CS 189 Introduction to Machine Learning
Fall 2017

HW2

This homework is due **Friday, September 8 at 10 p.m.**.

1 Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, “HW[n] Write-Up”
2. Submit all code needed to reproduce your results, “HW[n] Code”.
3. Submit your test set evaluation results, “HW[n] Test Set”.

After you’ve submitted your homework, be sure to watch out for the self-grade form.

(a) Before you start your homework, write down your team. Who else did you work with on this homework? List names and email addresses. In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

(b) Please copy the following statement and sign next to it:

I certify that all solutions are entirely in my words and that I have not looked at another student’s solutions. I have credited all external sources in this write up.

2 Geometry of Ridge Regression

One way to interpret "ridge regression" is as the Lagrangian form of a constrained problem.

(a) Given a matrix $X \in \mathbb{R}^{n \times d}$ and a vector $\vec{y} \in \mathbb{R}^n$, define the optimization problem

$$\begin{aligned} &\text{minimize } \|\vec{y} - X\vec{w}\|_2^2. \\ &\text{subject to } \|\vec{w}\|_2^2 \leq \beta^2. \end{aligned} \tag{1}$$

Using the spirit of the method of Lagrange multipliers (wherein we replace a constraint with a penalty on the thing that we are constraining, and then adjust the level of the penalty until the solution ends up satisfying the constraint. These penalties are sometimes referred to as "shadow prices," especially in the economics literature.), rewrite the constrained optimization problem an an unconstrained optimization with the constraint incorporated within the objective function.

Recall that ridge regression is given by the unconstrained optimization problem

$$w = \arg\min_w \|\vec{y} - X\vec{w}\|_2^2 + \lambda \|\vec{w}\|_2^2. \tag{2}$$

Hence, by performing ridge regression with penalty λ, we are essentially solving a constrained least squares problem with our parameter having bounded Euclidean norm β. **Qualitatively, how would increasing β be reflected in the desired penalty λ of ridge regression?**

Solution: Let us introduce the Lagrange multiplier $\nu > 0$ into the constrained problem, thus turning the constraint $\|w\|_2^2 \leq \beta^2$ into a "penalty" $\nu(\|w\|_2^2 - \beta^2)$. The unconstrained objective therefore takes the form

$$\min_w \|\vec{y} - X\vec{w}\|_2^2 + \nu(\|\vec{w}\|_2^2 - \beta^2).$$

Note that we have to set a value of ν to try and recover the solution to the constrained problem. How do we set such a value? Typically, we do so by trying to maximize the above objective with respect to ν, since this gives us the tightest possible solution to the original problem.

Keeping ν fixed and increasing β, we see that the objective above decreases (due to the $-\nu\beta^2$ term). Thus, in order to keep the objective tight (as discussed above), the optimal value the ν must decreases as β increases (to keep the $-\nu\beta^2$ term small in absolute value.)

Connecting this back to ridge regression by noting the correspondence between the penalties ν and λ, increasing β in the constrained problem has the effect of decreasing λ for ridge regression.

Alternative geometric interpretations that explain this correspondence will also be awarded full credit.

(b) One reason why we might want to have small weights $\vec{w}$ has to do with the sensitivity of the predictor to its input. Let $\vec{x}$ be a d-dimensional list of features corresponding to a new test point. Our predictor is $\vec{w}^\top \vec{x}$. **By how much can our prediction change if nature added an arbitrary disturbance vector of length ε to the test point's features $\vec{x}$?**

Solution: Call the disturbance δx. The change in prediction is given by

$$|w^\top \delta x| \leq \|w\|_2 \|\delta x\|_2 = \varepsilon \|w\|_2, \tag{3}$$

where the first step is a result of the Cauchy-Schwarz inequality. The prediction will thus vary from the original prediction within a distance of at most $\varepsilon \|w\|_2$.

(c) **Derive that the solution to ridge regression** (2) **is given by $\hat{w}_r = (X^\top X + \lambda I)^{-1} X^\top y$. What happens when $\lambda \to \infty$?** It is for this reason that sometimes regularization is referred to as "shrinkage."

Solution: Since this was derived in lecture, we keep the discussion brief. The objective is

$$\begin{aligned} f(w) &= \|y - Xw\|_2^2 + \lambda \|w\|_2^2 \\ &= w^T (X^T X + \lambda I) w - 2y^T X w + y^T y. \end{aligned}$$

Taking the gradient with respect to w and setting it to zero, we obtain

$$0 = 2w^\top (X^T X + \lambda I) - 2y^\top X.$$

Thus, the solution is given by

$$w = (X^\top X + \lambda I)^{-1} X^\top y.$$

In order to verify that this is indeed a minimizer, one needs to find the Hessian of f and verify that this is a positive definite matrix. Clearly, the Hessian (obtained by taking the "gradient" of the gradient) is given by the matrix $X^T X + \lambda I$, which is indeed positive definite, as we will show in the next part.

As $\lambda \to \infty$ the matrix $(X^\top X + \lambda I)^{-1}$ converges to the zero matrix, and so we have $w = \vec{0}$.

(d) Note that in computing $\hat{w}_r$, we are trying to invert the matrix $X^\top X + \lambda I$ instead of the matrix $X^\top X$. **If $X^\top X$ has eigenvalues $\lambda_1, \ldots, \lambda_d$, what are the eigenvalues of $X^\top X + \lambda I$? Comment on why adding the regularizer term λI can improve the inversion operation numerically.**

Solution: The eigenvalues of $X^T X + \lambda I$ are given by $\lambda_1 + \lambda, \lambda_2 + \lambda, \cdots, \lambda_d + \lambda$. In order to see this, note that if v_i is an eigenvector of $X^\top X$ with eigenvalue λ_i, then we have

$$(X^\top X + \lambda I) v_i = \lambda_i v_i + \lambda v_i = (\lambda_i + \lambda) v_i.$$

Notice that $\lambda_i \geq 0$ since the matrix $X^\top X$ is positive semi-definite, but some eigenvalues may be very close to 0. The eigenvalues of the inverse of a matrix are the inverses of the eigenvalues. Consequently, small eigenvalues lead to numerical instability of calculating the inverse of a matrix, since the inverse blows these up. By adding a regularizer, we increase the values of the eigenvalues of the original matrix above a threshold λ, and thus improve the inversion operation.

(e) Let $d = 3$, $n = 5$, and let the eigenvalues of $X^\top X$ be given by 1000, 1 and 0.001. We must now choose between two regularization parameters $\lambda_1 = 100$ and $\lambda_2 = 0.5$. **Which do you think is a better choice for this problem and why?**

Solution: $\lambda = 0.5$ is a better option. We are looking for a regularization constant for better numerical conditioning without changing the problem substantially.

Notice that the eigenvalue 0.01 causes us numerical issues as noted above, which we would like to eliminate. We would therefore like to preserve the relative size of the original eigenvalues. By choosing $\lambda = 100$, we also eliminate the effect of the eigenvalue 1, thereby altering our problem unnecessarily.

(f) Another advantage of ridge regression can be seen for under-determined systems. Say we have the data drawn from a 5 parameter model, but only have 4 samples of it, i.e. $X \in \mathbb{R}^{4\times 5}$. Now this is clearly an underdetermined system, since $n < d$. **Show that ridge regression with $\lambda > 0$ results in a unique solution, whereas ordinary least squares has an infinite number of solutions.**

Hint: To make this point, it may be helpful to expand any vector w as $w = w_0 + X^\top a$ for $w_0 \in \mathsf{nullspace}(X)$ and some a.

