CS 189 Introduction to Machine Learning
Spring 2018 HW2

Your self-grade URL is http://eecs189.org/self_grade?question_ids=1_1,1_2,2_1,2_2,2_3,2_4,2_5,2_6,2_7,2_8,2_9,3_1,3_2,3_3,3_4,3_5,3_6,3_7,3_8,4_1,4_2,4_3,4_4,4_5,4_6,5_1,5_2,5_3,5_4,5_5,5_6,5_7,6_1,6_2,6_3,6_4,7.

This homework is due **Friday, February 2 at 10 p.m.**.

## 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 $\mathbf{X} \in \mathbb{R}^{n \times d}$ and a vector $\mathbf{y} \in \mathbb{R}^n$, define the optimization problem

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

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.**

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

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

*Formal derivation:* Given a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ and a vector $\mathbf{y} \in \mathbb{R}^n$, define the optimization problem

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

In the first step we show that the above problem is equivalent (i.e. it has the same solution) as the following one:

$$\min_{\mathbf{w}} \max_{\lambda \geq 0} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \cdot \underbrace{(\|\mathbf{w}\|_2^2 - \beta^2)}_{\Delta(\mathbf{w})} \tag{3}$$

We see this as follows:

$$\max_{\lambda \geq 0} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \cdot \Delta(\mathbf{w}) = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \begin{cases} \infty & \text{if } \Delta(\mathbf{w}) > 0 \\ 0 & \text{if } \Delta(\mathbf{w}) \leq 0. \end{cases}$$

In the last step we used that if $\Delta(\mathbf{w}) > 0$, the term $\lambda\Delta(\mathbf{w})$ can be made arbitrarily large by making $\lambda$ large. On the other hand if $\Delta(\mathbf{w}) \leq 0$, the maximization problem is solved by $\lambda = 0$.

It turns out that something called strong duality[1] holds here, which means we can exchange the minimization and maximization in (3) without changing the resulting optimal objective function. We get

$$\max_{\lambda \geq 0} \min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \cdot (\|\mathbf{w}\|_2^2 - \beta^2). \tag{4}$$

For every $\beta$, the maximum will be attained at some $\lambda^*(\beta)$ and this establishes the equivalence of the constrained problem (2) with the unconstrained problem

$$\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda^*(\beta) \cdot (\|\mathbf{w}\|_2^2 - \beta^2). \tag{5}$$

i.e. There is a $\lambda^*(\beta)$ so that the problem has the same resulting $\mathbf{w}$ solution as if we had solved the original constrained optimization problem.

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

$$\min_{w} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \nu\|\mathbf{w}\|_2^2. \tag{6}$$

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

**Solution:** Intuitively, $\beta$ is the radius of the ball in which the solution $\mathbf{w}^*$ of the constrained optimization problem (1) is sought and $\nu$ is the weight that is put on making $\|\mathbf{w}\|^2$ small. Therefore, if we increase $\nu$, this corresponds to a stronger regularization of $\mathbf{w}$ to zero, which corresponds to a smaller $\beta$. In the limit $\nu \to 0$, no regularization takes place which corresponds to $\beta \to \infty$. Similarly, in the case $\nu \to \infty$, we force $\mathbf{w} = 0$, which corresponds to $\beta = 0$ in the constrained setting.

*Formal derivation:* Again to solve this formally, we need some tools from convex optimization like Lagrangians and KKT conditions that you are strongly recommended to get familiar with if they are not clear to you from taking EECS127 or similar[2]. Remember, machine

[1] An understanding of duality is one of the key concepts provided by studying optimization. In general without strong duality, there can be a gap between what are called the primal and dual problems. Strong duality is when there is no gap. A full treatment of duality is beyond the scope of 189/289A, but is referenced here so that students can begin to understand why such understanding can be relevant. The example being studied here is almost paradigmatic in its strong duality.

[2] see `http://www.eecs189.org/static/notes/n18.pdf` to read up on some of this material, but there is no substitute for taking a good optimization course.

learning is the application of optimization to problems formulated using the language of linear algebra and probability. There really is no avoiding optimization if you are serious.

The Lagrangian corresponding to the constrained optimization problem is

$$\mathscr{L}(\mathbf{w},\lambda) = \|\mathbf{y}-\mathbf{X}\mathbf{w}\|_2^2 + \lambda(\|\mathbf{w}\|_2^2 - \beta).$$

Writing down the KKT conditions on optimality, we get

$$\nabla_{\mathbf{w}}\mathscr{L}(\mathbf{w}^*,\lambda^*) = 0$$
$$\lambda^*(\|\mathbf{w}^*\|_2^2 - \beta) = 0.$$

If $\mathbf{w}^*(\nu)$ is a solution of (6), it satisfies the KKT conditions with $\beta = \|\mathbf{w}^*(\nu)\|_2^2$ and $\lambda^* = \nu$. Therefore, the constrained and the unconstrained problem are equivalent if and only if $\beta = \|\mathbf{w}^*(\nu)\|_2^2$ where $\|\mathbf{w}^*(\nu)\|_2^2$ is the solution of $\|\mathbf{w}^*(\nu)\|_2^2$.

Now how does $\beta$ behave if $\nu$ increases? From $\mathbf{w}^*(\nu) = (\mathbf{X}^\top\mathbf{X} + \nu\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$ and the singular value decomposition $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ we get

$$\|\mathbf{w}^*(\nu)\|_2^2 = \left\|(\Sigma^2 + \nu\mathbf{I})^{-1}\Sigma\mathbf{U}^\top\mathbf{y}\right\|_2^2$$

and therefore $\beta$ is monotonically decreasing for $\nu$ — in agreement with our intuition above.

