Aditya N. Prasad
adityaprasad@college.harvard.edu
CS281-F17

Assignment #1 v 1.0

Due: 11:59pm September 22, 2017

Collaborators: None

Problem 1 (A Classic on the Gaussian Algebra, 10pts)

Let X and Y be independent univariate Gaussian random variables. In the previous problem set, you likely used the closure property that $Z = X + Y$ is also a Gaussian random variable. Here you'll prove this fact.

(a) Suppose X and Y have mean 0 and variances σ_X^2 and σ_Y^2 respectively. Write the pdf of $X + Y$ as an integral.

(b) Evaluate the integral from the previous part to find a closed-form expression for the pdf of $X+Y$, then argue that this expression implies that $X + Y$ is also Gaussian with mean 0 and variance $\sigma_X^2 + \sigma_Y^2$. Hint: what is the integral, over the entire real line, of

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right),$$

i.e., the pdf of a univariate Gaussian random variable?

(c) Extend the above result to the case in which X and Y may have arbitrary means.

(d) Univariate Gaussians are supported on the entire real line. Sometimes this is undesirable because we are modeling a quantity with positive support. A common way to transform a Gaussian to solve this problem is to exponentiate it. Suppose X is a univariate Gaussian with mean μ and variance σ^2. What is the pdf of e^X?

(a) We write a simple proof that the probability sum of independent rvs corresponds to their convolution for discrete variables. This corresponds directly to the continuous case with PMFs replaced by PDFs and sums replaced by integrals. Let X and Y be independent random variables and $Z = X+Y$. Then,

$$\begin{aligned} P(Z = k) &= P(X + Y = k) \\ &= \sum_x P(X+Y=k, X=x) \\ &= \sum_x P(Y = k - x \,|\, X = x) P(X = x) \\ &= \sum_x P(Y = k-x) P(X = x) \end{aligned}$$

In our case, we have: $X \sim \mathcal{N}(0, \sigma_x^2)$ and $Y \sim \mathcal{N}(0, \sigma_y^2)$. The pdf of the sum, denoted Z, is simply the convolution:

$$\begin{aligned} f_Z(z) &= f_{X+Y}(z) \\ &= \int_{-\infty}^{\infty} f_{Y|X}(z - x|x) f_X(x) dx \\ &= \int_{-\infty}^{\infty} f_Y(z-x) f_X(x) dx \\ &= \left(\frac{1}{\sigma_x\sqrt{2\pi}}\right)\left(\frac{1}{\sigma_y\sqrt{2\pi}}\right)\int_{-\infty}^{\infty} e^{-\frac{1}{2\sigma_x^2}x^2} e^{-\frac{1}{2\sigma_y^2}(z-x)^2} dx \end{aligned}$$

(b) To evaluate this integral neatly, we complete separate the dependence on z and x, whilst completing the square on x so that its integral is Gaussian in form. Consider the negative of the exponent:

$$\begin{aligned}
\frac{x^2}{2\sigma_x^2} + \frac{(x-z)^2}{2\sigma_y^2} &= \frac{x^2\sigma_y^2 + (x-z)^2\sigma_x^2}{2\sigma_x^2\sigma_y^2} \\
&= \frac{x^2(\sigma_x^2+\sigma_y^2)}{2\sigma_x^2\sigma_y^2} + \frac{(z^2-2zx)(\sigma_x^2+\sigma_y^2)\sigma_x^2}{2\sigma_x^2\sigma_y^2(\sigma_x^2+\sigma_y^2)} \\
&= \frac{\left(\frac{z^2\sigma_x^4}{\sigma_x^2+\sigma_y^2}\right) - x\left(2z\sigma_x^2\right) + x^2(\sigma_x^2+\sigma_y^2)}{2\sigma_x^2\sigma_y^2} + \frac{z^2}{2(\sigma_x^2+\sigma_y^2)} \\
&= \frac{\left[x - \left(\frac{z\sigma_x^2}{\sigma_x^2+\sigma_y^2}\right)\right]^2}{2\left(\frac{\sigma_x^2\sigma_y^2}{\sigma_x^2+\sigma_y^2}\right)} + \frac{z^2}{2(\sigma_x^2+\sigma_y^2)} \\
&= \frac{1}{2}\left(\frac{x-\mu^2}{\sigma'}\right)^2 + \frac{z^2}{2(\sigma_x^2+\sigma_y^2)}
\end{aligned}$$

Therefore, we have:

$$\begin{aligned}
f_Z(z) &= \left(\frac{1}{\sigma_x\sqrt{2\pi}}\right)\left(\frac{1}{\sigma_y\sqrt{2\pi}}\right)\int_{-\infty}^{\infty} \exp\left(-\frac{1}{2\sigma_x^2}x^2 + -\frac{1}{2\sigma_y^2}(z-x)^2\right)dx \\
&= \left(\frac{1}{\sigma_x\sqrt{2\pi}}\right)\left(\frac{1}{\sigma_y\sqrt{2\pi}}\right)\int_{-\infty}^{\infty} \exp\left(-\frac{1}{2}\left(\frac{x-\mu^2}{\sigma'}\right)^2 - \frac{z^2}{2(\sigma_x^2+\sigma_y^2)}\right)dx \\
&= \left(-\frac{1}{\sigma_x\sigma_y 2\pi}\right)\exp\left(-\frac{z^2}{2(\sigma_x^2+\sigma_y^2)}\right)\int_{-\infty}^{\infty} \exp\frac{1}{2}\left(\frac{x-\mu^2}{\sigma'}\right)^2 dx \\
&= \left(\frac{1}{\sigma_x\sigma_y 2\pi}\right)\exp\left(\frac{-z^2}{2(\sigma_x^2+\sigma_y^2)}\right)\sqrt{2\pi}\sigma' \\
&= \frac{1}{\sqrt{2\pi(\sigma_x^2+\sigma_y^2)}}\exp\left(-\frac{z^2}{2(\sigma_x^2+\sigma_y^2)}\right)
\end{aligned}$$

Therefore, $Z \sim \mathcal{N}(0, \sigma_x^2+\sigma_y^2)$.

(c) This is simply corresponds to a translation of the random-variable. More concretely, assume $X' \sim \mathcal{N}(\mu_x, \sigma_x^2)$ and $Y' \sim \mathcal{N}(\mu_y, \sigma_y^2)$. Then, $Z' = X' + Y'$ is :

$$\begin{aligned}
Z' &= X' + Y' \\
&= (X+\mu_x) + (Y+\mu_y) && (\text{where } X \sim \mathcal{N}(0,\sigma_x^2), Y \sim \mathcal{N}(0,\sigma_y^2)) \\
&= (X+Y) + (\mu_x+\mu_y) \\
&= Z + \mu'
\end{aligned}$$

Therefore, $Z \sim \mathcal{N}(\mu_x+\mu_y, \sigma_x^2+\sigma_y^2)$.

(d) We write this in terms of CDFs. Let $Y = \exp(X)$. We note that the exponential function is monotonic, and that $y \in [0, \infty$.

$$F_Y(y) = P(Y \le y) = P(g(X) \le y) = P(X \le \log y) = \Phi(\log y)$$

Here, Φ is the CDF of the univariate normal. The PDF is therefore:

$$f_Y(y) = \partial y F_Y(y) = \psi(\log y)\frac{1}{y}, y > 0$$

Problem 2 (Regression, 13pts)

Suppose that $X \in \mathbb{R}^{n \times m}$ with $n \geq m$ and $Y \in \mathbb{R}^n$, and that $Y \sim \mathcal{N}(Xw, \sigma^2 I)$. You learned in class that the maximum likelihood estimate $\hat{w}$ of w is given by

$$\hat{w} = (X^T X)^{-1} X^T Y$$

(a) Why do we need to assume that $n \geq m$?

(b) Define $H = X(X^T X)^{-1} X^T$, so that the "fitted" values $\hat{Y} = X\hat{w}$ satisfy $\hat{Y} = HY$. Show that H is an orthogonal projection matrix that projects onto the column space of X, so that the fitted y-values are a projection of Y onto the column space of X.

(c) What are the expectation and covariance matrix of $\hat{w}$?

(d) Compute the gradient with respect to w of the log likelihood implied by the model above, assuming we have observed Y and X.