Solution: First, we show that ridge regression always leads to a unique solution. We know that the minimizer is given by

$$w = (X^\top X + \lambda I)^{-1} X^\top y.$$

We also know that the eigenvalues of the matrix $X^\top X + \lambda I$ are all at least λ, and so the matrix is invertible, thus leading to a unique solution.

For ordinary least squares, let us assume that w' minimizes $\|Xw - y\|_2$, i.e., $\|Xw' - y\|_2 \leq \|Xw - y\|_2$ for all $w \in \mathbb{R}^d$. However, since $d > n$, the matrix X has a non-trivial nullspace. Take any vector w_0 in the nullspace of X and consider the new vector $w''(\alpha) = w' + \alpha w_0$ for any scalar $\alpha \in \mathbb{R}$. Notice that $\|Xw''(\alpha) - y\|_2 = \|Xw' - y\|_2$ and so the vector $w''(\alpha)$ is also a minimizer for any choice of α. We have thus shown that the least squares problem has an infinite number of solutions.

(g) (**BONUS**) For the previous part, **what will the answer be if you take the limit $\lambda \to 0$ for ridge regression?**

Solution:

Let us provide a qualitative answer to this question. Notice that for each $\lambda > 0$, we are solving a modification of the least squares problem, and for $\lambda = 0$, we are solving the original least squares problem.

However, the least squares problem has an infinite number of solutions as pointed out above, of the form $w_0 + X^\top \alpha$, where $w_0 \in \mathsf{nullspace}(X)$. As λ approaches 0 (but is still positive), these null-space vectors are not permitted in our solution. We thus approach the solution $X^\top \alpha$, which is the *minimum norm solution* of the least squares problem.

This is also known as the Moore-Penrose pseudo-inverse. Look it up!

(h) Tikhonov regularization is a general term for ridge regression, where the constraint set takes the form of an ellipsoid instead of a ball. In other words, we solve the optimization problem

$$w = \arg\min_{w} \|y - Xw\|_2^2 + \lambda \|\Gamma w\|_2^2$$

for some full rank matrix $\Gamma \in \mathbb{R}^{d \times d}$. **Derive a closed form solution to this problem.**

Solution: The objective is

$$\begin{aligned} f(w) &= \|y - Xw\|_2^2 + \lambda \|\Gamma w\|_2^2 \\ &= w^T(X^T X + \lambda I)w - 2y^T X w + y^T y. \end{aligned}$$

Taking gradients with respect to w and setting the result to zero, we obtain

$$0 = 2w^\top (X^T X + \lambda \Gamma^T \Gamma) - 2y^\top X.$$

Thus, the solution is now given by $w = (X^T X + \lambda \Gamma^T \Gamma)^{-1} X^T y$, and one can again verify that this is a minimizer by showing that the Hessian is positive definite since the matrix Γ has full rank.

3 Polynomials and invertibility

Consider using a D-degree polynomial to fit a function $y = f(x)$ with n training samples, where both x and y are scalar. We know that this is equivalent to performing linear regression with a feature matrix F constructed from the n training data sampling positions $x_1, \ldots, x_n$. Assume all the training sampling positions are non-zero, and let this mapping be given by $F = [\vec{p}_D(x_1), \ldots, \vec{p}_D(x_n)]^T$ where $\vec{p}_D(x) = [x^0, x^1, \ldots, x^D]^T$.

(a) For $n = 2$ and $D = 1$, **show that the matrix F has full rank iff $x_1 \neq x_2$.**

Solution:

$$F = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \end{bmatrix}.$$

Notice that F has full rank iff $\det(F) = x_2 - x_1 \neq 0$, which proves the required result.

(b) More generally, let us now show that the columns of F are linearly independent provided the sampling data points are distinct and $n \geq D + 1$. It suffices to consider the case $n = D + 1$ and so assume that from this point forward for all the questions about univariate polynomials.

We do this by performing the following operations in sequence. From the matrix F, subtract the first row from rows 2 through n to obtain matrix F'.

Is it true that $\det(F) = \det(F')$**?**

Hint: Think about representing the row subtraction operation using a matrix multiplication, and argue why this additional matrix must have determinant 1. (What are the eigenvalues of a triangular matrix?)

Solution: Notice that $F' = TF$, where T is an $n \times n$ lower triangular matrix with the form

$$T = \begin{bmatrix} 1 & & & & \\ -1 & 1 & & & \\ -1 & & 1 & & \\ \vdots & & & \ddots & \\ -1 & & & & 1 \end{bmatrix},$$

where missing entries denote zeroes. Also, the eigenvalues of a triangular matrix are given by its diagonal entries, and so the determinant of T is 1. Furthermore, since $\det(F') = \det(T)\det(F)$, we have $\det(F') = \det(F)$.

(c) Perform the following sequence of operations to F', and obtain the matrix F''.

i) Subtract $x_1 * \text{column}_{n-1}$ from column_n.

ii) Subtract $x_1 * \text{column}_{n-2}$ from column_{n-1}.

$\vdots$

n-1) Subtract $x_1 * \text{column}_1$ from column_2.

Write out the matrix F'' and argue why $\det(F') = \det(F'')$.

Solution: We now have $F'' = F'C$, where C is an $n \times n$ upper triangular matrix taking the form

$$C = \begin{bmatrix} 1 & -x_1 & & & \\ & 1 & -x_1 & & \\ & & \ddots & \ddots & \\ & & & 1 & -x_1 \\ & & & & 1 \end{bmatrix}.$$

By an argument similar to the previous part, we have $\det(C) = 1$, and so $\det(F') = \det(F'')$.

(d) For any square matrix $A \in \mathbb{R}^{d\times d}$ and a matrix

$$B = \begin{bmatrix} 1 & \vec{0}^\top \\ \vec{0} & A \end{bmatrix},$$

argue that the $d+1$ eigenvalues of B are given by $\{1, \lambda_1(A), \lambda_2(A), \ldots, \lambda_d(A)\}$, and conclude that $\det(B) = \det(A)$. Here, $\vec{0}$ represents a column vector of zeros in $\mathbb{R}^d$.

Solution: For every eigenvector $\vec{v}_i$ of A corresponding to the eigenvalue λ_i, form the vector $\vec{u}_i = \begin{bmatrix} 0 \\ \vec{v}_i \end{bmatrix}$. Notice that by definition, we have $B\vec{u}_i = \lambda_i B$, and so B has eigenvalue λ_i and associated eigenvector $\vec{u}_i$. Also, construct the vector $\vec{u} = \begin{bmatrix} 1 \\ \vec{0} \end{bmatrix}$, and notice that $B\vec{u} = \vec{u}$, and so B also has a 1 eigenvalue with associated eigenvector $\vec{u}$.

Since the determinant is the product of eigenvalues, the conclusion $\det(B) = \det(A)$ is immediate.

(e) **Use the above parts to show by induction that we have** $\det(F) = \prod_{1\le i<j\le n}(x_j - x_i)$**.** Consequently, the matrix X is full rank unless two data points are equal.

Hint: First show that

$$\det(F) = \left(\prod_{i=2}^{n}(x_i - x_1)\right)\det([\vec{p}_{D-1}(x_2), \vec{p}_{D-1}(x_3), \ldots, \vec{p}_{D-1}(x_n)]^T).$$

Hint Hint: You can use the fact that multiplying a row of a matrix by a constant scales the determinant by this constant. (A fact that is clear from the oriented volume interpretation of determinants.)

