Question 1. Getting Started

Read through this page carefully. You may typeset your homework in latex or submit neatly handwritten/scanned solutions. Please start each question on a new page. Delivera

1. Submit a PDF of your writeup, **with an appendix for your code**, to assignment on Gradescope, "¡¡title¿¿ Write-Up". If there are graphs, include those graphs in the
2. If there is code, submit all code needed to reproduce your results, "¡¡title¿¿ Code".
3. If there is a test set, submit your test set evaluation results, "¡¡title¿¿ Test Set".

After you've submitted your homework, watch out for the self-grade form.

(a) Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

I work with Weiran Liu. I made stupid mistakes and struggled to correct GSI's mistakes at the same time.

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadverdently cheats.

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.

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.

Question 2. Geometry of Ridge Regression You recently learned ridge regression and how it differs from ordinary least squares. In this question we will explore how ridge regression is related to solving a constrained least squares problem in terms of their parameters and solutions.

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

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

We can utilize Lagrange multipliers to incorporate the constraint into the objective function by adding a term which acts to "penalize" the thing we are constraining. **Rewrite the constrained optimization problem into an unconstrained optimization problem.**

What the hack is Largrange multiplier? I checked Wikipedia and learned that. It's something like this:

$$\text{minimize } \|\vec{y} - \mathbb{X}\vec{w}\|_2^2 + \lambda(\|\vec{w}\|_2^2 - \beta^2)$$

λ is the so-called Largrange multiplier.

(b) Recall that ridge regression is given by the unconstrained optimization problem

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

One way to interpret "ridge regression" is as the Lagrangian form of a constrained problem. **Qualitatively, how would increasing β in our previous problem be reflected in the desired penalty λ of ridge regression (i.e. if our threshold β increases, what should we do to λ)?**

Haha! I have watched Jonathan Schewchuk's lectures so I know the geometric interpretation of the Ridge regression. Qualitatively, increasing β would decrease λ. β is like the radius of the sphere and λ is the penalty.

(c) 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 $\vec{w}^\top \vec{x}$. **What is an upper bound on how much our prediction could change if we added noise $\vec{\epsilon} \in \mathbb{R}^d$ to a test point's features $\vec{x}$?**

It turns out that we only know the norms of w and ϵ. That's interesting. We have:

$$||\vec{w}^\top \vec{\epsilon}|| \leq ||\vec{w}^\top||_2 ||\vec{\epsilon}||_2 < \sqrt{\beta}||\vec{\epsilon}||_2$$

On Piazza, it says this part is unrelated from the previous parts so the last inequality might be removed.

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

$$
\begin{aligned}
& \nabla(\|\vec{y} - \mathbb{X}\vec{w}\|_2^2 + \lambda\|\vec{w}\|_2^2) \\
= \; & \nabla((\vec{y}^\top - \vec{w}^\top\mathbb{X}^\top)(\vec{y} - \mathbb{X}\vec{w}) + \lambda\vec{w}^\top\vec{w}) \\
= \; & \nabla(\vec{y}^\top\vec{y} - \vec{w}^\top\mathbb{X}^\top\vec{y} - \vec{y}^\top\mathbb{X}\vec{w} + \vec{w}^\top\mathbb{X}^\top\mathbb{X}\vec{w}) + \lambda\vec{w}^\top\vec{w}) \\
= \; & \nabla(\vec{y}^\top\vec{y} - 2\vec{y}^\top\mathbb{X}\vec{w} + \vec{w}^\top\mathbb{X}^\top\mathbb{X}\vec{w} + \lambda\vec{w}^\top\vec{w}) \\
= \; & \nabla(-2\vec{y}^\top\mathbb{X}\vec{w} + \vec{w}^\top\mathbb{X}^\top\mathbb{X}\vec{w} + \lambda\vec{w}^\top\vec{w}) \\
= \; & -2\mathbb{X}^\top\vec{y} + 2\mathbb{X}^\top\mathbb{X}\vec{w} + 2\lambda\mathbb{I}\vec{w}
\end{aligned}
$$

$$
\begin{aligned}
0 &= -2\mathbb{X}^\top\vec{y} + 2\mathbb{X}^\top\mathbb{X}\,\hat{\vec{w}} + 2\lambda\mathbb{I}\,\hat{\vec{w}} \\
\mathbb{X}^\top\vec{y} &= \mathbb{X}^\top\mathbb{X}\,\hat{\vec{w}} + \lambda\mathbb{I}\,\hat{\vec{w}} \\
\hat{\vec{w}} &= (\mathbb{X}^\top\mathbb{X} + \lambda\mathbb{I})^{-1}\mathbb{X}^\top\vec{y}
\end{aligned}
$$

when $\lambda \to \infty$, it means there is infinite penalty on $\vec{w}$. Then the only solution would be the trivial one $\hat{\vec{w}} = \vec{0}$.

(e) Note that in computing $\hat{\vec{w}}$, we are trying to invert the matrix $\mathbb{X}^\top\mathbb{X} + \lambda\mathbb{I}$ instead of the matrix $\mathbb{X}^\top\mathbb{X}$. If $\mathbb{X}^\top\mathbb{X}$ has eigenvalues $\sigma_1^2, \ldots, \sigma_d^2$, what are the eigenvalues of $\mathbb{X}^\top\mathbb{X} + \lambda\mathbb{I}$? Comment on why adding the regularizer term $\lambda\mathbb{I}$ can improve the inversion operation numerically.

For a given eigenvalue σ_i^2 for $\mathbb{X}^\top\mathbb{X}$ we can calculate its corresponding eigenvalue for $\mathbb{X}^\top\mathbb{X} + \lambda\mathbb{I}$ denoted as λ_i:

$$
\begin{aligned}
|\lambda_i\mathbb{I} - \mathbb{X}^\top\mathbb{X} + \lambda\mathbb{I}| &= 0 \\
|(\lambda_i - \lambda)\mathbb{I} - \mathbb{X}^\top\mathbb{X}| &= 0 \\
\lambda_i &= \sigma_i^2 + \lambda
\end{aligned}
$$

Adding the regularizer term can increase all eigenvalues by the same amount which prevent $\mathbb{X}^\top\mathbb{X}$ from being singular and improve numerical calculation of its inverse.

(f) Let the number of parameters $d = 3$ and the number of datapoints $n = 5$, and let the eigenvalues of $\mathbb{X}^\top\mathbb{X}$ 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?

I would choose $\lambda_2 = 0.5$ so that we can increase the minimum eigenvalue without introducing too much bias.

(g) 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 training samples of it, i.e. $\mathbb{X} \in \mathbb{R}^{4\times 5}$. 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 $\vec{w}$ as $w(\vec{\alpha}) = \vec{w_0} + \mathbb{X}^\top \vec{\alpha}$ for $\vec{w_0} \in \mathsf{nullspace}(\mathbb{X})$ and some $\vec{\alpha} \in \mathbb{R}$.

For ordinary least square, we know $\mathbb{X}^\top\mathbb{X}$ is not full rank because $rank(\mathbb{X}^\top\mathbb{X}) <= rank(\mathbb{X})$ and $n < d$. Therefore, $\mathbb{X}^\top\mathbb{X}\vec{w} = \mathbb{X}^\top\vec{y}$ has infinite number of solutions. The other way to think about it is that we can always find a $\vec{w_0} \in nullspace(\mathbb{X})$ and add that to our original $\vec{w}$ will not change $\mathbb{X}\vec{w}$

For ridge regression, we know $(\mathbb{X}^\top\mathbb{X} + \lambda\mathbb{I})$ is invertible so there is an unique solution.

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

It will be same as ordinary least square. So both have infinite number of solutions. My friend told me I need to add this to get credit. So here you are:
$\lim_{\lambda\to 0} w = X^T\alpha$. Why is that? Because when $\lambda \neq 0$, we cannot add w_0 to our solution.

The purpose of this is to introduce kernelization, but the way the question is being asked is not clear at all.