(e) Suppose we place a normal prior on w. That is, we assume that $w \sim \mathcal{N}(0, \tau^2 I)$. Show that the MAP estimate of w given Y in this context is

$$\hat{w}_{MAP} = (X^T X + \lambda I)^{-1} X^T Y$$

where $\lambda = \sigma^2/\tau^2$. (You may employ standard conjugacy results about Gaussians without proof in your solution.)

[Estimating w in this way is called *ridge regression* because the matrix λI looks like a "ridge". Ridge regression is a common form of *regularization* that is used to avoid the overfitting (resp. underdetermination) that happens when the sample size is close to (resp. higher than) the output dimension in linear regression.]

(f) Do we need $n \geq m$ to do ridge regression? Why or why not?

(g) Show that ridge regression is equivalent to adding m additional rows to X where the j-th additional row has its j-th entry equal to $\sqrt{\lambda}$ and all other entries equal to zero, adding m corresponding additional entries to Y that are all 0, and then computing the maximum likelihood estimate of w using the modified X and Y.

For ease of notation, matrices are denoted with boldface uppercase letters, and vectors are denoted with boldface lowercase letters.

(a) We first make a purely mathematical argument, and then attempt to provide a reasonable interpretation. Consider the matrix: $\boldsymbol{X} \in \mathbb{R}^{m \times n}$. For argument's sake, assume $n < m$. Since the row rank equals column rank, the rank of matrix $\boldsymbol{X}$, denoted here by $\rho(\boldsymbol{X})$, is at most n. That is,

$$\rho(\boldsymbol{X}) = r \leq n < m$$

We also have that $\rho(\boldsymbol{X}) = \rho(\boldsymbol{X^T X})$. This follows directly from the fact that the null-space, $\mathcal{N}$, of $\boldsymbol{A}$ is the same as that of $\boldsymbol{A^T}$ (we provide a full-proof at the end of the problem). Therefore, we have that:

$$\rho(\boldsymbol{X^T X}) = r \leq n < m (\equiv \text{no. of columns in } \boldsymbol{X^T X})$$

As such, $\boldsymbol{X^T X}$ does not have full-rank in columns. With the regression problem as defined above, we know that the condition that maximizes the likelihood function is:

$$\boldsymbol{X^T X w} = \boldsymbol{X^T y}$$

This can be written as:

$$\boldsymbol{A}\boldsymbol{w} = \boldsymbol{b}$$

However, since $\rho(\boldsymbol{A}) = \rho(\boldsymbol{X^T X}) = n < m$, there are an infinite number of solutions. Hence, the problem is ill-posed.

There are many ways to interpret this:

(i) Gelman et al. (pg 377) make the argument that in such a regression problem, we cannot uniquely determine parameters because our columns of explanatory variables are not linearly independent. The data is 'collinear', or rather, 'co-planar'. The data forms a n dimensional subspace/hyperplane of the m dimensional parameter space. The data thus provides very little information about possible linear combinations of $\boldsymbol{w}$.

(ii) This is similar to the proof above, but perhaps is easier to understand. $\boldsymbol{X^T X w} = \boldsymbol{X^T y}$ are the normal form equations for the system $\boldsymbol{Xw} = \boldsymbol{y}$. Here, $\boldsymbol{w} \in \mathbb{R}^m$. Therefore, we have a system of equations that looks like:

$$\begin{pmatrix} x_1^1 w_1 + \cdots + x_m^1 w_m \\ \vdots \\ x_1^n w_1 + \cdots + x_m^n w_m \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

Therefore, in the best case scenario we have n equations for m unknowns ($\boldsymbol{w}$), where $n < m$. Therefore, we see that we have an infinite number of solutions.

(iii) (Geometric Intuition based on projections)

Geometric argument

(b) The easiest way to show this is to prove that H is idempotent and symmetric, and then show that $\boldsymbol{X}^T(\boldsymbol{y} - \hat{\boldsymbol{y}}) = 0$. We drop the bold-face for ease of typing here.

(i) Idempotence:

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

(ii) Symmetry (noting X^TX is obviously symmetric):

$$H^T = (X(X^TX)^{-1}X^T)^T = H$$

Therefore, H is an orthogonal, projection operator.

(iii) Orthogonality: consider $v = y - Hy$. We show that this vector is perpendicular to the column-space of X.

$$\begin{aligned} X^T(y - Hy) &= X^Ty - X^T(X(X^TX)^{-1}X^T)y \\ &= X^Ty - X^Ty = 0 \end{aligned}$$

Therefore, all the columns of X are perpendicular to $(y - Hy)$. Therefore, Hy is the projection of y onto the column-space of X.

There is a more sophisticated and general way of making the last point. This has been adapted from *Bhattacharya & Roy, Linear Algebra and Matrix Analysis for Statistics.*

Theorem

If $\boldsymbol{y} \in \mathbb{R}^n$ and $\boldsymbol{X} \in \mathbb{R}^{n \times m}$, the orthogonal projection of y onto the column-space of $\boldsymbol{X}$, denoted $\boldsymbol{w} \in \mathbb{R}^m$ satisfies the normal equations:

$$\boldsymbol{X^T X w} = \boldsymbol{X^T y}$$

The maximum-likelihood estimate of the weights is thus a distribution (because of it's dependence on Y) even though the weights themselves are simply parameters.

Proof. We know that $\mathbb{R}^n$ can be written as the direct sum: $\mathbb{R}^n = \mathcal{C}(X) \oplus \mathcal{C}(X)^\perp = \mathcal{C}(X) \oplus \mathcal{N}(X^T)$. Therefore, we can write $\boldsymbol{y} \in \mathbb{R}^n$ as:

$$y = u + v, \left\{u \in \mathcal{C}(X), v \in \mathcal{N}(X^T)\right\}$$

Since $u \in \mathcal{C}(X)$, there exists $w \in \mathbb{R}^m$ s.t. $u = Xw$. Therefore, we have:

$$v = y - u = y - Xw$$

Since $v \in \mathcal{N}(X^T)$,

$$X^T v = X^T(y - Xw) = 0$$

Therefore,

$$X^T X w = X^T y \tag{1}$$

□

Further, if $X^T X$ has full-rank, it has an inverse (because it is also square). Therefore, we have:

$$\boldsymbol{w} = \boldsymbol{(X^T X)}^{-1}\boldsymbol{X^T y} \tag{2}$$

(c) We are assuming here that $\boldsymbol{X}$ and $\boldsymbol{w}$ are parameters. Therefore, we have:

$$\begin{aligned}
\mathbb{E}(\hat{\boldsymbol{w}}) &= \mathbb{E}(\boldsymbol{(X^T X)}^{-1}\boldsymbol{X^T y}) \\
&= \boldsymbol{(X^T X)}^{-1}\boldsymbol{X^T}\mathbb{E}(\boldsymbol{y}) \\
&= \boldsymbol{(X^T X)}^{-1}\boldsymbol{X^T X w} \\
&= \boldsymbol{w}
\end{aligned}$$

The covariance matrix is similarly calculated:

$$\begin{aligned}
\operatorname{cov}(\hat{\boldsymbol{w}}, \hat{\boldsymbol{w}}) &= \mathbb{E}\left[(\hat{\boldsymbol{w}} - E(\hat{\boldsymbol{w}}))(\hat{\boldsymbol{w}} - E(\hat{\boldsymbol{w}}))^T\right] &(3)\\
&= \mathbb{E}\left[\hat{\boldsymbol{w}}\hat{\boldsymbol{w}}^T\right] - \mathbb{E}\left[\hat{\boldsymbol{w}}\right]\mathbb{E}\left[\hat{\boldsymbol{w}}^T\right] &(4)
\end{aligned}$$

We have:

$$\begin{aligned}
\mathbb{E}(\hat{\boldsymbol{w}}\hat{\boldsymbol{w}}^T) &= \mathbb{E}\left(\boldsymbol{(X^T X)}^{-1}\boldsymbol{X^T y}\left[\boldsymbol{(X^T X)}^{-1}\boldsymbol{X^T y}\right]^T\right) \\
&= \boldsymbol{(X^T X)}^{-1}\boldsymbol{X^T X}\boldsymbol{(X^T X)}^{-1}\mathbb{E}(\boldsymbol{y y^T}) \\
&= \boldsymbol{(X^T X)}^{-1}\mathbb{E}(\boldsymbol{y y^T})
\end{aligned}$$