Solution: Performing the sequence of operations suggested in previous parts, we have

$$F'' = \begin{bmatrix} 1 & 0 & 0 & \ldots & \\ 0 & (x_2-x_1)x_2 & (x_2-x_1)x_2^2 & \ldots & (x_2-x_1)x_2^{n-2} \\ 0 & (x_3-x_1)x_3 & (x_3-x_1)x_3^2 & \ldots & (x_3-x_1)x_3^{n-2} \\ & & \vdots & & \\ 0 & (x_n-x_1)x_n & (x_n-x_1)x_n^2 & \ldots & (x_n-x_1)x_n^{n-2} \end{bmatrix}.$$

Now, we may use the hint to factor $(x_i - x_1)$ from the ith row of the matrix to obtain

$$\det(F) = \det(F'') = \left(\prod_{i=2}^{n}(x_i - x_1)\right)\det(F'''),$$

where

$$F''' = \begin{bmatrix} 1 & 0 & 0 & \ldots & \\ 0 & x_2 & x_2^2 & \ldots & x_2^{n-2} \\ 0 & x_3 & x_3^2 & \ldots & x_3^{n-2} \\ & & \vdots & & \\ 0 & x_n & x_n^2 & \ldots & x_n^{n-2} \end{bmatrix}.$$

Now using the previous part and noticing that the lower right submatrix is given by $[\vec{p}_{D-1}(x_2), \vec{p}_{D-1}(x_3), \ldots$
we have $\det(F''') = \det([\vec{p}_{D-1}(x_2), \vec{p}_{D-1}(x_3), \ldots, \vec{p}_{D-1}(x_n)]^T)$.

Now note that the matrix $[\vec{p}_{D-1}(x_2), \vec{p}_{D-1}(x_3), \ldots, \vec{p}_{D-1}(x_n)]$ has the same monomial form, but with dimension $n-1$ instead of n. Performing the operations as above by using the sample x_2, we have

$$\det(F) = \left(\prod_{i=2}^{n}(x_i - x_1)\right)\left(\prod_{j=3}^{n}(x_j - x_2)\right)\det([\vec{p}_{D-2}(x_3), \vec{p}_{D-2}(x_4), \ldots, \vec{p}_{D-2}(x_n)].$$

Unrolling each of these determinants then yields the required answer

$$\det(F) = \prod_{1\le i<j\le n}(x_j - x_i).$$

This is equivalent to performing induction (but in the reverse direction).

(f) Let us now extend this argument to features from a multidimensional space of dimension ℓ, using a multivariate polynomial of degree D.

Using a stars and bars (link is here if you did not take CS70) argument, **show that now, we have $\binom{D+\ell}{\ell}$ features for each sampling point.**

Solution: Notice that each monomial takes the form $1^{D_0}x_{i,1}^{D_1}x_{i,2}^{D_2}\dots x_{i,\ell}^{D_\ell}$, with $D_0 + D_1 + \cdots + D_\ell = D$. The number of distinct monomials is equal to the number of different way we can split D among $\ell+1$ coefficients $D_0, \dots D_\ell$. This is the standard setting for a stars and bars argument: arranging D stars and ℓ bars, we see that the number of distinct patterns formed by this combination is precisely the number of unique monomials, since D_i can be associated to the number of stars between the $i-1$th and ith bars.

The number of unique patterns is therefore $\frac{(D+\ell)!}{D!\ell!} = \binom{D+\ell}{\ell}$.

(g) Choose n sample points $\{\vec{x}_i\}_{i=1}^n$ with $\vec{x}_i = (x_{i,1}, x_{i,2}, \dots, x_{i,\ell})^T$, and stack up all their features like in part (a) to form the feature matrix F_ℓ.

First, we will show that for some choice of distinct sampling points, this may not be full rank. Let us now form a particular instance of the matrix F_ℓ by choosing $x_{i,1} = x_{i,2} = \cdots = x_{i,\ell} = \alpha_i$ for distinct α_i as $i \in \{1,2,3\dots,n\}$. **Show that this leads to an F_ℓ with linearly dependent columns no matter how many samples n you take.**

Solution: It suffices to show that two of the columns are linearly dependent. Consider the columns formed by the monomials $x_{i,1}^2$ and $x_{i,1}x_{i,2}$. Both of these are identical column vectors whose ith entry is given by α_i^2. The columns are thus linearly dependent no matter how many samples we take.

(h) To show that the matrix can be full rank, **pick a set of sampling points $\vec{x}_i$ to show that there is a way to choose the samples to get the matrix F_ℓ to be full column rank.** You are allowed to pick as many sample points as you want. Although we are only asking you to show that there exists a way to sample to achieve full rank with enough samples taken, it turns out to be true that the F_ℓ matrix will be full rank as long as $n = \binom{D+\ell}{\ell}$ "generic" points are chosen.

Hint: Leverage earlier parts of this problem if you can.

Solution: We illustrate one such choice that allows us to use the results for univariate monomials we obtained in the previous parts. Choose $x_{i,j} = (\alpha_i)^{(D+1)^{j-1}}$, for $j = 1,2,\dots,\ell$. Now, we can verify that every distinct monomial $x_{i,1}^{D_1}x_{i,2}^{D_2}\dots x_{i,\ell}^{D_\ell} = (\alpha_i)^{D_1 + D_2(D+1)^1 + \cdots + D_\ell(D+1)^{\ell-1}}$. Thus, every choice of the tuple $(D_0, D_1, \dots, D_\ell)$ results in a different power of α_i. In order to see this, note that $D_\ell D_{\ell-1}\dots D_0$ can be thought of as a D-ary number. (Think of $D = 9$.)

Evaluating F_ℓ for such a choice of parameters results in a univariate polynomial matrix as in the first parts of the problem, but with some columns missing. We know that provided the number of samples exceeds the maximum degree of the univariate polynomial, the columns are linearly independent provided $\alpha_i \neq \alpha_j$ for all pairs $i \neq j$. Thus, taking $n = (D+1)^\ell$ samples is sufficient to ensure linear independence of columns.

(Notice that D^ℓ scales like $\binom{D+\ell}{\ell}$ when D grows.)

4 Polynomials and approximation

For a p-times differentiable function $f : \mathbb{R} \to \mathbb{R}$, the Taylor series expansion of order $m \leq p-1$ about the point x_0 is given by

$$f(x) = \sum_{i=0}^{m} \frac{1}{i!} f^{(i)}(x_0)(x-x_0)^i + \frac{1}{(m+1)!} f^{(m+1)}(a(x))(x-x_0)^{m+1}. \tag{4}$$

Here, $f^{(m)}$ denotes the mth derivative of the function f, and $a(x)$ is some value between x_0 and x.

The last term of this expansion is tyically referred to as the approximation-error term when approximating $f(x)$ by an m-th degree polynomial. For functions f whose derivatives are bounded in the neighborhood of interest, if we have $|f^{(m)}(x)| \leq T$ for $x \in (x_0 - s, x_0 + s)$, then we know that for $x \in (x_0 - s, x_0 + s)$ that $|f(x) - \sum_{i=0}^{m} \frac{1}{i!} f^{(i)}(x_0)(x-x_0)^i| \leq \frac{T(x-x_0)^{m+1}}{(m+1)!}$.