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

$$\vec{w} = \arg\min_{\vec{w}} \frac{1}{2}\|\vec{y} - \mathbb{X}\vec{w}\|_2^2 + \lambda\|\Gamma\vec{w}\|_2^2$$

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

$$\begin{aligned}
& \nabla(\tfrac{1}{2}\|\vec{y} - \mathbb{X}\vec{w}\|_2^2 + \lambda\|\Gamma\vec{w}\|_2^2) \\
= \; & \nabla(\tfrac{1}{2}(\vec{y}^\top - \vec{w}^\top\mathbb{X}^\top)(\vec{y} - \mathbb{X}\vec{w}) + \lambda\vec{w}^\top\Gamma^\top\Gamma\vec{w}) \\
= \; & \nabla(\tfrac{1}{2}(\vec{y}^\top\vec{y} - \vec{w}^\top\mathbb{X}^\top\vec{y} - \vec{y}^\top\mathbb{X}\vec{w} + \vec{w}^\top\mathbb{X}^\top\mathbb{X}\vec{w}) + \lambda\vec{w}^\top\Gamma^\top\Gamma\vec{w}) \\
= \; & \nabla(\tfrac{1}{2}\vec{y}^\top\vec{y} - \vec{y}^\top\mathbb{X}\vec{w} + \tfrac{1}{2}\vec{w}^\top\mathbb{X}^\top\mathbb{X}\vec{w} + \lambda\vec{w}^\top\vec{w}) \\
= \; & \nabla(-\vec{y}^\top\mathbb{X}\vec{w} + \tfrac{1}{2}\vec{w}^\top\mathbb{X}^\top\mathbb{X}\vec{w} + \lambda\vec{w}^\top\Gamma^\top\Gamma\vec{w}) \\
= \; & -\mathbb{X}^\top\vec{y} + \mathbb{X}^\top\mathbb{X}\vec{w} + 2\lambda\Gamma^\top\Gamma\vec{w}
\end{aligned}$$

$$\begin{aligned}
0 &= -\mathbb{X}^\top\vec{y} + \mathbb{X}^\top\mathbb{X}\,\vec{\hat{w}} + 2\lambda\Gamma^\top\Gamma\,\vec{\hat{w}} \\
\mathbb{X}^\top\vec{y} &= \mathbb{X}^\top\mathbb{X}\,\vec{\hat{w}} + 2\lambda\Gamma^\top\Gamma\,\vec{\hat{w}} \\
\vec{\hat{w}} &= (\mathbb{X}^\top\mathbb{X} + 2\lambda\Gamma^\top\Gamma)^{-1}\mathbb{X}^\top\vec{y}
\end{aligned}$$

Question 3. Polynomials and invertibility

This problem will walk through the properties of a feature matrix based on univariate polynomials and multivariate polynomials.

First, we consider fitting a function $y = f(x)$ where both x and y are scalars, using univariate polynomials. Given n training data points $\{(x_i, y_i), i = 1, \ldots, n\}$, our task reduces to performing linear regression with a feature matrix $\mathbb{F}$ where

$$\mathbb{F} = [\vec{p}_D(x_1), \ldots, \vec{p}_D(x_n)]^T \quad \text{and} \quad \vec{p}_D(x) = [x^0, x^1, \ldots, x^D]^T.$$

Note that $\mathbb{F} \in \mathbb{R}^{n \times (D+1)}$ and $\vec{y} = [y_1, \ldots, y_n]^T \in \mathbb{R}^n$.

Parts (a)–(e) study the rank of the feature matrix $\mathbb{F}$ as a function of the points x_i's. These parts are elementary and they are **optional** if you are already familiar with this material and feel comfortable in deriving the determinant of $\mathbb{F}$. In this case we encourage you to do Problem 6 about non-linear classification boundaries instead.

In parts (f)–(h), we consider the case when the sampling points are vectors $\vec{x}_i$ and we use multivariate polynomials $\vec{p}_D(\vec{x}_i)$ as the rows of the feature matrix. These parts are mandatory.

(a) **(OPTIONAL)** For $n = 2$ and $D = 1$, **show that the matrix $\mathbb{F}$ has full rank iff $x_1 \neq x_2$.** *Note that iff stands for if and only if.*

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

Are you kidding me? Isn't that obvious? Ok, let's do it.

1. If $rank(\mathbb{F}) = 2$, we know $det(\mathbb{F}) \neq 0$. Therefore $x_1 \neq x_2$.
2. If $x_1 \neq x_2$, we know $det(\mathbb{F}) \neq 0$. Therefore $rank(\mathbb{F}) = 2$.

(b) **(OPTIONAL)** From parts (b) through (e), we work through different steps to establish that the columns of $\mathbb{F}$ are linearly independent if the sampling data points are distinct and $n \geq D + 1$. Note that it suffices to consider the case $n = D + 1$. In other words, we have $D + 1$ sample points and have constructed a square feature matrix $\mathbb{F}$. Now as a first step, construct a matrix $\mathbb{F}'$ from the matrix $\mathbb{F}$ via this operation: subtract the first row of $\mathbb{F}$ from its rows 2 through n. **Is it true that** $\det(\mathbb{F}) = \det(\mathbb{F}')$**?**

Hint: Think about representing the row subtraction operation using a matrix multiplication, and then take determinants.

$$\mathbb{F}' = \mathbb{A}\mathbb{F} = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ -1 & 1 & 0 & \ldots & 0 \\ . & . & . & . & . \\ -1 & 0 & 0 & \ldots & 1 \end{bmatrix} \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^D \\ 1 & x_2 & x_2^2 & \ldots & x_2^D \\ . & . & . & . & . \\ 1 & x_n & x_n^2 & \ldots & x_n^D \end{bmatrix}$$

$$\det(\mathbb{F}') = \det(\mathbb{A}) \det(\mathbb{F}) = \det(\mathbb{F})$$

(c) **(OPTIONAL)** Perform the following sequence of operations to $\mathbb{F}'$, and obtain the matrix $\mathbb{F}''$.
i) Subtract $x_1 * \mathsf{column}_{n-1}$ from column_n.
ii) Subtract $x_1 * \mathsf{column}_{n-2}$ from column_{n-1}.
$\vdots$
n-1) Subtract $x_1 * \mathsf{column}_1$ from column_2.
Write out the matrix $\mathbb{F}''$ and argue why $\det(\mathbb{F}') = \det(\mathbb{F}'')$**.**

$$\mathbb{F}'' = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & x_2 - x_1 & x_2(x_2 - x_1) & \dots & x_2^{D-1}(x_2 - x_1) \\ . & . & . & . & . \\ 0 & x_n - x_1 & x_n(x_n - x_1) & \dots & x_n^{D-1}(x_n - x_1) \end{bmatrix}$$

$$\mathbb{F}'' = \begin{bmatrix} 1 & 0 & \dots & 0 \\ -1 & 1 & \dots & 0 \\ . & . & . & . \\ -1 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^D \\ 1 & x_2 & x_2^2 & \dots & x_2^D \\ . & . & . & . & . \\ 1 & x_n & x_n^2 & \dots & x_n^D \end{bmatrix} \begin{bmatrix} 1 & 0 & \dots & 0 \\ . & . & . & . \\ 0 & \dots & 1 & -x_1 \\ 0 & 0 & \dots & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & -x_1 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ . & . & . & . & . \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

All the new matrix multiplied to $\mathbb{F}'$ has only one element off-diagonal that means their determinants are all 1. Therefore, $\det(\mathbb{F}') = \det(\mathbb{F}'')$.

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

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

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

$$|\lambda_B \mathbb{I} - \mathbb{B}| = \begin{vmatrix} \lambda_B - 1 & \vec{0}^\top \\ \vec{0} & \lambda_B \mathbb{I} - \mathbb{A} \end{vmatrix} = (\lambda_B - 1)(\lambda_B \mathbb{I} - \mathbb{A}) = 0,$$

Therefore, $\lambda_B = 1$ is an eigenvalue of $\mathbb{B}$ and the other eigenvalues are the same as that of $\mathbb{A}$. No we conclude $\det(\mathbb{B}) = \det(\mathbb{A})$ because determinant is the product of all eigenvalues.

(e) **(OPTIONAL) Use the above parts and an induction argument to prove that** $\det(\mathbb{F}) = \prod_{1 \le i < j \le n}(x_j - x_i)$**.** Consequently, **argue** that the matrix $\mathbb{F}$ is full rank unless two input data points are equal.

Hint: First show that

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

where $D = n - 1$.

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.)

We have proved three things:
1.Subtract a row from another preserve the determinant
2.Subtract a column times a scalar from another preserve the determinant
3.We can convert a rectangle matrix into a square matrix and preserve the determinant

Let's do the following sequence of operation to $\mathbb{F}$

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

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

$\vdots$

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

We have something like:

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & x_2 - x_1 & x_2(x_2 - x_1) & \dots & x_2^{D-1}(x_2 - x_1) \\ . & . & . & . & . \\ 1 & x_n - x_1 & x_n(x_n - x_1) & \dots & x_n^{D-1}(x_n - x_1) \end{bmatrix}$$

Next we do the following sequence of operation:

i) Subtract row_1 from row_n.

ii) Subtract row_1 from row_{n-1}.

$\vdots$

n-1) Subtract row_1 from row_2.

Now we have a form like that in (d):

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & x_2 - x_1 & x_2(x_2 - x_1) & \dots & x_2^{D-1}(x_2 - x_1) \\ . & . & . & . & . \\ 0 & x_n - x_1 & x_n(x_n - x_1) & \dots & x_n^{D-1}(x_n - x_1) \end{bmatrix}$$

Here we use the hint to retrieve all $x_i - x_1$, then we get the formula we want to prove.