Therefore, we have:

$$\begin{aligned}
\operatorname{cov}(\hat{\boldsymbol{w}}, \hat{\boldsymbol{w}}) &= \boldsymbol{(X^T X)}^{-1}\left(\mathbb{E}(\boldsymbol{y y^T}) - \mathbb{E}(\boldsymbol{y})\mathbb{E}(\boldsymbol{y}^T)\right) &(5)\\
&= \boldsymbol{(X^T X)}^{-1}(\sigma^2 \boldsymbol{I}) &(6)
\end{aligned}$$

(d) We have:

$$p(\boldsymbol{y}; \boldsymbol{X}, \boldsymbol{w}) \propto \exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})\right]$$

Therefore, if we want to maximize the logarithm of this density w.r.t the parameter $\boldsymbol{w}$, we have:

$$\arg\max_{\boldsymbol{w}} \left[-(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})\right]$$

Considering this is a convex function, we can simply compute the derivative/gradient w.r.t $\boldsymbol{w}$ and set this to zero. Expanding and collecting terms that have a $\boldsymbol{w}$, we have:

$$\begin{aligned} f(\boldsymbol{w}) &= \boldsymbol{w^T X^T X w} - (\boldsymbol{w}T\boldsymbol{X}\boldsymbol{y} + \boldsymbol{y}^T\boldsymbol{X}\boldsymbol{w}) \\ &= \boldsymbol{w^T X^T X w} - 2\boldsymbol{w^T X^T y} \end{aligned}$$

The gradient (using matrix identities) is simply:

$$\partial_{\boldsymbol{w}} = 2\boldsymbol{X^T X w} - 2\boldsymbol{X^T y} \tag{7}$$

(e) Consider the log-posterior:

$$\ln p(\boldsymbol{w} \mid \boldsymbol{y}; \boldsymbol{X}) = \ln p(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{X}) + \ln p(\boldsymbol{w})$$

Given that we are going to optimize w.r.t $\boldsymbol{w}$, we only need worry about terms that have a $\boldsymbol{w}$ dependence. We have:

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{0}, \tau^2\boldsymbol{I})$$

Therefore,

$$\ln p(\boldsymbol{w}) = c - \frac{1}{2\tau^2}\boldsymbol{w^T w}$$

From above, we have:

$$\ln p(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{X}) = K - \frac{1}{2\sigma^2}\left[\boldsymbol{w^T X^T X w} - 2\boldsymbol{w^T X^T y}\right]$$

Computing the gradient with respect to the log-posterior and setting it to zero, we have:

$$\begin{aligned} \boldsymbol{X^T X w} + \frac{\sigma^2}{\tau^2}\boldsymbol{w} &= \boldsymbol{X^T y} \\ \implies \boldsymbol{w} &= (\boldsymbol{X^T X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X^T y} \end{aligned}$$

As shown above, taking the inverse is a valid-operation because it always exists.

(f) We do not need $n \geq m$ to perform ridge regression. First, we note that any matrix, $X^T X$ is positive semi-definite.

Proof. Consider the following inner product for some vector, v:

$$\langle v, X^T X v\rangle = \langle Xv, Xv\rangle \geq 0$$

□

The last inequality follows from the fact that we have assumed an inner-product (which is positive definite) on the space; however $v \neq 0$ can still be part of the null-space of X, which is why we have positive semi-definiteness. Now consider $X^T X + \lambda I, \lambda > 0$. This is positive definite:

Proof. Consider the following inner product for some vector, v:

$$\begin{aligned}\langle v, (X^T X + \lambda I)v\rangle &= \langle v, X^T X v\rangle + \lambda\langle v, v\rangle \\ &= \langle Xv, Xv\rangle + \lambda||v^2|| > 0\end{aligned}$$

□

Therefore, we have that $Z = X^T X + \lambda I$ is a positive-definite symmetric matrix. It is therefore always invertible; we can see this by either using the spectral theorem, or noting that it has a zero nullspace, and so full (column=row square) rank. As such, a unique w_{MAP} always exists; in other words, the linear regression problem is well-posed (even if it might be susceptible to generalization errors if $n \sim m$).

(g) Consider $\boldsymbol{X'} = \begin{pmatrix} \boldsymbol{X} \\ \sqrt{\lambda}\boldsymbol{I}_m \end{pmatrix}$ and $\boldsymbol{y'} = \begin{pmatrix} \boldsymbol{y} \\ \boldsymbol{O}_m \end{pmatrix}$

The required quadratic form is therefore:

$$\begin{aligned}\mathcal{L}(\boldsymbol{w}) &\propto -\frac{1}{2}(\boldsymbol{y'} - \boldsymbol{X'w})^T \frac{1}{\sigma^2}\boldsymbol{I}(\boldsymbol{y'} - \boldsymbol{X'w}) \\ &= -\frac{1}{2\sigma^2}(\boldsymbol{y'} - \boldsymbol{X'w})^T(\boldsymbol{y'} - \boldsymbol{X'w})\end{aligned}$$

This is the same form of the likelihood that we had in part (b), and therefore, we have:

$$\begin{aligned}\hat{\boldsymbol{w}} &= (\boldsymbol{X'^T X'})^{-1}\boldsymbol{X'^T Y'} \\ &= \left\{\begin{pmatrix}\boldsymbol{X^T} & \sqrt{\lambda}\boldsymbol{I}\end{pmatrix}\begin{pmatrix}\boldsymbol{X} \\ \sqrt{\lambda}\boldsymbol{I}\end{pmatrix}\right\}^{-1}\begin{pmatrix}\boldsymbol{X^T} & \sqrt{\lambda}\boldsymbol{I}\end{pmatrix}\begin{pmatrix}\boldsymbol{y} \\ \boldsymbol{O}_m\end{pmatrix} \\ &= (\boldsymbol{X^T X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X^T y}\end{aligned}$$

The interpretation of this method is slightly different from the projection interpretation we had earlier. Instead of simply defining the weights with a prior that makes sure the regression-problem is well-posed, we are physically adding m pieces of 'pseudo-data' to our data-matrix to make sure that each feature has at-least one data point that uses it. Further, the column-space spans $\mathbb{R}^m$.

Problem 3 (The Dirichlet and Multinomial Distributions, 12pts)

The Dirichlet distribution over K categories is a generalization of the beta distribution. It has a shape parameter $\alpha \in \mathbb{R}^K$ with non-negative entries and is supported over the set of K-dimensional positive vectors whose components sum to 1. Its density is given by

$$f(\theta_{1:K}|\alpha_{1:K}) = \frac{\Gamma\left(\sum_k \alpha_k\right)}{\prod_k \Gamma(\alpha_k)} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1}$$

(Notice that when $K = 2$, this reduces to the density of a beta distribution.) For the rest of this problem, assume a fixed $K \geq 2$.

(a) Suppose θ is Dirichlet-distributed with shape parameter α. Without proof, state the value of $E(\theta)$. Your answer should be a vector defined in terms of either α or K or potentially both.

(b) Suppose that $\theta \sim \text{Dir}(\alpha)$ and that $X \sim \text{Cat}(\theta)$, where Cat is a Categorical distribution. That is, suppose we first sample a K-dimensional vector θ with entries in $(0, 1)$ from a Dirichlet distribution and then roll a K-sided die such that the probability of rolling the number k is θ_k. Prove that the posterior $p(\theta|X)$ also follows a Dirichlet distribution. What is its shape parameter?

(c) Now suppose that $\theta \sim \text{Dir}(\alpha)$ and that $X^{(1)}, X^{(2)}, \ldots \overset{iid}{\sim} \text{Cat}(\theta)$. Show that the posterior predictive after $n-1$ observations is given by,

$$P(X^{(n)} = k|X^{(1)}, \ldots, X^{(n-1)}) = \frac{\alpha_k^{(n)}}{\sum_k \alpha_k^{(n)}}$$

where for all k, $\alpha_k^{(n)} = \alpha_k + \sum_{i=1}^{n-1} \mathbf{1}\{X^{(i)} = k\}$. (Bonus points if your solution does not involve any integrals.)

(d) Consider the random vector $Z_k = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{X^{(i)} = k\}$ for all k. What is the mean of this vector? What is the distribution of the vector? (If you're not sure how to rigorously talk about convergence of random variables, give an informal argument. Hint: what would you say if θ were fixed?) What is the marginal distribution of a single class $p(Z_k)$?