(a) **Compute the 2nd, 3rd, and 4th order Taylor approximation of the following functions about the point $x_0 = 0$. Also bound the error of approximation at $x = 3$.**

i) e^x

ii) $\sin x$

Solution:

i) The nth ($n > 0$) derivative of e^x is e^x, and $e^0 = 1$. We can write the expansion about $x = 0$ as

$$\begin{aligned} f(x) \approx & 1 + \\ \text{order 1: } & 1(0+x)^1 + \\ \text{order 2: } & \frac{1}{2} 1(x-0)^2 + \\ \text{order 3: } & \frac{1}{6} 1(x-0)^3 + \\ \text{order 4: } & \frac{1}{24} 1(x-0)^4 \end{aligned}$$

At $x = 3$, $f(3) = 20.0855$, and using the expansion above, we can see that the approximations are given by

$$\begin{aligned} \text{order 2: } 1 + 3 + \frac{1}{2} 9 &= 8.5 \\ \text{order 3: } 8.5 + \frac{1}{6} 27 &= 13 \\ \text{order 4: } 13 + \frac{1}{24} 81 &= 16.375 \end{aligned}$$

which gives us the following errors:

$$\begin{aligned}
&\text{order 2: } 20.0855 - 8.5 && = 11.5855\\
&\text{order 3: } 20.0855 - 13 && = 7.0855\\
&\text{order 4: } 20.0855 - 16.375 && = 3.7105
\end{aligned}$$

Alternatively, you can use the bound $\frac{e^3 3^{D+1}}{(D+1)!}$ to bound the approximation error by a D-degree polynomial.

ii) The first 4 derivatives of $\sin x$ are given by

$$\begin{aligned}
\text{first: } & \cos x\\
\text{second: } & -\sin x\\
\text{third: } & -\cos x\\
\text{fourth: } & \sin x
\end{aligned}$$

Putting this together in the expansion about $x_0 = 0$ gives us

$$\begin{aligned}
f(x) \approx & 0+\\
\text{order 1: } & 1(0+x)^1+\\
\text{order 2: } & \frac{1}{2}0(x-0)^2+\\
\text{order 3: } & \frac{1}{6}(-1)(x-0)^3+\\
\text{order 4: } & \frac{1}{24}0(x-0)^4
\end{aligned}$$

At $x = 3$, $f(3) = 0.1411$, and using the expansion above, we can see that the approximations are given by

$$\begin{aligned}
&\text{order 2: } 0+3+0 && =3\\
&\text{order 3: } 3+\frac{-1}{6}27 && =-1.5\\
&\text{order 4: } -1.5+0 && =-1.5
\end{aligned}$$

which gives us the following errors:

$$\begin{aligned}
&\text{order 2: } 3-0.1411 && = 2.8589\\
&\text{order 3: } |-1.5-0.1411| && = 1.6411\\
&\text{order 4: } |-1.5-0.1411| && = 1.6411
\end{aligned}$$

Again, upper bounds on this are acceptable as well.

(b) Let us say we would like an accurate polynomial approximation of the functions in part (a) for all $x \in [-3,3]$. In other words, we want a polynomial of degree D (call it ϕ_D) such that $|f(x) - \phi_D(x)| \leq \varepsilon$ for all $x \in [0,3]$. **How large does D need to be as a function of ε for such a guarantee for the two choices of $f(x)$ in part (a)?**

Solution:

The error is given by the remainder term in the Taylor expansion:

$$r(x) = \frac{1}{(D+1)!} f^{(D+1)}(a(x))(x-x_0)^{D+1}$$

We can use this as an upper bound on the approximation error of the expansion.

i) Given that $f^{(D+1)}(x) = e^x$ for any D, the upper bound is given by $\max_x e^x$ in the range $x \in [-3,3]$, which gives us $\max_x e^x = e^3$. Similarly, the term $(x-x_0)^{D+1}$ is has the largest magnitude when $x = \pm 3$. Thus, we can bound the remainder term and thus the error as a function of D by $\frac{1}{(D+1)!} e^3 3^{D+1}$ and it suffices for this to be less than ε. Using Stirling's approximation $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ and simplifying, we have

$$\frac{e^3}{\sqrt{2\pi(D+1)}} \left(\frac{3e}{D+1}\right)^{D+1} \sim \varepsilon.$$

Taking the reciprocal and then logarithm of both sides, and simplifying, we obtain

$$C + (D+1/2)\left(\log(D+1) - \log(3e)\right) \sim \log(1/\varepsilon),$$

where we have collected all the constant terms into C. Since we only require a value of D to within a constant factor, we can ignore the constants above, and write

$$D\log(D) \sim \log(1/\varepsilon).$$

Let us now guess a value for D, equal to $\log(1/\varepsilon)/\log\log(1/\varepsilon)$ for some constant. Substituting this value into the LHS, we obtain

$$\frac{\log(1/\varepsilon)}{\log\log(1/\varepsilon)}\left(\log\log(1/\varepsilon) - \log\log\log(1/\varepsilon)\right) = \log(1/\varepsilon)\left(1 - \frac{\log\log\log(1/\varepsilon)}{\log\log(1/\varepsilon)}\right).$$

Notice that the ratio term is bounded above by a constant, and so the expression (ignoring constant factors) evaluates to $\log(1/\varepsilon)$ as desired. Thus $D = O\left(\log(1/\varepsilon)/\log\log(1/\varepsilon)\right)$ is optimal.

(It is sufficient to have $D = O(\log(1/\varepsilon))$ to have the error bounded by ε, and this answer will also get full credit.)

ii) Following the arguments from the previous function, we can bound the $|f^{(D+1)}(x)| \leq 1$ for any D and $x \in [-3,3]$ because all of derivatives are either $\pm \sin x$ or $\pm \cos x$, which has a maximum magnitude of 1 for any value of x. Thus, the bound on the error as a function of D can be given by $\frac{1}{(D+1)!} 3^{D+1}$. Following the steps above to bound this by ε, we are again led to the choice $D = O\left(\log(1/\varepsilon)/\log\log(1/\varepsilon)\right)$.

(c) **What is $\lim_{m\to\infty} \frac{(x-x_0)^{m+1}}{(m+1)!}$?**

Conclude that a univariate polynomial of high enough degree can approximate any function that is sufficiently smooth. This is the power of using polynomial features, even when we don't know the underlying function f that is generating our data! This universal approximation property gives us some justification for using polynomial features. (Later, we will see that neural networks are also universal function approximators.)

Solution: When m becomes sufficiently large, the term $(m+1)!$ will grow faster than $(x-x_0)^{m+1}$, which indicates that the limit goes to 0 as m approaches infinity.

Given any sufficiently smooth function, which we denote as $f(x)$, we can bound the magnitude of the derivatives of the function as $|f^{(i)}(x)| \leq T$ for some constant $T \leq \infty$ and all $i > 0$ and all x in the range of the function we are approximating. The remainder term is now bounded by

$$r(x) \leq T\frac{(x-x_0)^{m+1}}{(m+1)!}.$$

Because T is a constant, the limit as $m \to \infty$ is still 0, and therefore the remainder $r(x) \to 0$ as well. Thus, as m becomes sufficiently large, a polynomial of high enough degree can approximate any sufficiently smooth function.

(Note that here, we only deal with functions whose derivatives do not identically vanish to zero about the point x_0. This was an implicit assumption.)