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

We keep doing this until we get to a 2×2 matrix as in (a) and we will finally get $\det(\mathbb{F}) = \prod_{1 \le i < j \le n} (x_j - x_i)$.
If $x_j \neq x_i$ for any $i \neq j$, $\det(\mathbb{F}) \neq 0$, therefore $\mathbb{F}$ is full rank. If $\exists i, j so x_i = x_j$, then $x_i - x_j = 0$, $\det(\mathbb{F}) = 0$, F cannot be full rank.

(f) We now consider multivariate polynomials. We have $\vec{x}_i = (x_{i,1}, x_{i,2}, \dots, x_{i,\ell})^T \in \mathbb{R}^\ell$ and we consider the multivariate polynomial $\vec{p}_D(\vec{x})$ of degree D. Here is an illustration of the new features for $\ell = 2$ and $D = 3$:

$$\vec{x}_i = (x_{i,1}, x_{i,2})^\top \quad \text{and} \quad \vec{p}_D(\vec{x}_i) = (1,\ x_{i,1},\ x_{i,2},\ x_{i,1}^2,\ x_{i,2}^2,\ x_{i,1}x_{i,2},\ x_{i,1}x_{i,2}^2,\ x_{i,1}^2x_{i,2},\ x_{i,1}^3,\ x_{i,2}^3)^\top.$$

For a more general ℓ and D, **show that the size of $\vec{p}_D(\vec{x})$ is $\binom{D+\ell}{\ell}$**. You may use a stars and bars argument (link is if you did not take CS70).

For L variables, d degrees polynomial, the power of all the terms can be converted to d.
For example, $x_1^2 = x_1^2 * 1^{d-2}$
Not all features appear in every single term so some features might be omitted and their powers are zero.
Rewrite the above term $x_1^2 = x_1^2 * x_2^0 * x_3^0 * x_4^0 * ... * 1^{d-2}$
The question is: how do we distribute these d degrees to our L features?
This becomes the following problem:
We have d " o ", and we want to insert L " X " into this sequence of " o ". Each " o " represents 1 degree, each " X " represents a multiplication sign.
For example, $x_1^2 = x_1^2 * x_2^0 * x_3^0 * x_4^0 * ... * 1^{d-2}$, there are L multiplication signs and d degrees in total
ooXXXXX.....XXXooo....ooo
$x_1^2 x_2^3 = x_1^2 * x_2^3 * x_3^0 * x_4^0 * ... * 1^{d-5}$
ooXoooXX.....XXXooo....ooo
There first two o represents the degree of the first term and the last one represents that of 1.
When I insert the first " X ", there are d+1 possibilities;
When I insert the second " X ", there are d+2 possibilities;
...
When I insert the L-th " X ", there are L+d possibilities.
We have (L+d)(L+d-1)...(d+1) possibilities in total.
Because the order we insert " X " doesn't matter, we have to divide it by the permutation of L, L(L-1)....1
Therefore, we have
$\frac{(L+d)(L+d-1)...(d+1)}{L(L-1)....1} = \binom{L+d}{L}$

(g) With n sample points $\{\vec{x}_i\}_{i=1}^n$, stack up the multivariate polynomial features $\vec{p}_D(\vec{x}_i)$ as rows to obtain the feature matrix $\mathbb{F}_\ell \in \mathbb{R}^{n \times \binom{D+\ell}{\ell}}$. Let $x_{i,1} = x_{i,2} = \cdots = x_{i,\ell} = \alpha_i$ where α_i's are distinct scalars for $i \in \{1, 2, 3 \ldots, n\}$. **Show that with these sample points, the feature matrix $\mathbb{F}_\ell$ always has linearly dependent columns for any value of $n > 1$.** Compare this fact with your conclusion from part (e).

As before, we only need to show the case when $n = \binom{D+\ell}{\ell}$. Then we find the first degree columns. Because $x_{i,1} = x_{i,2} = \cdots = x_{i,\ell} = \alpha_i$, these columns are always the same. This result is different from part(e) because we have cross terms and duplicated polynomial terms.

(h) Now **design a set of sampling points $\vec{x}_i$ such that the corresponding multivariate polynomial feature matrix $\mathbb{F}_\ell$ is full column rank.** You are free to choose the number of points at your convenience. 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 $\mathbb{F}_\ell$ 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.

I will play with the factors of my sample points. Let $\vec{x_i}$ be a vector of distinct prime numbers. For example, $\vec{x_i} = [2, 3, 5, \dots]$ I never reuse these prime numbers for different $\vec{x_i}$. For example, a degree 2 polynomial of two variables is something like:

$$\begin{bmatrix} 1 & 2 & 3 & 2\times 3 & 2^2 & 3^2 \\ 1 & 5 & 7 & 5\times 7 & 5^2 & 6^2 \\ 1 & 11 & 13 & 11\times 13 & 11^2 & 13^2 \\ 1 & 17 & 19 & 17\times 19 & 17^2 & 19^2 \\ 1 & 23 & 29 & 23\times 29 & 23^2 & 29^2 \\ 1 & 31 & 37 & 31\times 37 & 31^2 & 37^2 \end{bmatrix}$$

It's pretty clear that this is a full rank matrix. However, I don't know how to prove for arbitrary D and l Therefore I turned into what we have proved above and think about converting the problem into a polynomial without cross terms. I achieved that by choosing a base α_i for each sample and rewrite $\vec{x_i} = [1, x_{i,1}, x_{i,2}, \dots]$ to be like $\vec{x_i} = [1, x_{i,1}, x_{i,1}^2, \dots, x_{i,1}^D, x_{i,2}, x_{i,2}^2, \dots]$ Now I choose $x_{i,1} = \alpha_i$, it becomes $\vec{x_i} = [1, \alpha_i, \alpha_i^2, \dots, \alpha_i^D, x_{i,2}, x_{i,2}^2, \dots]$, Now I choose $x_{i,2} = \alpha_i^{D+1}$. Keep doing this, $x_{i,3} = \alpha_i^{D(D+1)+1} \dots$. We notice that cross terms will fall into the gaps between two exponents. For example, $x_{i,2} * x_{i,1}, x_{i,2} * x_{i,1}^2, \dots, x_{i,2} * x_{i,1}^{n-1} = \alpha_i^{D+2}, \alpha_i^{D+3}, \dots, \alpha_i^{2D-1}$ fall between $x_{i,2} = \alpha_i^{D+1}$ and $x_{i,2}^2 = \alpha_i^{2D+2}$.It's guaranteed that the exponents of these terms are all different. Particularly, we need a sample size of n where n is greater than the largest exponent we have given our degree D and number of variables l. I would take $n = (D + l)^{D+l}$ just to be safe.Therefore, as long as α_i are different for each i, we have proved that the matrix we get is full rank.

Question 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

$$f(x) = \sum_{i=0}^{m} \frac{1}{i!} f^{(i)}(x_0)(x - x_0)^i + \frac{1}{(m+1)!} f^{(m+1)}(a(x))(x - x_0)^{m+1}. \tag{3}$$

Here, $f^{(m)}$ denotes the mth derivative of the function f, and $a(x)$ is some value between x_0 and x. By definition, $f^{(0)} = f$.

The last term $r_m(x) := \frac{1}{(m+1)!} f^{(m+1)}(a(x))(x - x_0)^{m+1}$ of this expansion is tyically referred to as the remainder term when approximating $f(x)$ by an m-th degree polynomial.

We denote by ϕ_m the m-th degree Taylor polynomial (also called the Taylor *approximation*), which consists of the Taylor series expansion of order m without the remainder term and thus reads

$$f(x) \approx \phi_m(x) = \sum_{i=0}^{m} \frac{1}{i!} f^{(i)}(x_0)(x - x_0)^i$$

where the sign $\approx$ indicates approximation of the left hand side by the right hand side.

For functions f whose derivatives are bounded in the neighborhood of interest, if we have $|f^{(m)}(x)| \leq T$ for $x \in (x_0 - s, x_0 + s)$, we know that for $x \in (x_0 - s, x_0 + s)$ that the *approximation error* of the m-th order Taylor approximation $|f(x) - \phi_m(x)| = |r_m(x)|$ is upper bounded $|f(x) - \phi_m(x)| \leq \frac{T|x - x_0|^{m+1}}{(m+1)!}$.