(e) Suppose we have K distinct colors and an urn with α_k balls of color k. At each time step, we choose a ball uniformly at random from the urn and then add into the urn an additional new ball of the same color as the chosen ball. (So if at the first time step we choose a ball of color 1, we'll end up with $\alpha_1 + 1$ balls of color 1 and α_k balls of color k for all $k > 1$ at the start of the second time step.) Let $\rho_k^{(n)}$ be the fraction of all the balls that are of color k at time n. What is the distribution of $\lim_{n\to\infty} \rho_k^{(n)}$? Prove your answer.

(a) As mentioned in the problem, the Dirichlet distribution is the multidimensional generalization of the Beta distribution. The pdf is given by:

$$f(\boldsymbol{\theta}; \boldsymbol{\alpha}) \propto \prod_{k=1}^{K} \theta_k^{\alpha_k - 1} \mathbb{I}\left[\theta_k \in S_K\right] \tag{8}$$

From Murphy we have that:

$$\mathbb{E}\left[\theta_k\right] = \frac{\alpha_k}{\sum_{i=1}^{K} \alpha_i}$$

Therefore, we simply have:

$$\mathbb{E}\left[\boldsymbol{\theta}\right] = \frac{1}{C}\boldsymbol{\alpha} \tag{9}$$

where,

$$C = \sum_{i=1}^{K} \alpha_i \tag{10}$$

(b) We are given:

$$\begin{aligned}\boldsymbol{\theta} &\sim \mathrm{Dir}(\boldsymbol{\alpha})\\ \boldsymbol{X} \,|\, \boldsymbol{\theta} &\sim \mathrm{Cat}(\boldsymbol{\theta})\end{aligned}$$

We are asked for:

$$P(\boldsymbol{\theta} \,|\, \boldsymbol{X}) = \frac{P(\boldsymbol{X} \,|\, \boldsymbol{\theta})P(\boldsymbol{\theta})}{P(\boldsymbol{X})}$$

More clearly, because the random variables are continuous, we write clearly write this in terms of the pdfs (f) instead of implying that $(P \equiv f)$ Therefore, we have:

$$f_{\boldsymbol{\Theta} \,|\, \boldsymbol{X}}(\boldsymbol{\theta} \,|\, \boldsymbol{x}) = \frac{P_{\boldsymbol{X} \,|\, \boldsymbol{\Theta}}(\boldsymbol{x} \,|\, \boldsymbol{\theta}) f_{\boldsymbol{\Theta}}(\boldsymbol{\theta})}{f_{\boldsymbol{X}}(\boldsymbol{x})}$$

However, we have:

$$\int f_{\boldsymbol{\Theta} \,|\, \boldsymbol{X}}(\boldsymbol{\theta} \,|\, \boldsymbol{x}) d\boldsymbol{\theta} = 1 = \frac{P_{\boldsymbol{X} \,|\, \boldsymbol{\Theta}}(\boldsymbol{x} \,|\, \boldsymbol{\theta}) f_{\boldsymbol{\Theta}}(\boldsymbol{\theta})}{f_{\boldsymbol{X}}(\boldsymbol{x})}$$

Thus,

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \int P_{\boldsymbol{X} \,|\, \boldsymbol{\Theta}}(\boldsymbol{x} \,|\, \boldsymbol{\theta}) f_{\boldsymbol{\Theta}} d\boldsymbol{\theta}$$

We were able to do this because the terms on the r.h.s are a function of $\boldsymbol{\theta}$ and the terms on the l.h.s are function of $\boldsymbol{x}$. In other words, $f_{\boldsymbol{X}}(\boldsymbol{x})$ is simply a normalization constant for the integral on the right. We can therefore omit it, and write:

$$f_{\boldsymbol{\Theta} \,|\, \boldsymbol{X}}(\boldsymbol{\theta} \,|\, \boldsymbol{x}) \propto f_{\boldsymbol{X} \,|\, \boldsymbol{\Theta}}(\boldsymbol{x} \,|\, \boldsymbol{\theta}) f_{\boldsymbol{\Theta}}(\boldsymbol{\theta}) \tag{11}$$

Writing this out (and using the fact that we have a one-hot encoded vector), we see:

$$\begin{aligned}f_{\boldsymbol{\Theta} \,|\, \boldsymbol{X}}(\boldsymbol{\theta} \,|\, \boldsymbol{x}) &\propto \left(\prod_{i=1}^{K} \theta_i^{\mathbb{I}\{x_i=1\}}\right)\left(\prod_{j=1}^{K} \theta_j^{\alpha_j - 1}\right)\\ &= \prod_{k=1}^{K} \theta_k^{(x_k + \alpha_k - 1)}\end{aligned}$$

Compare this to the form of the Dirichlet distribution. Thus, we have that the posterior is simply Dirichlet with updated shape parameters: $\alpha_k^{'} = x_k + \alpha_k - 1$.

(c) First we write out what this would like if every distribution were discrete. This translates easily to the continuous case but is easy to reason about. We have: 3c bonus pts

$$
\begin{aligned}
P(A \mid B) &= \sum_{x \in X} P(A, X = x \mid B) \\
&= \sum_{x \in X} P(A \mid X = x, B) P(X = x \mid B)
\end{aligned}
$$

For our problem , we write this as:

$$
P(\boldsymbol{X}_k^{(n)} = 1 \mid \boldsymbol{X}^{(1)}, \cdots, \boldsymbol{X}^{(n)}) \propto P(X_k^n = 1 \mid \boldsymbol{X}^1, ..., \boldsymbol{X}^n, \boldsymbol{\theta}) f(\boldsymbol{\theta} \mid \boldsymbol{X}^{(1)} \cdots \boldsymbol{X}^{(n)}) \tag{12}
$$

First, we note that the first term on the right hand side reduces to:

$$
\begin{aligned}
P(X_k^n = 1 \mid \boldsymbol{X}^1, ..., \boldsymbol{X}^n, \boldsymbol{\theta}) &= P(X_k^n = 1 \mid \boldsymbol{\theta}) \\
&= P(X_k^n = 1 \mid \theta_k)
\end{aligned}
$$

The first equality comes from the independence assumption (given θ). The second equality comes from the structure of the multinoulli/categorical distribution. The probability of the kth event is given by θ_k. Therefore, our conditions reduce to:

$$
P(X_k^n = 1 \mid \boldsymbol{X}^1, ..., \boldsymbol{X}^n, \boldsymbol{\theta}) f(\boldsymbol{\theta} \mid \boldsymbol{X}^{(1)} \cdots \boldsymbol{X}^{(n)}) = P(X_k^n = 1 \mid \theta_k) f(\theta_k \mid \boldsymbol{X}_k^{(1)} \cdots \boldsymbol{X}_k^{(n)})
$$

If we wanted to be more explicit in this derivation, we could write this is as:

$$
\begin{aligned}
P(\boldsymbol{X}_k^{(n)} = 1 \mid \boldsymbol{X}^{(1)}, \cdots, \boldsymbol{X}^{(n-1)}) &= \int_{Supp(\boldsymbol{\theta})} P(X_k^n = 1 \mid \boldsymbol{X}^1, ..., \boldsymbol{X}^{n-1}, \theta_k) f(\boldsymbol{\theta} \mid \boldsymbol{X}_k^{(1)} \cdots \boldsymbol{X}_k^{(n-1)}) d\theta_1 \cdots d\theta_K \\
&= \int_{Supp(\theta_k)} P(X_k^n = 1 \mid \boldsymbol{X}^1, ..., \boldsymbol{X}^{n-1}, \theta_k) f(\theta_k \mid \boldsymbol{X}_k^{(1)} \cdots \boldsymbol{X}_k^{(n-1)}) d\theta_k \\
&= \int_{Supp(\theta_k)} P(X_k^n = 1 \mid \theta_k) P(X_k^{(1)} \mid \theta_k) \cdots P(X_k^{(n-1)} \mid \theta_k) f(\theta_k) d\theta_k \\
&= \int_{Supp(\theta_k)} \theta_k \left[\prod_i \theta_k^{I\{X_k^{(i)}=1\}} (1-\theta_k)^{I\{X_k^{(i)}=0\}} \right] \underbrace{\theta_k^{\alpha_k - 1} (1-\theta_k)^{\sum_{j \neq k} \alpha_j - 1}}_{f(\theta_k) \text{ reduces to a beta distribution}} d\theta_k
\end{aligned}
$$

The integral above is essentially an expectation of a beta distribution:

$$
\begin{aligned}
P(\boldsymbol{X}_k^{(n)} = 1 \mid \boldsymbol{X}^{(1)}, \cdots, \boldsymbol{X}^{(n-1)}) &= \int_{Supp(\theta_k)} \theta_k \theta_k^{\alpha_k^{(n)} - 1} (1-\theta_k)^{\sum_{j \neq k} \alpha_j^{(n)} - 1} d\theta_k \\
&= \mathbb{E}_{\theta_k \sim \text{Beta}\left(\alpha_k^{(n)}, \sum_{j \neq k} \alpha_j^{(n)}\right)} [\theta_k] \\
&= \frac{\alpha_k^{(n)}}{\sum_{j=1}^{n-1} \alpha_j^{(n)}}
\end{aligned}
$$

Here, from the previous equations, $\alpha_k^{(n)} = \alpha_k + \sum_{i=1}^{n-1} I\{X_k^i = 1\}$.