(d) Now let's extend this idea of approximating functions with polynomials to multivariable functions. The Taylor series expansion for a function $f(x,y)$ about the point (x_0,y_0) is given by

$$\begin{aligned} f(x,y) = f(x_0,y_0) + f_x(x_0,y_0)(x-x_0) + f_y(x_0,y_0)(y-y_0) +& \\ \frac{1}{2!}[f_{xx}(x_0,y_0)(x-x_0)^2 + f_{xy}(x_0,y_0)(x-x_0)(y-y_0) +& \\ f_{yx}(x_0,y_0)(x-x_0)(y-y_0) + f_{yy}(x_0,y_0)(y-y_0)^2] + \ldots& \end{aligned} \tag{5}$$

where $f_x = \frac{\partial f}{\partial x}$, $f_y = \frac{\partial f}{\partial y}$, $f_{xx} = \frac{\partial^2 f}{\partial x^2}$, $f_{yy} = \frac{\partial^2 f}{\partial y^2}$, and $f_{xy} = \frac{\partial^2 f}{\partial x \partial y}$

As you can see, the Taylor series for multivariate functions quickly becomes unwieldy after the second order. Let's try to make the series a little bit more manageable. **Using matrix notation, write the expansion for a function of two variables in a more compact form up to the second order terms where $f(\vec{x}) = f(x,y)$ with $\vec{x} = [x,y]^T$ and $\vec{x}_0 = [x_0,y_0]$. Clearly define any additional vectors and matrices that you use.**

Solution:

To write the expansion using vectors and matrices, we use the Hessian matrix

$$H = \begin{bmatrix} f_{xx} & f_{yx} \\ f_{xy} & f_{yy} \end{bmatrix}.$$

We also use the vector $\nabla f(\vec{x}_0) = [f_x(\vec{x}_0), f_y(\vec{x}_0)]^\top$.

With these vectors and matrices, we can write the second order Taylor expansion as

$$f(\vec{x}) \approx f(\vec{x}_0) + \nabla f^\top (\vec{x} - \vec{x}_0) + \frac{1}{2!}(\vec{x} - \vec{x}_0)^\top H(\vec{x} - \vec{x}_0).$$

(e) To see that we can do universal approximation, first just consider that we want to approximate the function in a single straight line path. For this problem, we will use the function

$f(\vec{x}) = e^x \sin y$

where $\vec{x} = [x, y]^T$. We're interested in the path $\vec{x}(t) = [\frac{1}{\sqrt{2}}t, \frac{1}{\sqrt{2}}t]^T$. **Write the 3rd order Taylor expansion of $f(\vec{x}(t))$ for the variable t about the point $t_0 = 0$.** Use the chain rule.

Solution:

We can start by expressing the function in terms of the variable t:

$$f(\vec{x}(t)) = e^{\frac{t}{\sqrt{2}}} \sin \frac{t}{\sqrt{2}}.$$

This gives us a single-variable function, and we can take the derivatives of this like normal:

$$\begin{aligned}
f^{(1)}(\vec{x}(t)) &= \frac{1}{\sqrt{2}} e^{\frac{t}{\sqrt{2}}} \sin \frac{t}{\sqrt{2}} + \frac{1}{\sqrt{2}} e^{\frac{t}{\sqrt{2}}} \cos \frac{t}{\sqrt{2}} \\
f^{(2)}(\vec{x}(t)) &= \frac{1}{2} e^{\frac{t}{\sqrt{2}}} \sin \frac{t}{\sqrt{2}} + \frac{1}{2} e^{\frac{t}{\sqrt{2}}} \cos \frac{t}{\sqrt{2}} + \frac{1}{2} e^{\frac{t}{\sqrt{2}}} \cos \frac{t}{\sqrt{2}} - \frac{1}{2} e^{\frac{t}{\sqrt{2}}} \sin \frac{t}{\sqrt{2}} \\
&= e^{\frac{t}{\sqrt{2}}} \cos \frac{t}{\sqrt{2}} \\
f^{(3)}(\vec{x}(t)) &= \frac{1}{\sqrt{2}} e^{\frac{t}{\sqrt{2}}} \cos \frac{t}{\sqrt{2}} - \frac{1}{\sqrt{2}} e^{\frac{t}{\sqrt{2}}} \sin \frac{t}{\sqrt{2}}.
\end{aligned}$$

Using $t_0 = 0$, we can evaluate the derivates at this point:

$$\begin{aligned}
f(\vec{x}(0)) &= 0 \\
f^{(1)}(\vec{x}(0)) &= \frac{1}{\sqrt{2}} \\
f^{(2)}(\vec{x}(0)) &= 1 \\
f^{(3)}(\vec{x}(0)) &= \frac{1}{\sqrt{2}}.
\end{aligned}$$

Gathering all of these expressions, we can now write the 3rd order Taylor expansion as

$$f(\vec{x}(t)) \approx 0 + \frac{1}{\sqrt{2}}(t - t_0) + \frac{1}{2!} 1 (t - t_0)^2 + \frac{1}{3!} \frac{1}{\sqrt{2}} (t - t_0)^3$$

(f) Similar to part **(b)**, **determine the degree D for the polynomial $\phi_D(\vec{x}(t))$ from the Taylor series about the point $t_0 = 0$ such that $|f(\vec{x}(t)) - \phi_D(\vec{x}(t))| \leq \varepsilon$ on the interval $t \in [0,3]$ for the function from the previous part.**

Solution:

If we take the 4th derivative of the function, we see that

$$\begin{aligned} f^{(4)}(\vec{x}(t)) &= \frac{1}{2}e^{\frac{t}{\sqrt{2}}}\cos\frac{t}{\sqrt{2}} - \frac{1}{2}e^{\frac{t}{\sqrt{2}}}\sin\frac{t}{\sqrt{2}} - \frac{1}{2}e^{\frac{t}{\sqrt{2}}}\sin\frac{t}{\sqrt{2}} + \frac{1}{2}e^{\frac{t}{\sqrt{2}}}\cos\frac{t}{\sqrt{2}} \\ &= -e^{\frac{t}{\sqrt{2}}}\sin\frac{t}{\sqrt{2}} \\ &= -f(\vec{x}(t)). \end{aligned}$$

This indicates that the value of the nth order derivative does not grow as n becomes larger. Therefore, the largest magnitude of any order derivative is given by

$$|f^{(i)}(\vec{x}(t))| \leq \max_t |e^{\frac{t}{\sqrt{2}}}\sin\frac{t}{\sqrt{2}}|$$

OR

$$|f^{(i)}(\vec{x}(t))| \leq \max_t |e^{\frac{t}{\sqrt{2}}}\cos\frac{t}{\sqrt{2}}|$$

on the interval $t \in [0,3]$

Plugging in values, we see that the maximum is achieved for $t = 3$ with $e^{\frac{3}{\sqrt{2}}}\sin\frac{3}{\sqrt{2}} = 7.11$.

Thus, the remainder term of the Taylor series is bounded by

$$\left|\frac{1}{(D+1)!}f^{(D+1)}(a(t))(t-t_0)^{D+1}\right| \leq 7.11 * \frac{1}{(D+1)!} * (t-t_0)^{D+1}.$$

Because $t \in [0,3]$ and $t_0 = 0$, we want to choose a D that satisfies the following inequality to ensure that the approximation is within ε of the original function

$$7.11 * \frac{1}{(D+1)!} * 3^{D+1} \leq \varepsilon$$

Proceeding as in the previous part leads us to the choice $D = O(\log(1/\varepsilon)/\log\log(1/\varepsilon))$.