(a) **Compute the 1st, 2nd, 3rd, and 4th order Taylor approximation of the following functions around the point $x_0 = 0$.**

i) e^x

ii) $\sin x$

i) $e^x \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24}$

i) $\sin x \approx 0 + x + 0\frac{x^2}{2} - \frac{x^3}{6} + 0\frac{x^4}{24}$

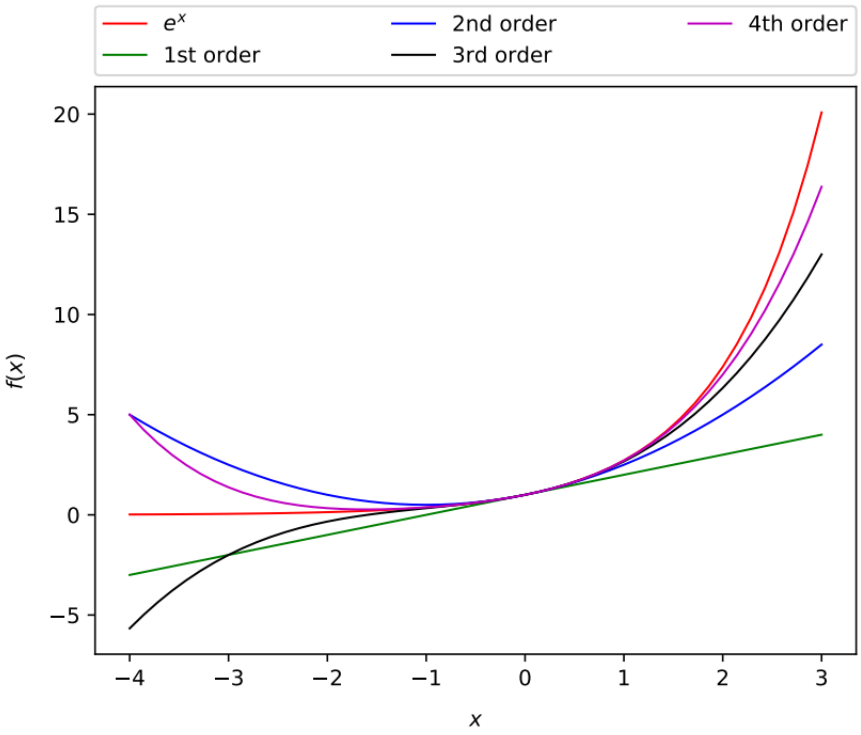
You can find 1st-4th order by picking the corresponding terms.

(b) For $f(x) = e^x$, plot the Taylor approximation from order 1 through 4 at $x_0 = 0$ for x in the domain $I := [-4, 3]$. If you are unfamiliar with plotting in Python, please refer to the starter code for this question which could save you time.

We denote the maximum approximation error on the domain I by $\|f - \phi_m\|_\infty := \sup_{x \in I} |f(x) - \phi_m(x)|$, where $\|\cdot\|_\infty$ is also called the sup-norm with respect to I. Compute $\|f - \phi_m\|_\infty$ for $m = 2$. What is an upper bound for arbitrary non-zero integers m? Compute the limit of $\|f - \phi_m\|_\infty$ as $m \to \infty$?

Hint: Use Stirling's approximation for integers n which is: $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

Now plot the Taylor approximation of f up to order 4 on the interval $[-20, 8]$. How does the approximation error behave outside the bounded interval I?



$|f - \phi_2| = |e^x - 1 - x - \frac{x^2}{2}|$. **There is a global minima at** $x = 0$. **We just need to compare** $x = -4$ **and** $x = -3$. **We get** $\|f - \phi_2\|_\infty = -\frac{17}{2} + e^3$.

To compute the limit, we use first have to now the bound of the derivate in the domain I. **It's** $|f^{(m)}(x)| \leq e^3$. **Therefore we know that**

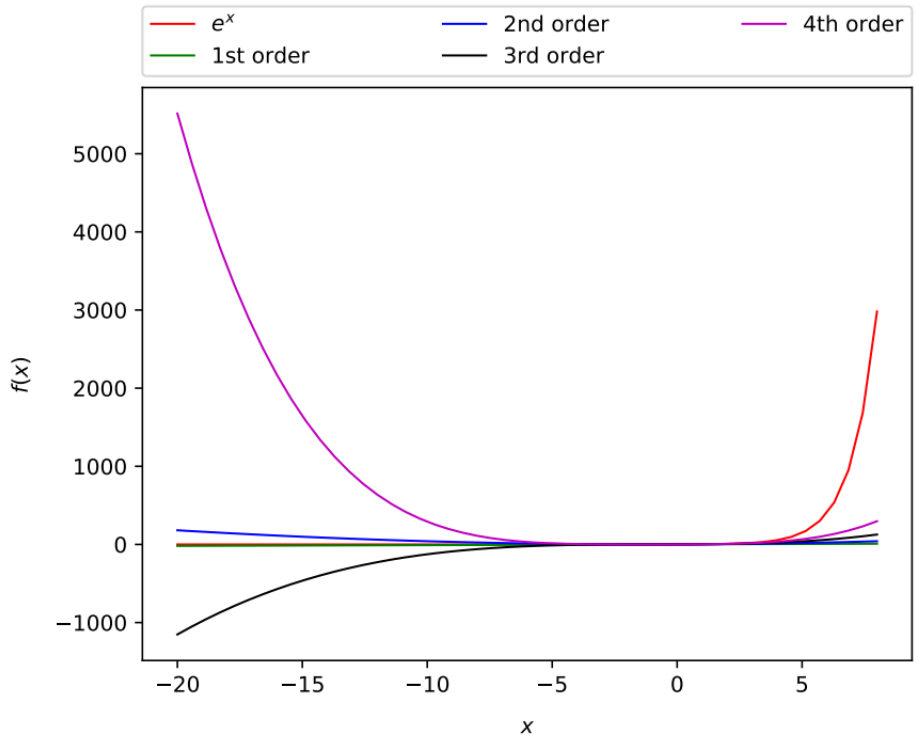
$$|f(x) - \phi_m(x)| \leq \frac{e^3|x|^{m+1}}{(m+1)!} \approx |\frac{e^{m+4}|x|^{m+1}}{\sqrt{2\pi(m+1)}(m+1)^{m+1}}|$$

We could substitute the limit of $|x| \leq 4$ **into the equation, and get**

$$|f(x) - \phi_m(x)| \leq |\frac{e^3}{\sqrt{2\pi(m+1)}}(\frac{4e}{m+1})^{m+1}|$$

Both parts will approach zero when m gets close to zero. Therefore $\lim_{m\to\infty} \|f - \phi_m\|_\infty = 0$

For the same order taylor approximation, the approximation error increases outside the bounded interval I.




```
import numpy as np
import matplotlib.pyplot as plt
import os
import math

plot_col = ['r', 'g', 'b', 'k', 'm']
plot_mark = ['o', '^', 'v', 'D', 'x', '+']

# Plots the rows in 'ymat' on the y-axis vs. 'xvec' on the x-axis
# with labels 'ylabels'
# and saves figure as pdf to 'dirname/filename'
def plotmatnsave(ymat, xvec, ylabels, dirname, filename):
no_lines = len(ymat)
fig = plt.figure(0)

if len(ylabels) > 1:
for i in range(no_lines):
xs = np.array(xvec)
ys = np.array(ymat[i])
plt.plot(xs, ys, color = plot_col[i % len(plot_col)], lw=1, label=ylabels[i])

lgd = plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), loc=3, ncol=3, mode="expand", borderaxespad=0.)

savepath = os.path.join(dirname, filename)
plt.xlabel('$x$', labelpad=10)
plt.ylabel('$f(x)$', labelpad=10)
plt.savefig(savepath, bbox_extra_artists=(lgd,), bbox_inches='tight')
plt.close()

# Sets the labels
labels = ['$e^x$', '1st order', '2nd order', '3rd order', '4th order']

# TODO: Given x values in "x_vec", save the respective function values e^x,
# and its first to fourth degree Taylor approximations
# as rows in the matrix "y_mat"
x_vec = np.linspace(-4, 3)
```

```
y_mat = np.vstack([np.exp(x_vec), np.ones(x_vec.size) + x_vec,
np.ones(x_vec.size) + x_vec + x_vec * x_vec / 2,
np.ones(x_vec.size) + x_vec + x_vec * x_vec / 2 + x_vec * x_vec * x_vec / 6,
np.ones(x_vec.size) + x_vec + x_vec * x_vec / 2 + x_vec * x_vec * x_vec / 6 + x_vec * x_vec * x_vec *
    x_vec / 24])

# Define filename, invoke plotmatnsave
filename = 'approx_plot.pdf'
plotmatnsave(y_mat, x_vec, labels, '.', filename)

# and its first to fourth degree Taylor approximations
# as rows in the matrix "y_mat"
x_vec = np.linspace(-20, 8)
y_mat = np.vstack([np.exp(x_vec), np.ones(x_vec.size) + x_vec,
np.ones(x_vec.size) + x_vec + x_vec * x_vec / 2,
np.ones(x_vec.size) + x_vec + x_vec * x_vec / 2 + x_vec * x_vec * x_vec / 6,
np.ones(x_vec.size) + x_vec + x_vec * x_vec / 2 + x_vec * x_vec * x_vec / 6 + x_vec * x_vec * x_vec *
    x_vec / 24])

# Define filename, invoke plotmatnsave
filename = 'approx_plot2.pdf'
plotmatnsave(y_mat, x_vec, labels, '.', filename)
```

(c) Let's say we would like an accurate polynomial approximation of the functions in part (a) for all $x \in I$. Given the results of the previous parts, we can in fact find a Taylor polynomial of degree D such that $|f(x) - \phi_D(x)| \leq \epsilon$ for all $x \in I$. Using the upper bound in (b), show that if D is larger than $O(\log(1/\epsilon))$, we can guarantee that $|f(x) - \phi_D(x)| \leq \epsilon$ for both choices of $f(x)$ in part (a). Note that constant factors are not relevant here, and assume sufficiently small positive $\epsilon \ll 1$.