Here, if we want to work quickly (to get the bonus points), we note from above that $P(\theta \mid \boldsymbol{X})$ is Dirichlet with an updated shape parameter. Intuitively, with what we've learnt from above, one would imagine that we just need to update the pseudo counts for the θ_k.

(d) We begin as follows:

$$\begin{aligned}
\mathbb{E}(Z_k) &= \lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}(I\{X^{(i)} = k\}) \\
&= \lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} P(X^{(i)} = k) \\
&= \lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} \int P(X^{(i)} = k \mid \theta_k) f_{\Theta_k}(\theta_k) d\theta_k \\
&= \lim_{n\to\infty} \frac{1}{n}\int \sum_{i=1}^{n} P(X^{(i)} = k \mid \theta_k) f_{\Theta_k}(\theta_k) d\theta_k \\
&= \lim_{n\to\infty} \frac{1}{n}\int \sum_{i=1}^{n} \theta_k f_{\Theta_k}(\theta_k) d\theta_k \\
&= \lim_{n\to\infty} \frac{1}{n} n \int \theta_k f_{\Theta_k}(\theta_k) d\theta_k \\
&= \mathbb{E}(\theta_k) \\
&= \frac{\alpha_k}{\sum_{j=1}^{K} \alpha_j}
\end{aligned}$$

This is initially somewhat un-intuitive: it states that in the large data-set limit, the expectation of the sample mean is simply the expectation of the prior. But, the key seems to be in realizing that our model has no updates here. Our decisions are based solely on the prior, and the conditional independence of samples. If we compute this for the entire random vector, it is simply the same as part (a):

$$\mathbb{E}[\boldsymbol{\theta}] = \frac{1}{C}\boldsymbol{\alpha}$$

where,

$$C = \sum_{i=1}^{K} \alpha_i$$

Thus, we can perhaps assume that $Z_k \sim$ Beta, and $\boldsymbol{Z} \sim$ Dirichlet. Further, if we assume that $I\{X^{(i)} = k\}$ has finite-variance - which it does because it's simply an indicator rvs - we have:

$$\begin{aligned}
\text{var}(Z_k) &= \lim_{n\to\infty} \frac{1}{n^2}\sum_{i=1}^{n} \text{var}(I\{X^{(i)} = k\}) \\
&= \lim_{n\to\infty} \frac{1}{n}\sigma^2
\end{aligned}$$

We can see directly that as $n \to \infty$, $\text{var}(Z_k) \to 0$. Thus, the marginal distribution tends to a dirac/experimental distribution centered at the mean of the prior. We can formalize by a simple application of the Chebyshev inequality:

$$P(|Z_k - \mu| > \epsilon) \leq \frac{\sigma^2}{n\epsilon}$$

For any $\epsilon > 0$, therefore, as $n \to \infty$, $P(|Z_k - \mu| > \epsilon) \to 0$.

(e) We decide to tackle this probably in a fully Bayesian sense. In the frequentist sense, we have a probability distribution that changes with each draw; the draws then are not independent. This makes calculations tedious.

Instead, we attempt to frame the question in terms of the problem above. We begin by assuming that there exists an underlying stationary probability distribution, $\boldsymbol{\theta}$ that defines how we sample our draws. Given $\boldsymbol{\theta}$, each draw is independent and distributed as $\text{Cat}(\boldsymbol{\theta})$. Given that we don't know what $\boldsymbol{\theta}$, we assume a Dirichlet prior on it - herein lies our big assumption. But, we show that this consistent with our beliefs. Consider $P(X^n = k|X^{(1)} \cdots X^{(n-1)})$. From part (c), we know that this is:

$$\begin{aligned} P(X^n = k|X^{(1)} \cdots X^{(n-1)}) &= \mathbb{E}_{\theta_k \sim \text{Beta}\left(\alpha_k^{(n)}, \sum_{j \neq k} \alpha_j^{(n)}\right)} [\theta_k] \\ &= \frac{\alpha_k^{(n)}}{\sum_{j=1}^{n-1} \alpha_j^{(n)}} \end{aligned}$$

Here, from the previous equations, $\alpha_k^{(n)} = \alpha_k + \sum_{i=1}^{n-1} I\{X_k^i = 1\}$. This is consistent with the our frequentist probability. It is worth noting that our interpretation of probability here is that it is a measure of our own uncertainty. Our best initial guess for the problem is the pseudo-counts. Whenever we draw a new ball, we update our beliefs with the new counts, and then make the next prediction. Here, we have: $\rho_k^{(n)} = Z_k$. Therefore, as $n \to \infty$, we have:

$$\rho_k^{(n)} \to \text{Dirichlet}(\boldsymbol{\alpha})$$

Physicochemical Properties of Protein Tertiary Structure

In the following problems we will code two different approaches for solving linear regression problems and compare how they scale as a function of the dimensionality of the data. We will also investigate the effects of linear and non-linear features in the predictions made by linear models.

We will be working with the regression data set Protein Tertiary Structure: `https://archive.ics.uci.edu/ml/machine-learning-databases/00265/CASP.csv`. This data set contains information about predicted conformations for 45730 proteins. In the data, the target variable y is the root-mean-square deviation (RMSD) of the predicted conformations with respect to the true properly folded form of the protein. The RMSD is the measure of the average distance between the atoms (usually the backbone atoms) of superimposed proteins. The features $\mathbf{x}$ are physico-chemical properties of the proteins in their true folded form. After downloading the file CASP.csv we can load the data into python using

```
>>> import numpy as np
>>> data = np.loadtxt("CASP.csv", delimiter = ",", skiprows = 1)
```

We can then obtain the vector of target variables and the feature matrix using

```
>>> y = data[:, 0]
>>> X = data[:, 1:]
```

We can then split the original data into a training set with 90% of the data entries in the file CASP.csv and a test set with the remaining 10% of the entries. Normally, the splitting of the data is done at random, but here **we ask you to put into the training set the first 90% of the elements from the file CASP.csv** so that we can verify that the values that you will be reporting are correct. (This should not cause problems, because the rows of the file are in a random order.)

We then ask that you **normalize** the features so that they have zero mean and unit standard deviation in the training set. This is a standard step before the application of many machine learning methods. After these steps are done, we can concatenate a **bias feature** (one feature which always takes value 1) to the observations in the normalized training and test sets.

We are now ready to apply our machine learning methods to the normalized training set and evaluate their performance on the normalized test set. In the following problems, you will be asked to report some numbers and produce some figures. Include these numbers and figures in your assignment report. **The numbers should be reported with up to 8 decimals**.

Problem 4 (7pts)

Assume that the targets y are obtained as a function of the normalized features $\mathbf{x}$ according to a Bayesian linear model with additive Gaussian noise with variance $\sigma^2 = 1.0$ and a Gaussian prior on the regression coefficients $\mathbf{w}$ with *precision* matrix $\Sigma^{-1} = \tau^{-2}\mathbf{I}$ where $\tau^{-2} = 10$. Code a routine using the **QR decomposition** (see Section 7.5.2 in Murphy's book) that finds the Maximum a Posteriori (MAP) value $\hat{\mathbf{w}}$ for $\mathbf{w}$ given the normalized training data

- Report the value of $\hat{\mathbf{w}}$ obtained.
- Report the root mean squared error (RMSE) of $\hat{\mathbf{w}}$ in the normalized test set.