(g) BONUS: **Sketch how the argument above can be extended to show that multivariate polynomials are universal function approximators for sufficiently smooth functions of many variables.**

Solution:

We just showed that we can approximate a smooth function along a particular direction given a high degree polynomial. In higher (say d) dimensions, we can use the same argument to approximate the function along d orthogonal directions to obtain d approximating polynomials. Now any other direction in d dimensions can be approximated by using a combination of approximations along the d directions we just approximated using polynomials, but making the degrees of our polynomials sufficiently large.

Here, the degrees of the approximating polynomials get much larger than when we only care about approximation along one direction, but still, there exists a high enough degree (as before) that approximates the function within arbitrary precision ε.

5 Jaina and her giant peaches

NOTE: In response to student questions, a new data file 1D_plot_new.mat was added to the website at 3.30PM on September 7th. You will be awarded full credit for implementations on the first data file 1D_poly.mat, but the second dataset 1D_poly_new.mat is bonus. Please use the new file to see the expected behavior of errors and for extra credit!

In another alternative universe, Jaina is a mage testing how long she can fly a collection of giant peaches. She has n training peaches – with masses given by $x_1, x_2, \ldots x_n$ – and flies all the peaches once to collect training data. The experimental flight time of peach i is given by y_i. She believes that the flight time is well approximated by a polynomial function of the mass, and her goal is to fit a polynomial of degree D to this data. Include all text responses and plots in your write-up.

(a) **Show how Jaina's problem can be formulated as a linear regression problem.**

Solution: The problem is to find the coefficients w_d such that the squared error is minimized:

$$\min_{w_0,\ldots,w_D} \sum_i (w_0 + w_1 x_i + w_2 x_i^2 \cdots + w_D x_i^D - y_i)^2$$

Assume that we construct the following matrix:

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^D \\ 1 & x_2 & x_2^2 & \ldots & x_2^D \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^D \end{bmatrix}$$

Then the problem can be written as:

$$\min_{\vec{w}} ||X\vec{w} - \vec{y}||^2$$

(b) You are given data of the masses $\{x_i\}_{i=1}^n$ and flying times $\{y_i\}_{i=1}^n$ in the "x_train" and "y_train" keys of the file 1D_POLY.MAT, respectively (the new data is in the file 1D_POLY_NEW.MAT), with the masses centered and normalized to lie in the range $[-1,1]$. **Write a routine to do a least-squares fit (taking care to include a constant term) of a polynomial function of degree D to the data.** Letting f_D denote the fitted polynomial, **plot the average training error $R(D) = \frac{1}{n}\sum_{i=1}^n (y_i - f_D(x_i))^2$ against D in the range $D \in \{1,2,3,\ldots,n-1\}$.**

Solution:

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as spio

data = spio.loadmat('1D_poly_new.mat', squeeze_me=True) #loading 1D_poly.mat also
    acceptable
x_train = np.asmatrix(data['x_train'])
y_train = np.asmatrix(data['y_train'])
y_train = y_train.transpose()

n = 20
Training_Error = np.zeros((n-1,1))

for j in range(n-1):
    D = j+1
    #Xf = extract_features(x_train ,D)
    for i in range(D+1):
        if i==0:
            Xf = np.asmatrix([1]*n)
        else:
            Xf = np.vstack([np.power(x_train ,i),Xf])
    Xf = Xf.transpose()

    # ordinary least squares
    W1 = np.linalg.solve(Xf.transpose().dot(Xf),Xf.transpose().dot(y_train));
    y_predicted = Xf.dot(W1)
    Training_Error[j] = (np.linalg.norm(y_train-y_predicted)**2)/n

plt.plot(Training_Error)
plt.xlabel('degree of polynomial')
plt.ylabel('Training Error')
```

(c) **How does the average training error behave as a function of D, and why? What happens if you try to fit a polynomial of degree n with a standard matrix inversion method?**

Solution: The training error decreases since we have more degrees of freedom to fit the dataset. If we try to fit a polynomial of degree $n-1$, we will have enough parameters to fit the data exactly, so the training error will be zero. Using a polynomial of degree n results in a non-invertible data matrix, and so we cannot use a standard matrix inversion method to find our solution.

(d) Jaina has taken Mystical Learning 189, and so decides that she needs to run another experiment before deciding that her prediction is true. She runs another fresh experiment of flight times

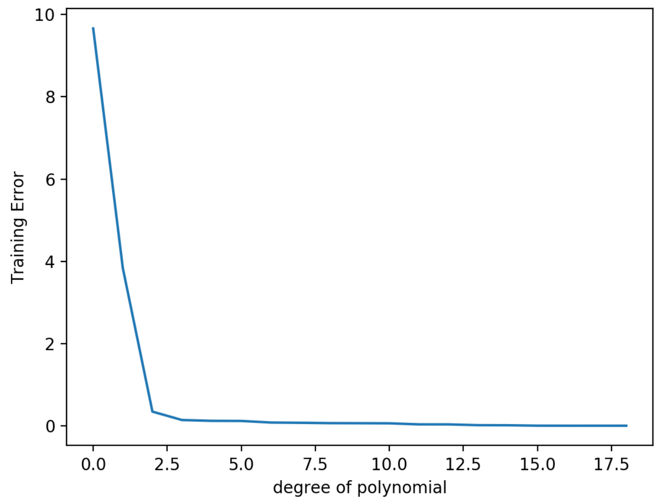

Figure 1: Training Error: **Result for 1D_poly.mat**

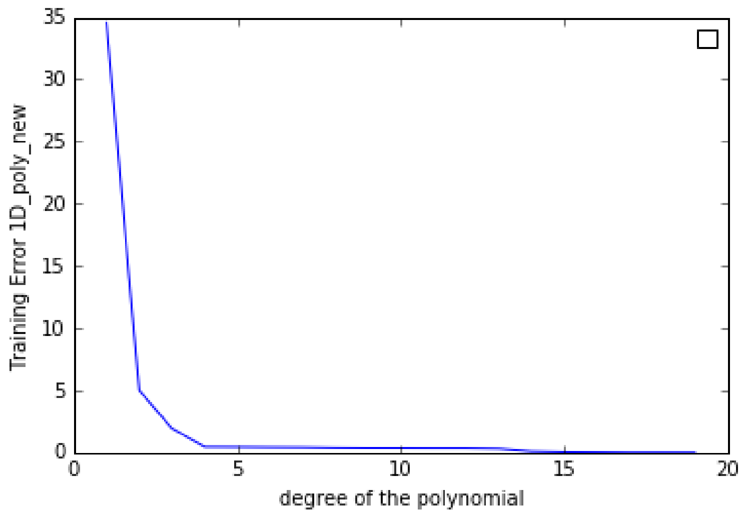

Figure 2: Training Error: **Result for 1D_poly_new.mat**

using the same peaches, to obtain the data with key "y_fresh" in 1D_POLY.MAT. Denoting the fresh flight time of peach i by $\tilde{y}_i$, **plot the average error $\tilde{R}(D) = \frac{1}{n}\sum_{i=1}^{n}(\tilde{y}_i - f_D(x_i))^2$ for the same values of D as in part (b) using the polynomial approximations f_D also from the previous part. How does this plot differ from the plot in (b) and why?**

Solution: The plots are shown in Figures 3 and 4. The plots are different from the training errors since the data was generated afresh. While increasing the polynomial degree served to fit the noise in the training data, we cannot hope to fit noise by using the same model for fresh data. Hence, our performance degrades as we increase polynomial degree, with a minimum seen at the true model order.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as spio

data = spio.loadmat('1D_poly_new.mat', squeeze_me=True) #loading file 1D_poly.mat
    also accepted
x_train = np.asmatrix(data['x_train'])
y_train = np.asmatrix(data['y_train'])
y_train = y_train.transpose()
y_fresh = np.asmatrix(data['y_fresh'])
y_fresh = y_fresh.transpose()

n = 20
R1 = np.zeros((n-1,1))
R2 = np.zeros((n-1,1))


for j in range(n-1):
    D = j+1
    #Xf = extract_features(x_train ,D)
    for i in range(D+1):
        if i==0:
            Xf = np.asmatrix([1]*n)
        else:
            Xf = np.vstack([np.power(x_train ,i),Xf])
    Xf = Xf.transpose()
    # ordinary least squares
    W1 = np.linalg.solve(Xf.transpose().dot(Xf),Xf.transpose().dot(y_train));
    y_predicted = Xf.dot(W1)
    R1[j] = (np.linalg.norm(y_train-y_predicted)**2)/n
    R2[j] = (np.linalg.norm(y_fresh-y_predicted)**2)/n

plt.plot(R2)
plt.xlabel('degree of the polynomial')
plt.ylabel('Fresh Error')
```

