B555: Assignment 2

Pawan Patel

October 14, 2015

1. Problem 1:

a)

$$\text{ML Estimate } = \text{argmax}_\lambda \, P(X_1, ..., X_n|\lambda) = \text{argmax}_\lambda \prod_{i=1}^{n} P(X_i|\lambda)$$

$$= \text{argmax}_\lambda \prod_{i=1}^{n} \frac{\lambda^{X_i} e^{-\lambda}}{X_i!} = \text{argmax}_\lambda \sum_{i=1}^{n} X_i \ln(\lambda) - \lambda - \ln(X_i!)$$

Where the last equality follows from monotonicity of the natural log. Taking derivatives and setting equal to zero to maximize, we get the ML estimate $\hat{\lambda}$:

$$\sum_{i=1}^{n} (\frac{X_i}{\hat{\lambda}} - 1) = 0 \implies \hat{\lambda} = \frac{\sum_{i=1}^{n} X_i}{n} = \frac{79}{9}$$

b)

$$\text{MAP Estimate } = \text{argmax}_\lambda \, P(X_1, ..., X_n|\lambda)P(\lambda) = \text{argmax}_\lambda \prod_{i=1}^{n} P(X_i|\lambda)P(\lambda)$$

$$= \text{argmax}_\lambda (\prod_{i=1}^{n} \frac{\lambda^{X_i} e^{-\lambda}}{X_i!})(\frac{1}{2} e^{-\frac{1}{2}\lambda}) = \text{argmax}_\lambda \sum_{i=1}^{n} X_i \ln(\lambda)P(\lambda) - \lambda - \ln(X_i!) + \ln\frac{1}{2} - \frac{1}{2}\lambda$$

Again, we take derivatives and set equal to get the MAP Estimate $\hat{\lambda}$:

$$\sum_{i=1}^{n} (\frac{X_i}{\hat{\lambda}} - 1) - \frac{1}{2} \implies \hat{\lambda} = \frac{\sum_{i=1}^{n} X_i}{n + \frac{1}{2}} = \frac{79}{9.5}$$

c) As θ decreases, the mass is pushed to the right under the density of the exponential distribution. Thus, as θ decreases, more weighting will be given to the possibility of larger values of λ. Thus, if one is certain $\lambda = 4$ is high estimate for λ, then one should set a large θ so most of the mass is less than 4.

2. Problem 2:

a)

$$\begin{aligned}\text{MAP Estimate } &= \text{argmax}_\theta\, P(X_1,...,X_n|\theta)P(\theta) = \text{argmax}_\theta \prod_{i=1}^{n} P(X_i|\theta)P(\theta) \\ &= \text{argmax}_\theta(\prod_{i=1}^{n} \frac{1}{\sigma_0\sqrt{2\pi}} e^{-\frac{(X_i-\theta)^2}{2\sigma_0^2}})(\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(\mu-\theta)^2}{2\sigma^2}}) \\ = \text{argmax}_\theta(\sum_{i=1}^{n} \ln(\frac{1}{\sigma_0\sqrt{2\pi}}) - \frac{1}{2\sigma_0^2}(X_i-\theta)^2) + \ln(\frac{1}{\sigma\sqrt{2\pi}}) - \frac{1}{2\sigma^2}(\mu-\theta)^2 \\ &= \text{argmax}_\theta -(\sum_{i=1}^{n} \frac{1}{2\sigma_0^2}(X_i-\theta)^2) - \frac{1}{2\sigma^2}(\mu-\theta)^2\end{aligned}$$

Taking derivatives, setting equal to 0 and solving, we get the MAP Estimate $\hat{\theta}$:

$$\begin{aligned}\sum_{i=1}^{n} \frac{1}{\sigma_0^2}(X_i-\hat{\theta}) + \frac{1}{\sigma^2}(\mu-\hat{\theta}) &= 0 \\ \implies \hat{\theta} &= \frac{\frac{1}{\sigma_0^2}\sum_{i=1}^{n} X_i + \frac{1}{\sigma^2}\mu}{\frac{1}{\sigma_0^2}n + \frac{1}{\sigma^2}}\end{aligned}$$

b)

$$\begin{aligned}\text{MAP Estimate } &= \text{argmax}_\theta\, P(X_1,...,X_n|\theta)P(\theta) = \text{argmax}_\theta \prod_{i=1}^{n} P(X_i|\theta)P(\theta) \\ &= \text{argmax}_\theta(\prod_{i=1}^{n} \frac{1}{\sigma_0\sqrt{2\pi}} e^{-\frac{(X_i-\theta)^2}{2\sigma_0^2}})(\frac{1}{2b} e^{-\frac{|\theta|}{b}}) \\ = \text{argmax}_\theta(\sum_{i=1}^{n} \ln(\frac{1}{\sigma_0\sqrt{2\pi}}) - \frac{1}{2\sigma_0^2}(X_i-\theta)^2) + \ln(\frac{1}{2b}) - \frac{1}{b}|\theta| \\ &= \text{argmax}_\theta -(\sum_{i=1}^{n} \frac{1}{2\sigma_0^2}(X_i-\theta)^2) - \frac{1}{b}|\theta|\end{aligned}$$