Note: all code has been submitted at the end.

(a)

$$w = \begin{pmatrix} [7.74153395] \\ [5.55782079] \\ [2.25190765] \\ [1.07880135] \\ [-5.91177796] \\ [-1.73480336] \\ [-1.63875478] \\ [-0.26610556] \\ [0.81781409] \\ [-0.65913397] \end{pmatrix}$$

(b) $RMSE$: 5.2098893712934

Problem 5 (14pts)

L-BFGS is an iterative method for solving general nonlinear optimization problems. For this problem you will use this method as a black box that returns the MAP solution by sequentially evaluating the objective function and its gradient for different input values. The goal of this problem is to use a built-in implementation of the L-BFGS algorithm to find a point estimate that maximizes our posterior of interest. Generally L-BFGS requires your black box to provide two values: the current objective and the gradient of the objective with respect to any parameters of interest. To use the optimizer, you need to first write two functions: (1) to compute the loss, or the *negative* log-posterior and (2) to compute the gradient of the loss with respect to the weights w.

As a preliminary to coming work in the class, we will use the L-BFGS implemented in PyTorch. [Warning: For this assignment we are using a small corner of the PyTorch world. Do not feel like you need to learn everything about this library.]

There are three parts to using this optimizer:

1. Create a vector of weights in NumPy, wrap in a pytorch `Tensor` and `Variable`, and pass to the optimizer.

   ```
   from torch import Tensor
   from torch.autograd import Variable

   # Construct a PyTorch variable array (called tensors).
   weights = Variable(Tensor(size))

   # Initialize an optimizer of the weights
   optimizer = torch.optim.LBFGS(weights)

   ...
   ```

2. Write a python function that uses the current weights to compute the log-posterior **and** sets weights.grad to be the gradient of the log-posterior with respect to the current weights.

   ```
   def black_box():
       # Access the value of the variable as a numpy array.
       weights_data = weights.data.numpy()

       ...

       # Set the gradient of the variable.
       weights.grad = Tensor({numpy})

       return {objective}
   ```

3. Repeatedly call `optimizer.step(black_box)` to optimize.

[If you are feeling adventurous, you might find it useful to venture into the land of autograd and check your computation with PyTorch's `torch.autograd.gradcheck.get_numerical_jacobian`.]

- After running for 100 iterations, report the value of $\hat{\mathbf{w}}$ obtained.
- Report the RMSE of the predictions made with $\hat{\mathbf{w}}$ in the normalized test set.

Note: all code has been submitted at the end.

(a) $\begin{pmatrix} [7.74153376] \\ [5.55781555] \\ [2.25191164] \\ [1.07879972] \\ [-5.9117775] \\ [-1.73480093] \\ [-1.63875604] \\ [-0.26610556] \\ [0.81781441] \\ [-0.65913391] \end{pmatrix}$

(b) 5.2098893822576393

Problem 6 (14pts)

Linear regression can be extended to model non-linear relationships by replacing the original features $\mathbf{x}$ with some non-linear functions of the original features $\boldsymbol{\phi}(\mathbf{x})$. We can automatically generate one such non-linear function by sampling a random weight vector $\mathbf{a} \sim \mathcal{N}(0, \mathbf{I})$ and a corresponding random bias $b \sim \mathrm{U}[0, 2\pi]$ and then making $\phi(\mathbf{x}) = \cos(\mathbf{a}^{\mathrm{T}}\mathbf{x} + b)$. By repeating this process d times we can generate d non-linear functions that, when applied to the original features, produce a non-linear mapping of the data into a new d dimensional space. We can encode these d functions into a matrix $\mathbf{A}$ with d rows, each one with the weights for each function, and a d-dimensional vector $\mathbf{b}$ with the biases for each function. The new mapped features are then obtained as $\boldsymbol{\phi}(\mathbf{x}) = \cos(\mathbf{A}\mathbf{x} + \mathbf{b})$, where cos applied to a vector returns another vector whose elements are the result of applying cos to the individual elements of the original vector.

Generate 4 sets of non-linear functions, each one with $d = 100, 200, 400, 600$ functions, respectively, and use them to map the features in the original normalized training and test sets into 4 new feature spaces, each one of dimensionality given by the value of d. After this, for each value of d, find the MAP solution $\hat{\mathbf{w}}$ for $\mathbf{w}$ using the corresponding new training set and the method from problem 4. Use the same values for σ^2 and τ^{-2} as before. You are also asked to record the time taken by the method QR to obtain a value for $\hat{\mathbf{w}}$. In python you can compute the time taken by a routine using the time package:

```
>>> import time
>>> time_start = time.time()
>>> routine_to_call()
>>> running_time = time.time() - time_start
```

Next, compute the RMSE of the resulting predictor in the normalized test set. Repeat this process with the method from problem 5 (L-BFGS).

- Report the test RMSE obtained by each method for each value of d.

You are asked to generate a plot with the results obtained by each method (QR and L-BFGS) for each value of d. In this plot the x axis should represent the time taken by each method to run and the y axis should be the RMSE of the resulting predictor in the normalized test set. The plot should contain 4 points in red, representing the results obtained by the method QR for each value of d, and 4 points in blue, representing the results obtained by the method L-BFGS for each value of d. Answer the following questions:

- Do the non-linear transformations help to reduce the prediction error? Why?
- What method (QR or L-BFGS) is faster? Why?
- (Extra Problem, Not Graded) Instead of using random $\mathbf{A}$, what if we treat $\mathbf{A}$ as another parameter for L-BFGS to optimize? You can do this by wrapping it as a variable and passing to the constructor. Compute its gradient as well in *black_box* either analytically or by using PyTorch *autograd*.

- We present just the average for 3 runs and reduce significant figures:

Table 1: RMSE Comparison for LBFGS and QR with dimensionality

	d=100	d=200	d=400	d=800
LBFGS	4.93773384	4.72245696	4.55136800	4.447570
QR	4.93773277	4.73752263	4.55136823	4.447611

The RMSE-time figure is below:

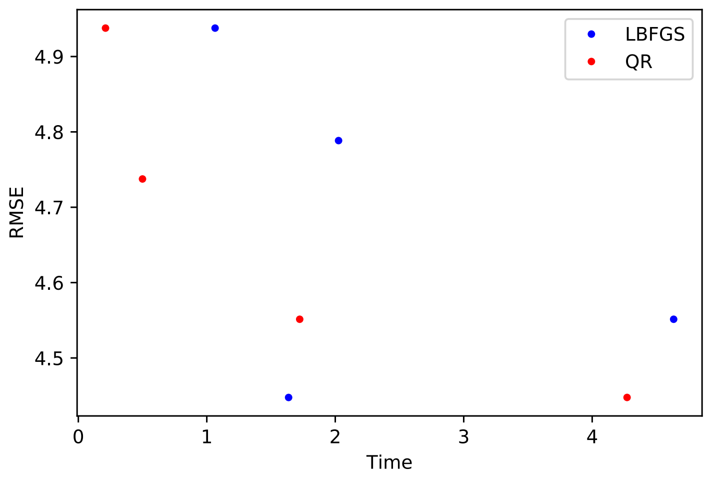

Figure 1: Figure 1: RMSE vs. Time for LBFGS and QR

- Non-linear transformations help reduce the prediction error presumably because the underlying data has non-linear relationships in feature-space. Further, non-linear transformations allow us to increase the dimensionality of feature-space; therefore, we are not only able to capture more intricate non-linear relationships but also able to project our data onto a higher-dimensional space.
- QR is faster. This is perhaps because of implementation. We have written explicit loops into LBFGS, while for QR we have used optimized numpy matrices. Further LBFGS is iterative by design, and is making more complicated calculations.