We have solved the upper bound of $|f(x) - \phi_D(x)|$ above, which gives us:

$$\frac{T|x|^{D+1}e^{D+1}}{\sqrt{2\pi(D+1)}(D+1)^{D+1}}$$

Take the log and Simplify it gives us $\alpha D + \beta - (D+1)\ln(D+1)$, where $\alpha, \beta,$ are some constant numbers and it's dominant by the last term when D gets large.

$$ln(|f(x) - \phi_D(x)|) \leq -(D+1)\ln(D+1) < -D$$

Because $D \geq \log(1/\epsilon)$, the above formula $\ln(|f(x) - \phi_D(x)|) < -\ln(1/\epsilon)$. Therefore $|f(x) - \phi_D(x)| \leq \epsilon$ We notice that the choice of T doesn't really affect our conclusion here so it's true for both choices of $f(x)$

(d) Conclude that a univariate polynomial of high enough degree can approximate any function f on a closed interval I, that is continuously differentiable infinitely many times and has bounded derivatives $|f^{(m)}(x)| \leq T$ for all $m \geq 1$ and $x \in I$. Mathematically speaking, we need to show that for any $\epsilon > 0$, there exists a degree $D \geq 1$ such that $\|f - \phi_D\|_\infty < \epsilon$, where the sup-norm is taken with respect to the interval I.

This universal approximation property illustrates the power of polynomial features, even when we don't know the underlying function f that is generating our data! Later, we will see that neural networks are also universal function approximators.)

I have already shown that in part(c). The only difference is that although T is a constant, we don't know the range of its value anymore. Therefore we need to take into account T and find a new D. Rewrite the above formula, but keep T in it:

$$ln(|f(x) - \phi_D(x)|) \leq \ln T - (D+1)\ln(D+1) < \ln T - D < \ln(\epsilon)$$

Therefore, we have $D > \ln(T/\epsilon)$. That is, as long as we choose a D that's greater than $\ln(T/\epsilon)$, we are safe to say $|f(x) - \phi_D(x)| < \epsilon$. In this case, I will be crazy, so I will take $e^{e^{e^{T/\epsilon}}}$. It looks really cool, isn't it?

(e) 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

$$\begin{aligned} f(x,y) = f(x_0,y_0) + f_x(x_0,y_0)(x-x_0) + f_y(x_0,y_0)(y-y_0) + \\ \frac{1}{2!}[f_{xx}(x_0,y_0)(x-x_0)^2 + f_{xy}(x_0,y_0)(x-x_0)(y-y_0) + \\ f_{yx}(x_0,y_0)(x-x_0)(y-y_0) + f_{yy}(x_0,y_0)(y-y_0)^2] + \dots \end{aligned} \tag{4}$$

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 $f(\vec{v}) = f(x, y)$ with $\vec{v} = [x, y]^T$ and $\vec{v}_0 = [x_0, y_0]$. Clearly define any additional vectors and matrices that you use.

Consider the multivariate function $f(\vec{v}) = e^x y^2$ where $\vec{v} = [x, y]^T$. Please write down the second order multivariate Taylor approximation for f at $\vec{v}_0$.

$$f(\vec{v}) = f(\vec{v}_0) + \nabla f(\vec{v}_0)^\top (\vec{v} - \vec{v}_0) + \frac{1}{2!}(\vec{v} - \vec{v}_0)^\top \nabla^2 f(\vec{v}_0)^\top (\vec{v} - \vec{v}_0) + \dots$$

$$\nabla f(\vec{v}_0) = \begin{bmatrix} f_x(x_0, y_0) \\ f_y(x_0, y_0) \end{bmatrix} \qquad \nabla^2 f(\vec{v}_0) = \begin{bmatrix} f_{xx}(x_0, y_0) & f_{xy}(x_0, y_0) \\ f_{yx}(x_0, y_0) & f_{yy}(x_0, y_0) \end{bmatrix}$$

if $f(\vec{v}) = e^x y^2$, we just use the formula above:

$$\nabla f(\vec{v}_0) = \begin{bmatrix} e^{x_0} y_0^2 \\ 2e^{x_0} y_0 \end{bmatrix} \qquad \nabla^2 f(\vec{v}_0) = \begin{bmatrix} e^{x_0} y_0^2 & 2e^{x_0} y_0 \\ 2e^{x_0} y_0 & 2e^{x_0} \end{bmatrix}$$

I don't think it's necessary to write it out.

(f) In this part we want to show how the univariate approximation discussed in the previous parts can be used as a stepping stone to understand polynomial approximation in multiple dimensions.

Let us consider the approximation of the function $f(\vec{v}) = e^x y^2$ around $\vec{v}_0 = \vec{0}$ along a direction $\vec{v} - \vec{v}_0 = \vec{v}$. All vectors along this direction are on the path $\vec{v}(t) := \vec{0} + t(\vec{v} - \vec{0})$ for $t \in \mathbb{R}$. Write the second order Taylor expansion of $g(t) = f(\vec{v}(t))$ around the point $t_0 = 0$. Note that g is a function mapping a scalar to a scalar.

By considering all such paths $\vec{v}(t)$ over different directions $\vec{v}$, we can reduce the multi-dimensional setting to the univariate setting. The example hopefully helped you to get an idea why the approximation behavior of Taylor polynomials holds similarly in higher dimensions.

$$\nabla f(\vec{v}_0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \nabla^2 f(\vec{v}_0) = \begin{bmatrix} 0 & 0 \\ 0 & 2 \end{bmatrix}$$

We use the taylor series above and get

$$f(\vec{t}) = (ty)^2 + \dots$$
$$f(\vec{t}) = y^2 t^2 + \dots$$

Question 5. Jaina and her giant peaches

Make sure to submit the code you write in this problem to "HW2 Code" on Gradescope.

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 these 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

$$y_i \approx w_0 + w_1 x_i + w_2 x_i^2 \cdots + w_D x_i^D$$

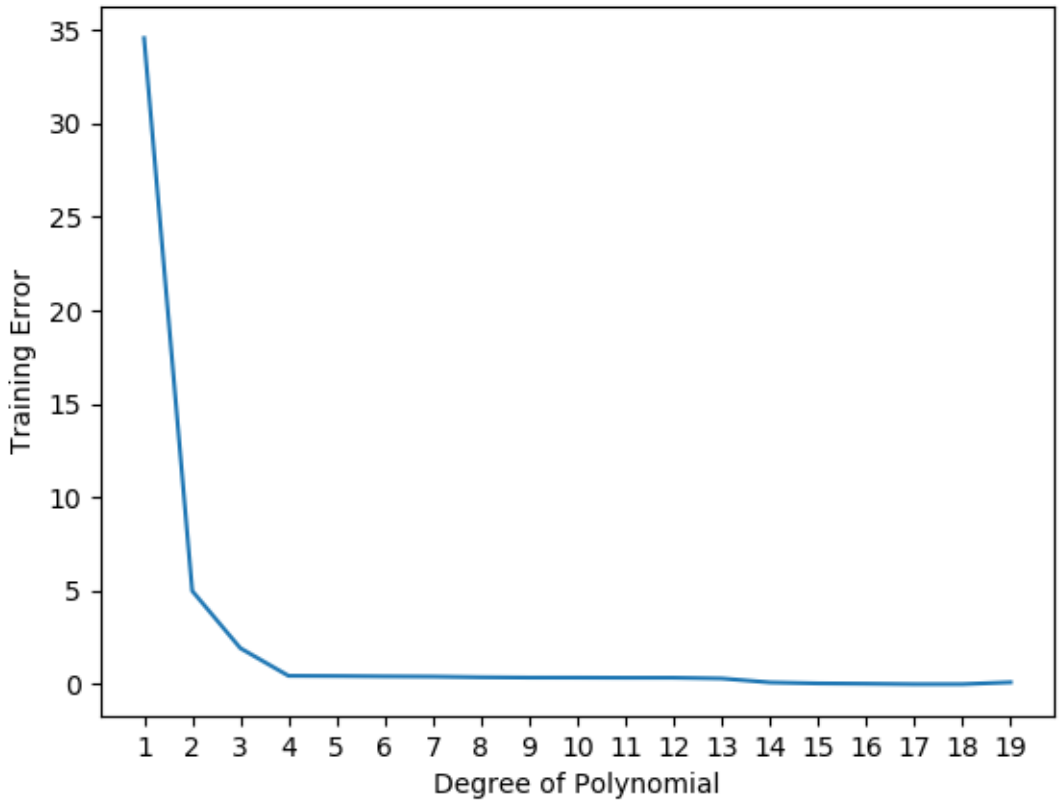
where her goal is to fit a polynomial of degree D to this data. Include all text responses and plots in your write-up.