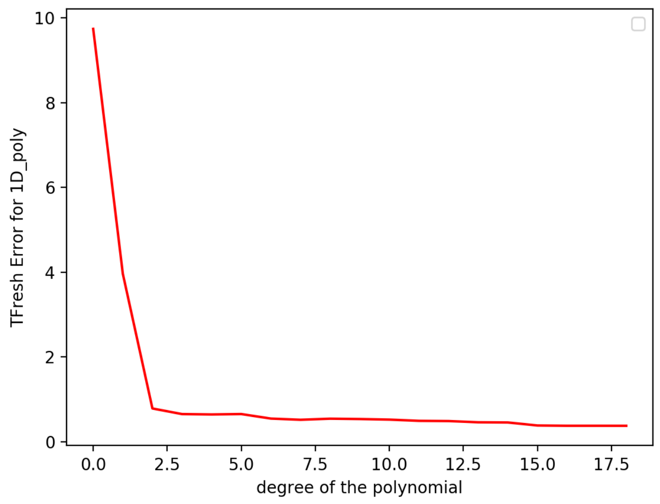

Figure 3: Fresh Error yfresh: **Result for 1D_poly.mat**

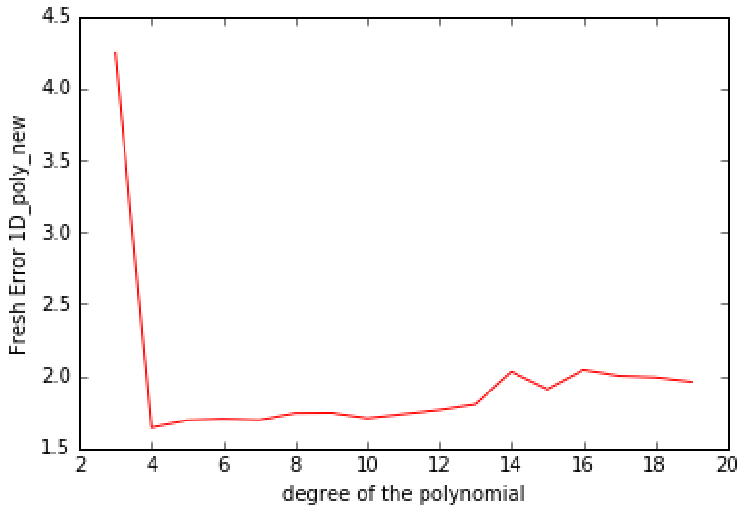

Figure 4: Fresh Error yfresh: **Result for 1D_poly_new.mat**

(e) **How do you propose using the two plots from parts (b) and (d) to "select" the right polynomial model for Jaina?**

Solution: The right model is the one that minimizes the error in the fresh dataset. In the dataset 1D_poly.mat, this minimizer was given by $D = 19$. In the new dataset 1D_poly_new.mat, the minimizer is at $D = 4$.

(f) Jaina has a new hypothesis – the flying time is actually a function of the mass, smoothness, size, and sweetness of the peach, and some multivariate polynomial function of all of these parameters. The data in POLYNOMIAL_REGRESSION_SAMPLES.MAT (100000×5) with columns corresponding to the 5 attributes of the peach. **Use 4-fold cross-validation to decide which of $D \in \{1,2,3,4\}$ is the best fit for the data provided**. For this part, compute the polynomial coefficients via ridge regression with penalty $\lambda = 0.1$, instead of ordinary least squares.

Solution: Please refer to the next part for the general implementation.

(g) Now **redo the previous part, but use 4-fold cross-validation on all combinations of $D \in \{1,2,3,4\}$ and $\lambda \in \{0.05, 0.1, 0.15, 0.2\}$** - this is referred to as a grid search. **Find the best D and λ that best explains the data using ridge regression.**

Solution: In the following implementation, we run cross-validation to find errors for each D separately, by varying the value of λ. This is purely for pedagogical purposes. You can combine these two steps by running through D in an outer loop. Note that we can collect all of these errors, and then choose the model that minimizes the cross-validation error.

Also, we construct the feature matrix without additional libraries; note however that you may use the itertools library to generate polynomial features in order to make your code more compact.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as spio
import plotly.plotly as py
import plotly.graph_objs as go
import pandas as pd


data = spio.loadmat('polynomial_regression_samples.mat', squeeze_me=True)
data_x = data['x']
data_y = data['y']
Kc = 4 # for cross validation
Ntotoal = data_x.shape[0]
Ns = int(data_x.shape[0]*(Kc-1)/Kc) # training
n = data_x.shape[1]
Nv = int(Ns/(Kc-1)) # validation

Etrain = np.zeros((4,5))
Evalid = np.zeros((4,5))
Ecv = np.zeros((4,1))
Ecvt = np.zeros((4,1))

D = 1
a = [0,0.05,0.1,0.15,0.2]



for l in range(5):
    lmbda = a[l]
    for c in range(4):
        xv = data_x[c*Nv:(c+1)*Nv]
        yv = data_y[c*Nv:(c+1)*Nv]
        x = np.delete(data_x, list(range(c*Nv,(c+1)*Nv)),0)
        y = np.delete(data_y, list(range(c*Nv,(c+1)*Nv)))

        X = np.concatenate([x,np.ones((Ns,1))],axis=1)
        Nf = 6 # number of features
        Xf = np.zeros((Ns,Nf))
        for i in range(Ns):
            k = 0
            for i1 in range(n+1):
                Xf[i][k] = X[i][i1]
                k += 1

        Xv = np.concatenate([xv,np.ones((Nv,1))],axis=1)
        Xfv = np.zeros((Nv,Nf))
        for i in range(Nv):
            k = 0
            for i1 in range(n+1):
                Xfv[i][k] = Xv[i][i1]
                k += 1

        W = np.linalg.solve(Xf.transpose().dot(Xf)+lmbda*np.eye(Nf),Xf.transpose().
    dot(y))
        y_predicted = Xfv.dot(W)
        Ecv[c] = np.linalg.norm(yv-y_predicted)**2
        y_predicted = Xf.dot(W)
        Ecvt[c] = np.linalg.norm(y-y_predicted)**2