PS1

September 22, 2017

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import pandas as pd
        import seaborn as sns
```

0.0.1 Data Preliminaries

Loading data

```
In [2]: # loading data using numpy
        data = np.loadtxt('CASP.csv', delimiter=',', skiprows=1)
```

```
In [3]: Y = data[:, 0]
        X = data[:,1:]
        X.shape
```

```
Out[3]: (45730, 9)
```

Splitting into training and testing data

```
In [4]: # creating training data - 90;10 split
        l = int(.9*Y.shape[0])

        Y_train = Y[:l]
        Y_test = Y[l:]
        X_train = X[:l]
        X_test = X[l:]

        assert X_train.shape[0]==.9*X.shape[0]
        assert X_test.shape[0]==0.1*X.shape[0]
```

Normalizing features.
For each feature, X_i, we have $X_i = \sigma Z + \mu \implies Z = \frac{(X_i - \mu)}{\sigma}$

```
In [5]: # normalizing test data across each axis. This returns a vector with the mean of each co
        mu_train = X_train.mean(axis=0)
        var_train = X_train.var(axis=0)

        mu_test = X_test.mean(axis=0)
        var_test = X_test.var(axis=0)

        #print(mu_x.shape)

        # renaming normalized variables; there is numpy broadcasting magic going on here: mu_x h
        # has dim (41157,9) => X_train - mu_x will have dim (41157,9)
        Z = (X_train - mu_train)/np.sqrt(var_train)
        Z_test = (X_test - mu_test)/np.sqrt(var_test)
        Z.shape
```

```
Out[5]: (41157, 9)
```

Adding bias feature (to first column)

```
In [6]: D = np.ones((Z.shape[0], Z.shape[1]+1))
        D[:, 1:] = Z
```

```
In [7]: # bias for test data
        D_test = np.ones((Z_test.shape[0], Z_test.shape[1]+1))
        D_test[:, 1:] = Z_test
```

```
In [8]: D_test
```

```
Out[8]: array([[ 1.        , -0.39767095,  0.64721867, ..., -0.64707638,
                -0.10005648,  0.7050291 ],
               [ 1.        ,  0.29926327, -0.0535137 , ..., -0.08606622,
                -0.98803634,  0.02224952],
               [ 1.        ,  0.09910086, -0.38016704, ..., -0.11615054,
                -0.5795656 , -0.22750538],
               ...,
               [ 1.        , -0.53920322, -0.37045806, ..., -0.35353095,
                -0.43748883,  0.5025857 ],
               [ 1.        , -0.25834212,  0.01129376, ..., -0.28837636,
                -0.52628681,  0.1926041 ],
               [ 1.        ,  0.68091731,  0.9475229 , ...,  0.30947023,
                 1.24967292, -0.76901449]])
```

0.0.2 Problem 4

Here, we are asked to essentially conduct ridge-regression. However, we compute this in a numerically stable and efficient method, using QR Decomposition. See Murphy 7.5.2

We begin by adding "pseudo-data" to our matrix based on our prior distribution on weights.

$$\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{ø}^2\mathbf{I})$$

```
In [9]: from scipy import linalg
        import time
```

```
In [10]: tau_inv_sq = 10.
         l = np.sqrt(10)

         # adding pseudo-data to X
         X = np.ones((D.shape[0] + D.shape[1], D.shape[1]))
         X[:D.shape[0],:] = D
         X[D.shape[0]:, :] = np.identity(D.shape[1])*l

         # adding pseudo-data to y
         y = np.zeros((D.shape[0]+D.shape[1],1))
         y[:D.shape[0], :] = Y_train[:, np.newaxis]

         print("X shape: ", X.shape)
         print("Y shape: ", y.shape)
```

```
X shape:  (41167, 10)
Y shape:  (41167, 1)
```

```
In [11]: qr_t0 = time.time()
         # QR decomposition
         Q, R= np.linalg.qr(X, mode='reduced')
         Q1, R1 = linalg.qr(X, mode='economic')
         qr_tf = time.time()
         print("qr time: ", qr_tf - qr_t0)
```

```
qr time:  0.07271003723144531
```

```
In [12]: Q.dot(R) - Q1.dot(R1)
```

```
Out[12]: array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
                [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                ...,
                [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

```
In [13]: # optimized way of finding an inverse
         inv_t0 = time.time()
         R_inv = linalg.solve_triangular(R, np.identity(R.shape[0]))
         A = np.dot(R_inv, Q.T)
         w = np.dot(A, y)
         inv_tf = time.time()
         print("inversion and product time: ", inv_tf - inv_t0)
```

```
inversion and product time:  0.0028769969940185547
```

RMSE

```
In [14]: def RMSE(X_test, w):
             error = Y_test[:, np.newaxis] - X_test.dot(w)
             sq_e = error**2
             return np.sqrt(np.sum(sq_e)/Y_test.shape[0])

         RMSE(D_test,w)

Out[14]: 5.2098893712934
```

```
In [15]: from sklearn.metrics import mean_squared_error

         y_pred = D_test.dot(w)
         np.sqrt(mean_squared_error(Y_test[:, np.newaxis], y_pred))

Out[15]: 5.2098893712934
```

```
In [16]: error = Y_test[:, np.newaxis] - D_test.dot(w)
         sq_e = error.T.dot(error)
         np.sqrt(sq_e/Y_test.shape[0])
         Y_test.shape
         w.shape

Out[16]: (10, 1)
```

0.0.3 Problem 5

We now try and compute this using PyTorch and the L-BFGS optimizer. The model is the same; we have some 10D feature set (including the bias term). We have assumed that our data was generated with additive Gaussian noise, and that we also have a prior probability distribution on our weights.

Here, the L-BFGS function essentially optimizes our log-posterior function for us.

```
In [17]: import torch
         from torch import Tensor
         from torch.autograd import Variable
```

```
In [18]: '''
         Full PyTorch treatment.

         Every matrix is constructed as PyTorch variable. Most important part is to consider the
         '''

         size = 10
         # construct a PyTorch variable array
         weights = Variable(torch.randn(10,1), requires_grad=True)
         #weights = Variable(torch.Tensor(10,1), requires_grad=True)
         #weights = Variable(torch.randn(10,1).type(Torch.DoubleTensor), requires_grad=True)
```

```
# LBFGS optimizer
optimizer = torch.optim.LBFGS([weights])

X_torch = Variable(torch.Tensor(X), requires_grad=False)
y_torch = Variable(torch.Tensor(y), requires_grad=False)

for i in range(100):
    def torch_black_box():
        # setting gradients to zero
        optimizer.zero_grad()

        # calculating the logloss
        y_pred = X_torch.mm(weights)
        v = y_torch.sub(y_pred)
        v_T = torch.transpose(v, 0,1)
        logloss = v_T.mm(v)

        #print('loss: ', logloss.data.numpy())
        # calculating the gradients
        logloss.backward()

        #print(logloss.size())
        # returns logloss; needs the [0,0] argument because logloss is a [1,1] vector
        return logloss[0,0]
    optimizer.step(torch_black_box)
```

Notice that if we just ask for weights, PyTorch lists only 4 digits after the decimal. However, this is a facet of just how it prints. We can force it to display everything by just converting to numpy

```
In [19]: weights
```

```
Out[19]: Variable containing:
          7.7415
          5.5578
          2.2519
          1.0788
         -5.9118
         -1.7348
         -1.6388
         -0.2661
          0.8178
         -0.6591
         [torch.FloatTensor of size 10x1]
```

```
In [20]: weights.data.numpy()
```

```
Out[20]: array([[ 7.74153376],
                [ 5.55781555],
                [ 2.25191164],
                [ 1.07879972],
                [-5.9117775 ],
                [-1.73480093],
                [-1.63875604],
                [-0.26610556],
                [ 0.81781441],
                [-0.65913391]], dtype=float32)
```

Because we have already put in our information about **w** into our data-set as pseudo-counts, the log-posterior distribution is simply the same as that of least-squares.

If we want to manually enter the gradient, it is simply:

$$\partial_{\mathbf{w}} = 2\mathbf{X}^{\mathbf{T}}\mathbf{X}\mathbf{w} - 2\mathbf{X}^{\mathbf{T}}\mathbf{y}$$

```
In [21]: '''
         Manual PyTorch: remember that weights.grad has to be directly upgraded. logloss has to
         '''

         weights = Variable(torch.randn(10), requires_grad=True)
         optimizer = torch.optim.LBFGS([weights])
         print(weights.size())
         for i in range(100):
             def closure():
                 optimizer.zero_grad()
                 weights_data = weights.data.numpy()

                 # computing Xw
                 Xw = X.dot(weights_data[:, np.newaxis])

                 # computing (y - Xw).T (y-Xw)
                 v = y - Xw
                 logloss = v.T.dot(v)

                 # gradient
                 grad = X.T.dot(Xw) - X.T.dot(y)
                 # changing shape of gradient back to (10,)
                 g = grad[:,0]

                 weights.grad = Variable(Tensor(g))
                 return logloss[0,0]

             optimizer.step(closure)

torch.Size([10])


In [22]: weights.data.numpy()

Out[22]: array([ 7.74153423,  5.5578208 ,  2.25190759,  1.07880139, -5.91177797,
                -1.73480332, -1.63875473, -0.26610556,  0.81781411, -0.65913397], dtype=float32)
```

0.0.4 Problem 6

First, we note that $\boldsymbol{x} \in \mathbb{R}^m, m = 9$. In other words, our current feature-space consists of a 9-dim. vector. This is excluding the bias term.