Again, we take derivatives, set equal to 0 and solve to get a MAP Estimate $\hat{\theta}$:

$$\begin{aligned}\sum_{i=1}^{n} \frac{1}{\sigma_0^2}(X_i-\hat{\theta}) - \frac{1}{b}\text{sign}(\hat{\theta}) &= 0 \\ \implies \hat{\theta} &= \frac{\frac{1}{\sigma_0^2}\sum_{i=1}^{n} X_i - \frac{1}{b}\text{sign}(\hat{\theta})}{\frac{1}{\sigma_0^2}n}\end{aligned}$$

c)

$$\text{MAP Estimate } = \text{argmax}_\theta \, P(X_1, ..., X_n|\theta)P(\theta) = \text{argmax}_\theta \prod_{i=1}^{n} P(X_i|\theta)P(\theta)$$

$$= \text{argmax}_\theta (\prod_{i=1}^{n} \frac{1}{\sqrt{(2\pi)^d |I|}} e^{-\frac{1}{2}(X_i-\theta)^T(X_i-\theta)}) (\frac{1}{\sqrt{(2\pi)^d |\frac{1}{\sigma^2} I|}} e^{-\frac{1}{2}\theta^T(\frac{1}{\sigma^2} I)\theta})$$

$$= \text{argmax}_\theta (\sum_{i=1}^{n} \ln(\frac{1}{\sqrt{(2\pi)^d}}) - \frac{1}{2}(X_i-\theta)^T(X_i-\theta)) + \ln(\frac{1}{\sqrt{(2\pi)^d \sigma^2 d}}) - \frac{1}{2}\theta^T\theta$$

$$= \text{argmax}_\theta (\sum_{i=1}^{n} -\frac{1}{2}(X_i-\theta)^T(X_i-\theta)) - \frac{1}{2}\theta^T\theta$$

We take derivatives and set equal to 0, in order to maximize. We get:

$$\sum_{i=1}^{n}(X_i - \hat{\theta}) - \frac{1}{\sigma^2}\hat{\theta} = 0 \implies \hat{\theta} = \frac{\sum_{i=1}^{n} X_i}{n + \frac{1}{\sigma^2}} \in \mathbb{R}^d$$

Note: There is a closed form solution here as $\Sigma_0 = I$ and multiplication by $\Sigma = \frac{1}{\sigma^2} I$ ammounts to multiplication by a constant. Effectively, the X_i are independent and the components of θ are independent, so one is able to get a solution component-wise.

3. Problem 3:

First, notice that $f(x_1, x_2) = 100(x_2 - x_1)^2 + (1 - x_1)^2$ is the sum of two squares and is therefore nonnegative. Thus if it acheives 0, then that is the minimum. Thus, the minimum is 0 at the point $(1, 1)$, and this is the only point that achieves the minimum. This will serve as a check for the algorithms we perform.

In order to complete the algorithms, we must compute the gradient and inverse hessian:

$$\nabla f = [400x_1^3 - 400x_1x_2 + 2x_1 - 2, -200x_1^2 + 200x_2]$$

and

$$Hf = \begin{bmatrix} 1200x_1^2 - 400x_2 + 2 & -400x_1 \\ -400x_1 & 200 \end{bmatrix}$$

Which gives us,

$$H^{-1}f = \frac{1}{200(1200x_1^2 - 400x_2 + 2) - (400x_1)^2} \begin{bmatrix} 200 & 400x_1 \\ 400x_1 & 1200x_1^2 - 400x_2 + 2 \end{bmatrix}$$

i) We first consider the gradient descent algorithm:

Starting at the point (1.2, 1.2) and using a step size of 1, .1, or .01 with 100, 1000, or 10,000 iterations all yield NaN values for both the minimum achieved and minimum point. This is because the gradient is very large and the minimum is always greatly overshot. When the step size is reduced to .001, we get good results. With 100 iterations, we get a min of .012 at (1.1, 1.2). With 1000 iterations we get a min of 6×10^{-3} at (1.07. 1.16). And, with 10000 iterations, we get a min of 6×10^{-6} (1.002, 1.005)

Starting at (-1.2, 1) we get a similar issue of NaNs using step sizes of 1, .1, or .01 with 100, 1000, or 10000 iterations. But, with a step size of .001, we get better answers. With 100 iterations we get a min of 3.8 at (-.948,.9), with 1000 iterations we get a min of .45 at (.327,.104), and with 10000 iterations we get a min of 5×10^{-5} at (.992, .985).

ii) We now consider Newton's method:

This algorithm is predictably much better. Starting at (1.2, 1.2), with a step size of 1, in just 50 iterations we can get a minimum of 4.7×10^{-8} at (1.00002, 1.00002). In 100 iterations python returns a min of 0 at (1.1). A smaller step size only hurts, as it takes us more iterations to get to the minimum because we traverse along the gradient very slowly. With a step size of .1 and with 100 iterations we only get to a min of 3.4 at (1.16, 1.16). Increasing the number of iterations to 1000 gets us to a min of 0 at (1,1).