    Evalid[D-1,l]=np.mean(Ecv)
    Etrain[D-1,l]=np.mean(Ecvt)

```

```
D = 2

for l in range(5):
    lmbda = a[l]
    for c in range(4):
        xv = data_x[c*Nv:(c+1)*Nv]
        yv = data_y[c*Nv:(c+1)*Nv]
        x = np.delete(data_x,list(range(c*Nv,(c+1)*Nv)),0)
        y = np.delete(data_y,list(range(c*Nv,(c+1)*Nv)))

        X = np.concatenate([x,np.ones((Ns,1))],axis=1)
        Nf = 21 # number of features
        Xf = np.zeros((Ns,Nf))
        for i in range(Ns):
            k = 0
            for i1 in range(n+1):
                for i2 in range(i1,n+1):
                    Xf[i][k] = X[i][i1]*X[i][i2]
                    k += 1

        Xv = np.concatenate([xv,np.ones((Nv,1))],axis=1)
        Xfv = np.zeros((Nv,Nf))
        for i in range(Nv):
            k = 0
            for i1 in range(n+1):
                for i2 in range(i1,n+1):
                    Xfv[i][k] = Xv[i][i1]*Xv[i][i2]
                    k += 1


        W = np.linalg.solve(Xf.transpose().dot(Xf)+lmbda*np.eye(Nf),Xf.transpose().
    dot(y))
        y_predicted = Xfv.dot(W)
        Ecv[c] = np.linalg.norm(yv-y_predicted)**2
        y_predicted = Xf.dot(W)
        Ecvt[c] = np.linalg.norm(y-y_predicted)**2

    Evalid[D-1,l]=np.mean(Ecv)
    Etrain[D-1,l]=np.mean(Ecvt)


D = 3
for l in range(5):
    lmbda = a[l]
    for c in range(4):
        xv = data_x[c*Nv:(c+1)*Nv]
        yv = data_y[c*Nv:(c+1)*Nv]
        x = np.delete(data_x,list(range(c*Nv,(c+1)*Nv)),0)
        y = np.delete(data_y,list(range(c*Nv,(c+1)*Nv)))

        X = np.concatenate([x,np.ones((Ns,1))],axis=1)
        Nf = 56 # number of features
        Xf = np.zeros((Ns,Nf))
        for i in range(Ns):
            k = 0
            for i1 in range(n+1):
                for i2 in range(i1,n+1):
                    for i3 in range(i2,n+1):
                        Xf[i][k] = X[i][i1]*X[i][i2]*X[i][i3]
                        k += 1

        Xv = np.concatenate([xv,np.ones((Nv,1))],axis=1)
        Nf = 56 # number of features
        Xfv = np.zeros((Nv,Nf))
        for i in range(Nv):
            k = 0
            for i1 in range(n+1):
                for i2 in range(i1,n+1):
                    for i3 in range(i2,n+1):
                        Xfv[i][k] = Xv[i][i1]*Xv[i][i2]*Xv[i][i3]
                        k += 1
        W = np.linalg.solve(Xf.transpose().dot(Xf)+lmbda*np.eye(Nf),Xf.transpose().
    dot(y));
        y_predicted = Xfv.dot(W)
        Ecv[c] = np.linalg.norm(yv-y_predicted)**2
        y_predicted = Xf.dot(W)
        Ecvt[c] = np.linalg.norm(y-y_predicted)**2

    Evalid[D-1,l]=np.mean(Ecv)
    Etrain[D-1,l]=np.mean(Ecvt)


D = 4
for l in range(5):
    lmbda = a[l]
    for c in range(4):
        xv = data_x[c*Nv:(c+1)*Nv]
        yv = data_y[c*Nv:(c+1)*Nv]
        x = np.delete(data_x,list(range(c*Nv,(c+1)*Nv)),0)
        y = np.delete(data_y,list(range(c*Nv,(c+1)*Nv)))

        X = np.concatenate([x,np.ones((Ns,1))],axis=1)
        Nf = 126 # number of features
        Xf = np.zeros((Ns,Nf))
        for i in range(Ns):
            k = 0
            for i1 in range(n+1):
                for i2 in range(i1,n+1):
                    for i3 in range(i2,n+1):
                        for i4 in range(i3,n+1):
                            Xf[i][k] = X[i][i1]*X[i][i2]*X[i][i3]*X[i][i4]
                            k += 1

        Xv = np.concatenate([xv,np.ones((Nv,1))],axis=1)
        Nf = 126 # number of features
        Xfv = np.zeros((Nv,Nf))
        for i in range(Nv):
            k = 0
            for i1 in range(n+1):
                for i2 in range(i1,n+1):
                    for i3 in range(i2,n+1):
                        for i4 in range(i3,n+1):
                            Xfv[i][k] = Xv[i][i1]*Xv[i][i2]*Xv[i][i3]*Xv[i][i4]
                            k += 1
        W = np.linalg.solve(Xf.transpose().dot(Xf)+lmbda*np.eye(Nf),Xf.transpose().
    dot(y));
        y_predicted = Xfv.dot(W)
        Ecv[c] = np.linalg.norm(yv-y_predicted)**2
```

```
            y_predicted  =  Xf.dot(W)
            Ecvt[c]  =  np.linalg.norm(y-y_predicted)**2

        Evalid[D-1,l]=np.mean(Ecv)
        Etrain[D-1,l]=np.mean(Ecvt)
```

Minimum of cross validation error happens at $D = 3$ and $\lambda = 0.15$

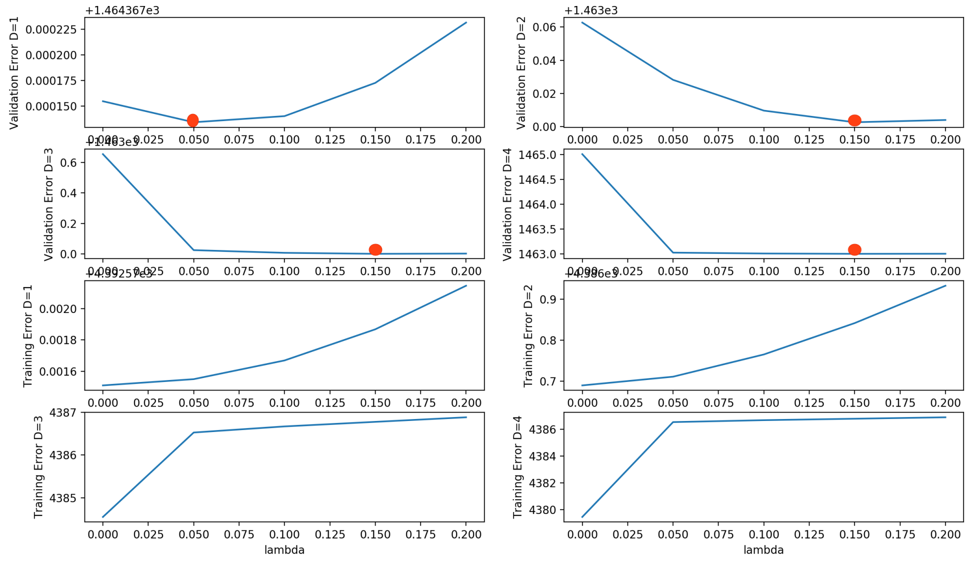

Figure 5: Training and Cross Validation Errors

6 Your Own Question

Write your own question, and provide a thorough solution.

This problem should show your understanding of a key concept in the class.

Writing your own problems is a very important way to really learn material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.