The required matrix A is then $d \times m$. b is also d-dimensional. The affine-transformation is what we compute regression on. In other words, we have:

$$y - \boldsymbol{w}^T \boldsymbol{cos}(\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b})$$

If we want to write the entire computation out in matrix form, we note that the design matrix, X is simply of dimension $N \times d$, where N is the number of data-points. X looks like:

$$X = \begin{pmatrix} \cdots \boldsymbol{cos}(\boldsymbol{A}\boldsymbol{x}^{(1)} + \boldsymbol{b})^T \cdots \\ \vdots \\ \cdots \boldsymbol{cos}(\boldsymbol{A}\boldsymbol{x}^{(n)} + \boldsymbol{b})^T \cdots \end{pmatrix}$$

The final regression simply looks like:

$$\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}$$

QR With this form, we can can add the bias term (as an extra-column) and continue with our way of add-pseudocounts to calculate the MLE.

```
In [23]: d=800
         m=9

         # constructing affine-transformation
         A = np.random.normal(size=(m,d))
         b = np.random.uniform(low=0.0, high=2*np.pi, size=d)

In [24]: # constructing linear transformation for train data
         phi = np.ones((D.shape[0], d+1))
         X1 = D[:, :9]
         X1.shape
         phi[:,1:] = np.cos(X1.dot(A))

         # constructing transformation for test data
         phi_test = np.ones((D_test.shape[0], d+1))
         X2 = D_test[:, :9]
         X2.shape
         phi_test[:,1:] = np.cos(X2.dot(A))

In [25]: X.shape

Out[25]: (41167, 10)

In [26]: phi
```

```
Out[26]: array([[ 1.        , -0.2478469 ,  0.32884896, ...,  0.86579964,
                  0.93999889,  0.15572291],
                [ 1.        ,  0.99665489,  0.94539411, ..., -0.98457515,
                 -0.48289715, -0.57179493],
                [ 1.        ,  0.88336845, -0.69312933, ..., -0.0394592 ,
                 -0.91290043, -0.89367066],
                ...,
                [ 1.        , -0.22022831,  0.17898387, ...,  0.35213247,
                  0.57747238,  0.01745905],
                [ 1.        ,  0.93094205,  0.997037  , ...,  0.99937776,
                 -0.54567086,  0.5090761 ],
                [ 1.        ,  0.96827967,  0.67535047, ...,  0.93360984,
                 -0.01811214,  0.07640484]])
```

```
In [27]: def QR(Phi):
             '''
             Takes the input matrix and produces the weight vector
             '''

             tau_inv_sq = 10.
             l = np.sqrt(10)

             # adding pseudo-data to X
             X = np.ones((Phi.shape[0] + Phi.shape[1], Phi.shape[1]))
             X[:Phi.shape[0],:] = Phi
             X[Phi.shape[0]:, :] = np.identity(Phi.shape[1])*l

             # adding pseudo-data to y
             y = np.zeros((Phi.shape[0]+Phi.shape[1],1))
             y[:Phi.shape[0], :] = Y_train[:, np.newaxis]

             print("X shape: ", X.shape)
             print("Y shape: ", y.shape)

             qr_t0 = time.time()
             # QR decomposition
             Q, R= np.linalg.qr(X, mode='reduced')
             inv_t0 = time.time()
             R_inv = linalg.solve_triangular(R, np.identity(R.shape[0]))
             A = np.dot(R_inv, Q.T)
             w = np.dot(A, y)
             inv_tf = time.time()
             print("inversion and product time: ", inv_tf - inv_t0)
             return w
```

```
In [28]: t0 = time.time()
         w = QR(phi)
         tf = time.time()-t0
         print("time", tf)
         w.shape
```

```
X shape:  (41958, 801)
Y shape:  (41958, 1)
inversion and product time:  0.5902242660522461
time 4.27150821685791
```

```
Out[28]: (801, 1)
```

```
In [29]: RMSE(phi_test,w)
```

```
Out[29]: 4.4374408003731141
```

d = 100 1. Trial 1: RMSE - 4.9274613131034215; time: 0.14908885955810547 2. Trial 2: RMSE - 5.0312886993334871; time: 0.3584740161895752 3. Trial 3: RMSE - 4.8544483102786247; time: 0.211525917053222

d = 200 1. RMSE: 4.7517805676202389; time: 0.527772903442382 2. 4.7224569427197922; time 0.4477860927581787 3. 4.7383303898457196; 0.4993929862976074

d = 400 1. 4.536067377039938; 1.7231628894805908 2. 4.5549660329440762; 2.02138614654541 3. 4.5630712841943097; 1.285445213317871

d = 800 1. 4.4547643426722745; 4.23047399520874 2. 4.4506298382389291; time 4.3787009716033936 3. 4.4374408003731141; time 4.27150821685791

```
In [30]: '''
         Full PyTorch treatment.

         Every matrix is constructed as PyTorch variable. Most important part is to consider the
         '''
         t0_torch = time.time()
         # adding pseudo-data to X
         X = np.ones((phi.shape[0] + phi.shape[1], phi.shape[1]))
         X[:phi.shape[0],:] = phi
         X[phi.shape[0]:, :] = np.identity(phi.shape[1])*l

         # adding pseudo-data to y
         y = np.zeros((phi.shape[0]+phi.shape[1],1))
         y[:phi.shape[0], :] = Y_train[:, np.newaxis]
         size = 10
         # construct a PyTorch variable array
         weights = Variable(torch.randn(d+1,1), requires_grad=True)
         #weights = Variable(torch.Tensor(10,1), requires_grad=True)
         #weights = Variable(torch.randn(10,1).type(Torch.DoubleTensor), requires_grad=True)

         # LBFGS optimizer
         optimizer = torch.optim.LBFGS([weights])

         X_torch = Variable(torch.Tensor(X), requires_grad=False)
```

```
        y_torch = Variable(torch.Tensor(y), requires_grad=False)

        for i in range(100):
            def torch_black_box():
                # setting gradients to zero
                optimizer.zero_grad()

                # calculating the logloss
                y_pred = X_torch.mm(weights)
                v = y_torch.sub(y_pred)
                v_T = torch.transpose(v, 0,1)
                logloss = v_T.mm(v)

                #print('loss: ', logloss.data.numpy())
                # calculating the gradients
                logloss.backward()

                #print(logloss.size())
                # returns logloss; needs the [0,0] argument because logloss is a [1,1] vector
                return logloss[0,0]
            optimizer.step(torch_black_box)
        tf_torch = time.time() - t0_torch
```

```
In [31]: r = RMSE(phi_test, weights.data.numpy())
         print("Time: ", tf_torch)
         print("RMSE: ", r)
```

```
Time:  11.092233180999756
RMSE:  4.43740281478
```

d=100 1. Trial 1: RMSE: 4.92746128426; Time: 1.020986795425415 2. Trial 2: RMSE: 5.03128867109; Time: 1.160470962524414 3. Trial 3: RMSE: 4.85444859004; Time: 1.0120360851287842

d=200 1. Time: 1.0120360851287842; RMSE: 4.85444859004 2. Time: 2.025758981704712; RMSE: 4.72245696161 3. 0.4993929862976074; 2.143263101577759

d=400 1. Time: 4.634213924407959; RMSE: 4.53606708706 2. Time: 4.8693249225616455; RMSE: 4.55496555986 3. Time: 4.485335111618042; RMSE: 4.56307137668

d=800 1. Time: 11.636758089065552; RMSE: 4.45467260382 2. Time: 12.416048288345337; RMSE: 4.45063399708 3. Time: 11.092233180999756; RMSE: 4.43740281478

```
In [ ]:
```