Starting at (-1.2, 1) we encounter problems. I cannot get the algorithm to converge to the solution. Indeed, (-1.2, 1) lies in the second quadrant and the parabola which describes the points at which the Hessian is non-invertible, sits in between this point and the global minimum (1,1). So, it is not surprising that this should be a problematic point. In this situation, I would use regular gradient descent which converges to the solution from this point so we can avoid this issue.

4. a)

As we increase the number of features, we get an error because a solution for w^* cannot be computed as the matrix X^TX becomes singular. We can remedy this in a few ways: by adding a regularization term so that we push singular values of X^TX away from zero (as we will do in part b), or we can do a singular value decomposition and discard linear combinations of features with zero eigenvalue. Along the lines of the latter, we can do a principal component analysis and select linear combinations of features which account for a large amount of variation in the data.

b)

We add a regularization term, which changes the solution to $w^* = (X^TX + \lambda I)^{-1}X^Ty$. The λ allows us to take the inverse so that we do not have issues with invertibility of the matrix. We split the 5000 data points into 10 subsets of 500. We train on 9 of them and test on the last, and compute the average error accross all choices of the test subset. As baselines, Linear Regression with 3 features gives us an error of 598.48 and Mean regression gives us 666.46. Note, Linear Regression with 10 features gives a much worse error, so we use Linear Regression with 3 features as a good baseline. Using Ridge Regression on all the feautres, we try values of $\lambda = .01, .1, 1, 10, 100, 1000$, and get errors of 585, 584, 577, 558, 547, 566. We can see that as λ increases, the error does drop as large values are punished. However, at $\lambda = 1000$, we gain error, showing that too large a λ is also problematic. We settle on a $\lambda = 100$, though more attempts with values between 100 and 1000 will probably yield an optimal choice of λ.

c)

There are many ways to subselect 10 features. In my implementation, I do a principal component analysis on the data and then represent the data using the first ten principal components, corresponding to the 10 linear combinations of all the features, that are themselves orthogonal, which account for the most variation amongst the data. In the code, one can choose an option as pca, which will run this analysis. This will represent the data using the first 10 principal components, and then do an analysis similar to part (b), in wich mean, random, linear regression and ridge regression are run on these 10 features on 10 splits of the data and the average error is reported. Note that to run on 10 features, one must remember to set self.features to 10 in both FSLinearRegression and RidgeRegression and algorithms.py or one will get an error. This analysis is very promising, as Linear Regression now reports an error of 171.44 and Ridge Regression with $\lambda = 100$ reports an error of 171.26. Both are big improvements from before. Notice, Ridge Regression does not do so much better than Linear Regression in this case. In this case, since the 10 principal components are linearly independent, we are not going to have issues with invertibility and so a regularization term is unnecesary.

d)

I implement stochastic gradient descent. In the code, one simply sets the option to sgd and runs the program. The default number of features to use is 10. This function calls the original splitdataset() to get back a random ordering of the data, using 4500

samples to train and 500 to test. We then go through the all of the training data entirely for some number of passes, using the stochastic gradient descent algorithm. The three parameters that can affect performance here are the number of passes made through the entire data set, the values one initializes the first w to, and the step size alpha. This algorithm is difficult to get working well. An intial w made very close to zero seems to perform the best as it keeps the error from blowing up to a NaN value in the first few iterations. Moreover, a very small step size of alpha is preferable for the same reason. The number of passes is set to 10 as one pass can do somewhat poorly with only one pass throught he data. Even with 10 passes, one can get errors ranging from as good as 384 to as bad as over 900. The initial setting of w seems to be the only random paramter here and seems to lead to the variation in different errors. To improve the algorithm, one could randomly select the data again before each pass.

e)

We now do a Poisson Regression. To run this in my program, set option to poisson, which will then call the usePoisson(). Inside of this function, one can set usepca to True or False, depending on wether one would like to use principal component analysis to select some number of features or not. In either case, the dataset is sent to the splitdataset() again to get back a training set of 4500 samples and a test set of 500 samples. We use 10 features, but once can set to use more. As poisson regression does not have a closed form solution, one must utilize a gradient descent method. This requires initializing an intial w, which we do as before, as normal random variables very close to zero, again to keep the error from being to large in the first few passes which could lead to NaNs. As we are using an exponential transfer, large values in the data lead to math overflow errors, so we rescale all the data by a factor of 1/10000. Note, as we are using a L_2 error, we can simple rescale the corresponding error by 10000 so that is comparable to our previous errors. Also, since it is necessary to take the inverse of the Hessian, we introduce a very small regularizer to avoid invertibility issues.

We get two very different answers depending on whether or not we use PCA. Without PCA, the errors are generally very large and require many iterations, into the 100s, which is very time consuming. Initially, the error can slowly rise but after some number of iterations, will begin to drop. The entire process is long. As stated in the assignment, poisson regression is very sensitive to linearly dependent features so this is a possible reason for the issues.

Using PCA, however, we get great results. Recall, the principal components are linear combinations of the original features which themselves are orthogonal, so Poisson Regression should perform much better. Indeed, within the first few passes we get an error less than 1.