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

We can convert the problem into a optimization problem that find $\hat{w} = \min_{\hat{w}} \|Xw - y\|_2^2$, where

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^D \\ 1 & x_2 & x_2^2 & \ldots & x_2^D \\ . & . & . & . & . \\ 1 & x_n & x_n^2 & \ldots & x_n^D \end{bmatrix}, w = \begin{bmatrix} w_0 \\ w_1 \\ . \\ w_D \end{bmatrix}, y = \begin{bmatrix} y_1 \\ y_2 \\ . \\ y_n \end{bmatrix}$$

(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` with the masses centered and normalized to lie in the range $[-1, 1]$. Write a script 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 $D \in \{1, 2, 3, \ldots, n-1\}$. You may not use any library other than `numpy` and `numpy.linalg` for computation.




```
#!/usr/bin/env python3

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


# There is numpy.linalg.lstsq, whicn you should use outside of this classs
def lstsq(A, b):
return np.linalg.solve(A.T @ A, A.T @ b)


def main():
data = spio.loadmat('1D_poly.mat', squeeze_me=True)
x_train = np.array(data['x_train'])
y_train = np.array(data['y_train']).T

n = 20 # max degree
err = np.zeros(n - 1)
# fill in err
# YOUR CODE HERE
x_poly = np.ones((1, n))
for i in range(1, n):
x_poly = np.vstack((x_poly, np.power(x_train, i)))
w = lstsq(x_poly.T, y_train)
err[i - 1] = np.mean((x_poly.T.dot(w) - y_train) ** 2)

plt.plot(np.linspace(1, n - 1, n - 1), err)
plt.xlabel('Degree of Polynomial')
plt.ylabel('Training Error')
plt.xticks(np.linspace(1, n - 1, n - 1))
```

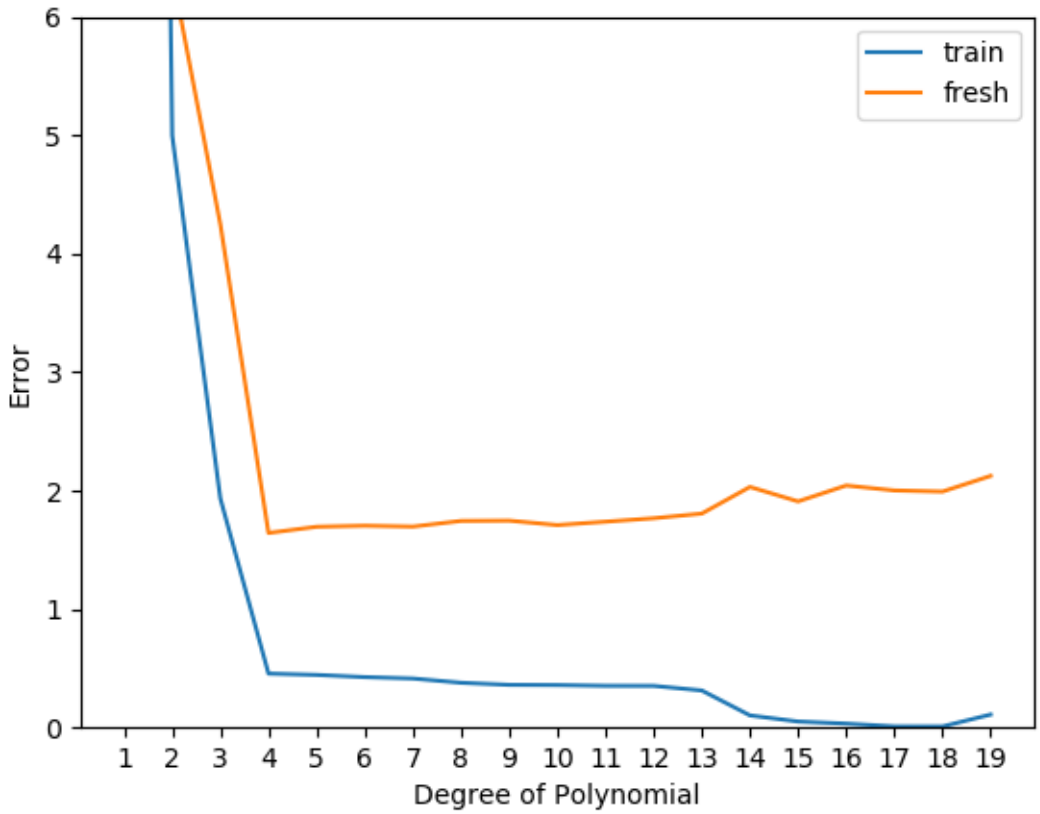
```
    plt.show()


if __name__ == "__main__":
main()
```

(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?

The average training error decreases as a function of D. This is because we have more features and degrees of freedom that we can always overfit.

If I try to fit a polynomial of degree n with a standard matrix inversion method, I will fail because it's not invertible. For a polynomial of degree n, we know $rank(X) \leq n$ and $rank(X^\top X) \leq n$. However, $X^\top X \in \mathbb{R}^{(n+1)\times(n+1)}$. Therefore, $X^\top X$ is not full rank and not invertible.

(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 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?



1. This plot has two curves and the plot in (b) has only one.
2. We plot test error here too.
3. It has a different range for y,
4. The test error is not strictly decreasing. This is because we overfit for high degrees of polynomial. The true model might a low degree of polynomial so adding higher order term might not help us.

```
#!/usr/bin/env python3

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


# There is numpy.linalg.lstsq, whicn you should use outside of this classs
def lstsq(A, b):
return np.linalg.solve(A.T @ A, A.T @ b)


def main():
data = spio.loadmat('1D_poly.mat', squeeze_me=True)
x_train = np.array(data['x_train'])
y_train = np.array(data['y_train']).T
y_fresh = np.array(data['y_fresh']).T

n = 20 # max degree
err_train = np.zeros(n - 1)
err_fresh = np.zeros(n - 1)
# fill in err_fresh and err_train
# YOUR CODE HERE
```

```
x_poly = np.ones((1, n))

for i in range(1, n):
x_poly = np.vstack((x_poly, np.power(x_train, i)))
w = lstsq(x_poly.T, y_train)
err_train[i - 1] = np.mean((x_poly.T.dot(w) - y_train) ** 2)
err_fresh[i - 1] = np.mean((x_poly.T.dot(w) - y_fresh) ** 2)

plt.figure()
plt.ylim([0, 6])
plt.plot(np.linspace(1, n-1, n-1), err_train, label='train')
plt.plot(np.linspace(1, n-1, n-1), err_fresh, label='fresh')
plt.legend()
plt.xlabel('Degree of Polynomial')
plt.ylabel('Error')
plt.xticks(np.linspace(1, n-1, n-1))
plt.show()


if __name__ == "__main__":
main()
```

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

I would propose looking at the minimal test error.It appears that a polynomial of degree 4 because the test error reaches the minim at 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. A D-multivariate polynomial function looks like

$$f_D(\vec{x}) = \sum_j \alpha_j \prod_i x_i^{p_{ji}},$$

where $\forall j : \sum_i p_{ji} \leq D$. Here α_j is the scale constant for jth term and p_{ji} is the exponent of x_i in jth term. 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 $D \in \{1, 2, 3, 4, 5, 6\}$ 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. You are not allowed to use any library other than `numpy` and `numpy.linalg`.

Average test error: [0.0585605836 0.05854458001 0.05854454239 0.05854454143 0.05854454144 0.05854454144] It looks like $D = 4$ is the best one.

(g) Now redo the previous part, but use 4-fold cross-validation on all combinations of $D \in \{1, 2, 3, 4, 5, 6\}$ 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. Print the average training/validation error per sample for all D and λ.

