < 4$
$\phi_3(x) = x - 4$ if $4 \leq x < 5$
$\phi_3(x) = 6 - x$ if $5 \leq x < 6$
$\phi_3(x) = 0$ if $6 \leq x$

Figure 2 : Basis functions for regression (c) with one real-valued input (x as horizontal axis) and one real-valued output (y as vertical axis).
We plan to run regression with the basis functions shown as above, i.e.t $y = \beta_1\phi_1(x) + \beta_2\phi_2(x) + \beta_3\phi_3(x)$ functions to use ? If "yes", explain their prime advantage. If no, explain their biggest drawback. (1 to 2 sentences of explanation are expected.)

In the case of this problem, we have 3 learnable parameters, and we are assuming that x comes from a distribution that is constrained between 1 and 5. We briefly covered linear piecewise basis functions during our talk on nonlinear regression, but didn't dive too deeply into it. The first basis function almost guarantees that the linear combination of $\beta_1\phi_1(x)$ will equal zero unless x = 1. The second basis function sends x to zero in all cases except if x =3, and the third basis function send x to zero unless x = 5. We can see that if we had an input of 2 or 4 drawn from our data distribution, we'd have:

$$f(2) = \beta_1(0) + \beta_2(0) + \beta_3(0) = 0$$

$$f(4) = \beta_1(0) + \beta_2(0) + \beta_3(0) = 0$$

e.g., the output of our estimator would be forced to predict a response of 0 because our weights, no matter what they were, would be multipled by $\phi(x) = 0$.

3.2 Question 2. Polynomial Regression

Suppose you are given a labeled dataset (with one real-valued input and one real-valued output) including points as shown in Figure 3 :

(a) Assuming there is no bias term in our regression model and we fit a quadratic polynomial regression (i.e. the model is $y = \beta_1 x + \beta_2 x^2\Big)$ on the data, what is the mean squared LOOCV (leave one out cross validation) error?

Since there is no bias term included in this problem, we can derive the weight vector using the normal equation and storing our inputs into a 2x2 matrix, in which column one is the value of x, and column 2 is the value of x squared.

$$\text{Loocv}_1 : \begin{bmatrix} 2 & 4 \\ 3 & 9 \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \end{bmatrix} \qquad \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\text{Loocv}_2 : \begin{bmatrix} 1 & 1 \\ 3 & 9 \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 9 \end{bmatrix} \qquad \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\text{Loocv}_3 : \begin{bmatrix} 1 & 1 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \end{bmatrix} \qquad \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

What we've done above is construct 3 degree-2 polynomial estimators - The operations that would follow to compute the loss for each estimator are trivial, as we'd see the first term β_1 (with a value of 0 across all three models) would squash the first term/linear combination, leaving our output as $\beta 1 x^2$. Because we can eyeball and see that the output is exactly the squared input for all data points, we could say that the LOOCV error for all individual models is 0. Dividing by 3 to get the Mean LOOCV error still gives us 0.

$$\frac{0 + 0 + 0}{3} = 0$$

References

[1] Petersen, K. B. Pedersen, M. S. (2008). The Matrix Cookbook Technical University of Denmark (Version 20081110)