Average train error:

$$\begin{bmatrix} [0.058560578820.05856058360.058560597930.058560621760.05856065507] \\ [0.05845630410.05845905850.058464435340.058470372560.05847614587] \\ [0.058376096130.058456605330.058463194260.058469541960.05847552173] \\ [0.058187067520.058456587390.058463184160.058469534060.05847551477] \\ [0.057929081610.058456586990.058463183950.058469533920.05847551466] \\ [0.057439124370.058456586980.058463183950.058469533920.05847551466] \end{bmatrix}$$

Average valid error:

$$\begin{bmatrix} [0.058581691780.058581684360.05858168650.058581698180.05858171935] \\ [0.058554827600.058544580010.058540716830.058539741230.05854014400] \\ [0.058629583760.058544542390.05854069380.058539725180.05854013182] \\ [0.058805171770.058544541430.058540693120.058539724290.05854013065] \\ [0.059085211640.058544541440.058540693120.058539724290.05854013065] \\ [0.059622590800.058544541440.058540693120.058539724290.05854013065] \end{bmatrix}$$

Best degree of polynomial: 4
Best lambda value: 0.15

```
import time

import numpy as np
import scipy.io as spio

data = spio.loadmat('polynomial_regression_samples.mat', squeeze_me=True)
# data = spio.loadmat('matlab.mat', squeeze_me=True)
data_x = data['x']
data_y = data['y']
Kc = 4 # 4-fold cross validation
KD = 6 # max D = 6
LAMBDA = [0, 0.05, 0.1, 0.15, 0.2]

# my global variables
sample_size = data_y.size / Kc
elongated_data_x = np.vstack([data_x, data_x])
elongated_data_y = np.hstack([data_y, data_y])


def ridge(A, b, lambda_):
# Make sure data are centralized
return np.linalg.solve(A.T.dot(A) + lambda_ * np.eye(A.shape[1]), A.T.dot(b))
```

```
def get_error(X, w, y):
return np.mean((X.dot(w) - y) ** 2)


def fit(D, lambda_):
# YOUR CODE TO COMPUTE THE AVERAGE ERROR PER SAMPLE
errors = np.asarray([0.0, 0.0])
for iv in range(0, Kc):
validation_x = elongated_data_x[int(iv * sample_size): int((iv + 1) * sample_size), :]
train_x = elongated_data_x[int((iv + 1) * sample_size):
int((iv + Kc) * sample_size), :]
validation_y = elongated_data_y[int(iv * sample_size): int((iv + 1) * sample_size)]
train_y = elongated_data_y[int((iv + 1) * sample_size):
int((iv + Kc) * sample_size)]
poly_train_x = polynomial(train_x, D)
w = ridge(poly_train_x, train_y, lambda_)
errors[0] += get_error(poly_train_x, w, train_y)
errors[1] += get_error(polynomial(validation_x, D), w, validation_y)
errors = errors / Kc
return errors


def polynomial(poly_data, degree):
if poly_data.ndim != 2:
pass
nvar = poly_data.shape[1]
npoints = poly_data.shape[0]
results = np.ones((npoints, comb(degree + nvar, degree)))
if degree == 0:
return results
start = 0
counts = np.zeros(nvar, dtype=int)
cur_index = 1
cur_len = 1
for d in range(1, degree + 1):
last_len = cur_len
cur_start = 0
cur_len = 0
for i in range(0, nvar):
for j in range(start + cur_start, start + last_len):
results[:, cur_index] = poly_data[:, i] * results[:, j]
cur_index += 1
temp = counts[i]
counts[i] = last_len - cur_start
cur_start += temp
cur_len += counts[i]
start = start + last_len

return results


def comb(n, k):
result = 1
for i in range(k):
result *= (n - i)
for i in range(k):
result /= (k - i)
return int(np.round(result))


def main():
np.set_printoptions(precision=11)
```

```
Etrain = np.zeros((KD, len(LAMBDA)))
Evalid = np.zeros((KD, len(LAMBDA)))
for D in range(KD):
print(D)
for i in range(len(LAMBDA)):
Etrain[D, i], Evalid[D, i] = fit(D + 1, LAMBDA[i])

print('Average train error:', Etrain, sep='\n')
print('Average valid error:', Evalid, sep='\n')

# YOUR CODE to find best D and i
minTestError = float('inf')
minD = 0
minLambda = 0
for D in range(KD):
for i in range(len(LAMBDA)):
if Evalid[D, i] < minTestError:
minD = D + 1
minLambda = LAMBDA[i]
minTestError = Evalid[D, i]
print('Best degree of polynomial:', minD, sep='\n')
print('Best lambda value:', minLambda, sep='\n')


if __name__ == "__main__":
main()
```

Question 6. Nonlinear Classification Boundaries

Make sure to submit the code you write in this problem to "HW2 Code" on Gradescope.

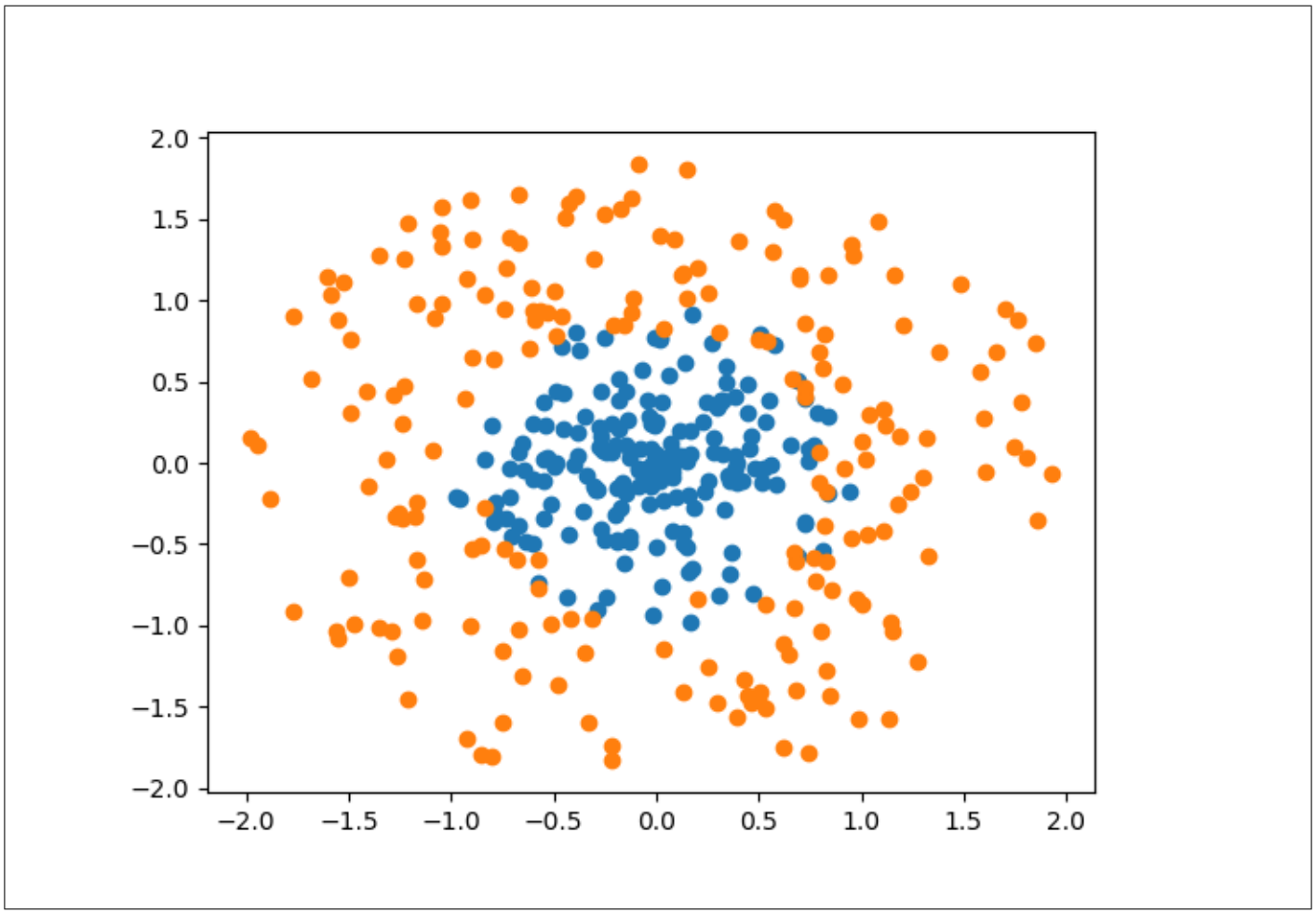
In this problem we will learn how to use polynomial features to learn nonlinear classification boundaries.

In Problem 7 on HW1, we found that linear regression can be quite effective for classification. We applied it in the setting where the training data points were *approximately linearly separable*. This means there exists a hyperplane such that most of the training data points in the first class are on one side of the hyperplane and most training data points in the second class are on the other side of the hyperplane.

However, often times in practice classification datasets are not linearly separable. In this case we can create features that are linearly separable by augmenting the original data with polynomial features as seen in Problem 5. This embeds the data points into a higher dimensional space where they are more likely to be linearly separable.

In this problem we consider a simple dataset of points $(x_i, y_i) \in \mathbb{R}^2$, each associated with a label b_i which is -1 or $+1$. The dataset was generated by sampling data points with label -1 from a disk of radius 1.0 and data points with label $+1$ from a ring with inner radius 0.8 and outer radius 2.0.

(a) **(OPTIONAL) Run the starter code to load and visualize the dataset and submit a scatterplot of the points with your homework. Why can't these points be classified with a linear classification boundary?**



(b) **(OPTIONAL)** Classify the points with the technique from Problem 7 on HW1 ("A Simple classification approach"). Use the feature matrix $\mathbb{X}$ whose first column consists of the x-coordinates of the training points and whose second column consists of the y-coordinates of the training points. The target vector $\vec{b}$ consists of the class label -1 or $+1$. Perform the linear regression $\vec{w}_1 = \arg\min_{\vec{w}} \left\| \mathbb{X}\vec{w} - \vec{b} \right\|_2^2$. **Report the classification accuracy on the test set.**

OLS: 203 out of 400 points (50.75%) are correctly classified

(c) **(OPTIONAL)** Now augment the data matrix $\mathbb{X}$ with polynomial features $1, x^2, xy, y^2$ and classify the points again, i.e. create a new feature matrix

$$\Phi = \begin{pmatrix} x_1 & y_1 & x_1^2 & x_1 y_1 & y_1^2 & 1 \\ x_2 & x_2 & x_2^2 & x_2 y_2 & y_2^2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \\ x_n & y_n & x_n^2 & x_n y_n & y_n^2 & 1 \end{pmatrix}$$

and perform the linear regression $\vec{w}_2 = \arg\min_{\vec{w}} \left\| \Phi\vec{w} - \vec{b} \right\|_2^2$. **Report the classification accuracy on the test set.**

PolyFit: 356 out of 400 points (89.00%) are correctly classified

(d) **(OPTIONAL) Report the weight vector that was found in the feature space with the polynomial features. Show that up to small error the classification rule has the form $\alpha x^2 + \alpha y^2 \leq \beta$. What is the interpretation of β/α here? Why did the classification in the augmented space work?**

$$0.083x + 0.047y + 0.654x^2 + -0.036xy + 0.716y^2 + -0.815 = 0$$

We can take $\alpha = 0.685$, and $\beta = 0.815$. Other coefficients do not have the same order of magnitude as these two numbers. (10x smaller).

$\sqrt{\beta/\alpha}$ is the radius of the circle that best separates two sets of data points.

Because we added new polynomial term and converted a nonlinear classifier in lower dimensional space into a linear classifier in the higher dimensional space.

```
import numpy as np
import matplotlib.pyplot as plt

Xtrain = np.load("Xtrain.npy")
ytrain = np.load("ytrain.npy")
threshold_label = 0
```

```
def visualize_dataset(X, y):
plt.scatter(X[y < 0.0, 0], X[y < 0.0, 1])
plt.scatter(X[y > 0.0, 0], X[y > 0.0, 1])
plt.show()


# visualize the dataset:
visualize_dataset(Xtrain, ytrain)

# TODO: solve the linear regression on the training data
w = np.linalg.lstsq(Xtrain, ytrain)[0]

Xtest = np.load("Xtest.npy")
ytest = np.load("ytest.npy")


# TODO: report the classification accuracy on the test set
def getNcorrect(X, w, y, threshold):
y_predicted = X.dot(w)
ncorrect = np.count_nonzero(
np.logical_or(
(y_predicted > threshold) == (y > threshold),
(y_predicted <= threshold) == (y <= threshold)
)
)
return ncorrect


correct_point = getNcorrect(Xtest, w, ytest, threshold_label)
print("(6b) OLS: %d out of %d points (%.2f%%) are correctly classified" % (
correct_point, ytest.size, correct_point / ytest.size * 100))

# TODO: Create a matrix Phi_train with polynomial features from the training data
# and solve the linear regression on the training data

Xtrain_poly = np.vstack([
Xtrain[:, 0],
Xtrain[:, 1],
Xtrain[:, 0] ** 2,
Xtrain[:, 0] * Xtrain[:, 1],
Xtrain[:, 1] ** 2,
np.ones(ytrain.size),
]).T

Xtest_poly = np.vstack([
Xtest[:, 0],
Xtest[:, 1],
Xtest[:, 0] ** 2,
Xtest[:, 0] * Xtest[:, 1],
Xtest[:, 1] ** 2,
np.ones(ytest.size),
]).T

w_poly = np.linalg.lstsq(Xtrain_poly, ytrain)[0]

# TODO: Create a matrix Phi_test with polynomial features from the test data
# and report the classification accuracy on the test set

correct_point = getNcorrect(Xtest_poly, w_poly, ytest, threshold_label)
print("(6c) PolyFit: %d out of %d points (%.2f%%) are correctly classified" % (
correct_point, ytest.size, correct_point / ytest.size * 100))
```

```
a, b, c, d, e, f= np.squeeze(w_poly)
print("(6d) {:.3f} x + {:.3f} y + {:.3f} x^2 + {:.3f} xy + {:.3f} y^2 + {:.3f} = 0".format(a, b, c, d, e,
    f))
```

Question 7. Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the 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.

Why don't we penalize the bias term in Ridge Regression?

I actually found an answer on stackexchange (`https://stats.stackexchange.com/questions/86991/reason-for-not-shrinking-the-bias-intercept-term-in-regression`).

Basically, there are two reasons mentioned:

1. A shift of c in all y_i, will not only shift the weights of bias
2. The average of $\hat{y}$ will be different from the average of y

I will prove them here:

$$\left(\begin{bmatrix} X^\top \\ \vec{1}^\top \end{bmatrix} \begin{bmatrix} X & \vec{1} \end{bmatrix} + \begin{bmatrix} \lambda\mathbb{I} & \vec{0} \\ \vec{0}^\top & \beta \end{bmatrix}\right) \begin{bmatrix} w \\ w_0 \end{bmatrix} = \begin{bmatrix} X^\top \\ \vec{1}^\top \end{bmatrix} y$$

$$\begin{bmatrix} X^\top X + \lambda\mathbb{I} & X^\top \vec{1} \\ \vec{1}^\top X & n+\beta \end{bmatrix} \begin{bmatrix} w \\ w_0 \end{bmatrix} = \begin{bmatrix} X^\top y \\ \vec{1}^\top y \end{bmatrix}$$

$$\begin{bmatrix} (X^\top X + \lambda\mathbb{I})w + X^\top \vec{1} w_0 \\ \vec{1}^\top X w + (n+\beta) w_0 \end{bmatrix} = \begin{bmatrix} X^\top y \\ \vec{1}^\top y \end{bmatrix}$$

If we shift y by c, from the second term we have

$$\vec{1}^\top X w + (n+\beta) w_0 = \vec{1}^\top (y + \vec{c})$$

$$\vec{1}^\top X w + (n+\beta) w_0 = \sum_{i=1}^{n} y_i + nc$$

if $\beta = 0$, we have

$$\vec{1}^\top X w + n(w_0 - c) = \sum_{i=1}^{n} y_i$$

Double check that $\hat{w} = w_0 - c$ also satisfies the first equation. A shift of c in y will only affect the weight of the bias term. When $\beta \neq 0$, we have:

$$\vec{1}^\top X w + (n+\beta)(w_0 - \frac{n}{n+\beta} c) = \sum_{i=1}^{n} y_i$$

However, this is not the solution to the first equation, where

$$(X^\top X + \lambda\mathbb{I})w + X^\top \vec{1} w_0 = X^\top (y + \vec{c})$$

$$(X^\top X + \lambda \mathbb{I})w + X^\top \vec{1}(w_0 - c) = X^\top (y)$$

You can see that two solutions are not the same meaning, after penalizing the bias, a shift of c in y must also affects other weights. We also notice that the left hand side of the second equation is very close to the sum of all predicted values $\hat{y}$, where

$$\sum_{i=1}^{n} \hat{y}_i = \vec{1}^\top \hat{y} = \vec{1}^\top \begin{bmatrix} X & \vec{1} \end{bmatrix} \begin{bmatrix} w \\ w_0 \end{bmatrix} = \vec{1}^\top X w + n w_0$$

Therefore, we get to the other point, the average of $\hat{y}$ will be different from the average of y iff. $\beta = 0$ or $w_0 = 0$. w_0 means there is no bias term and $\sum_{i=1}^{n} y_i = 0$.