(c) One reason why we might want to have small weights $\mathbf{w}$ has to do with the sensitivity of the predictor to its input. Let $\mathbf{x}$ be a $d$-dimensional list of features corresponding to a new test point. Our predictor is $\mathbf{w}^\top\mathbf{x}$. **What is an upper bound on how much our prediction could change if we added noise $\varepsilon \in \mathbb{R}^d$ to a test point's features $\mathbf{x}$?**

**Solution:** The change in prediction is given by

$$|\mathbf{w}^\top(\mathbf{x}+\varepsilon) - \mathbf{w}^\top\mathbf{x}| = |\mathbf{w}^\top\varepsilon| \le \|\mathbf{w}\|_2\|\varepsilon\|_2 \tag{7}$$

where the second step is a result of the Cauchy-Schwarz inequality. To understand intuition as to why this is true, you can divide both sides by $\|\mathbf{w}\|_2\|\varepsilon\|_2$ which gives you

$$\left|\langle \frac{\mathbf{w}}{\|\mathbf{w}\|_2}, \frac{\varepsilon}{\|\varepsilon\|_2}\rangle\right| \le 1 \tag{8}$$

In words, this inequality is saying that if you project a unit vector $\frac{\varepsilon}{\|\varepsilon\|_2}$ onto some direction $\frac{\mathbf{w}}{\|\mathbf{w}\|_2}$, the resulting length must be less than or equal to 1. The prediction will thus vary from the original prediction within a distance of at most $\|\mathbf{w}\|_2\|\varepsilon\|_2$.

(d) **Derive that the solution to ridge regression** (6) **is given by $\hat{\mathbf{w}}_\mathbf{r} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$. 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

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

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

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

Thus, the solution is given by

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

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

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

**Solution:** The eigenvalues of $\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I}$ are given by $\sigma_1^2+\lambda, \sigma_2^2+\lambda, \cdots, \sigma_d^2+\lambda$. In order to see this, note that if $v_i$ is an eigenvector of $\mathbf{X}^\top\mathbf{X}$ with eigenvalue $\sigma_i^2$, then we have

$$(\mathbf{X}^\top\mathbf{X}+\lambda\mathbf{I})v_i = \sigma_i^2 v_i + \lambda v_i = (\sigma_i^2+\lambda)v_i.$$

Notice that $\sigma_i^2 \geq 0$ since the matrix $\mathbf{X}^\top\mathbf{X}$ 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 $\lambda$, and thus improve the inversion operation.

A careful student would notice that there is an issue with the simple explanation here. "Small" relative to what? The answer to this (take a numerical linear algebra course to understand this more deeply) is that we consider small eigenvalues relative to the largest eigenvalue — it is the dynamic range of the scale of eigenvalues that matter. This is what is effectively captured in something called the condition number of the matrix. The operation of adding $\lambda$ changes greatly the relative scale of eigenvalues that are smaller than $\lambda$ and does almost nothing (in a relative sense) to eigenvalues that are much bigger than $\lambda$. So regularization improves the numerical conditioning of the problem.

(f) Let the number of parameters $d = 3$ and the number of datapoints $n = 5$, and let the eigenvalues of $\mathbf{X}^\top\mathbf{X}$ 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 0.001 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.

(g) Another advantage of ridge regression can be seen for under-determined systems. Say we have the data drawn from a $d = 5$ parameter model, but only have $n = 4$ training samples of it, i.e. $\mathbf{X} \in \mathbb{R}^{4\times 5}$. 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 consider $\mathbf{w} = \mathbf{w}_0 + \mathbf{w}^*$ where $\mathbf{w}_0$ is in the null space of $\mathbf{X}$ and $\mathbf{w}^*$ is a solution.

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

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

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

For ordinary least squares, let us assume that $\mathbf{w}'$ minimizes $\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2$, i.e., $\|\mathbf{X}\mathbf{w}' - \mathbf{y}\|_2 \leq \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2$ for all $\mathbf{w} \in \mathbb{R}^d$.

Since $d > n$, the matrix $\mathbf{X}$ has a non-trivial nullspace. We can take any vector $\mathbf{w_0}$ in the nullspace of $\mathbf{X}$ and consider the new vector $\mathbf{w}''(\alpha) = \mathbf{w}' + \alpha \mathbf{w_0}$.

Notice that $\|\mathbf{X}\mathbf{w}''(\alpha) - \mathbf{y}\|_2 = \|\mathbf{X}\mathbf{w}' - \mathbf{y}\|_2 \leq \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2$ for all $\mathbf{w} \in \mathbb{R}^d$ because $\mathbf{w_0}$ is in the null space of $\mathbf{X}$. Thus, the vector $\mathbf{w}''(\alpha)$ is a minimizer for any choice of $\alpha$. We have thus shown that the least squares problem has an infinite number of solutions.

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

**Solution:**

Plugging the singular value decomposition $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^\top$ into the solution of the ridge regression, we get

$$\begin{aligned}\mathbf{w}^*(\lambda) &= (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y} \\ &= \mathbf{V}(\Sigma^\top \Sigma + \lambda \mathbf{I})^{-1} \Sigma^\top \mathbf{U}^\top \mathbf{y}\end{aligned}$$

and for $\lambda \to 0$ this converges to the unique solution $\mathbf{w}^* = \mathbf{V}\Sigma^{-1}\mathbf{U}^\top \mathbf{y}$, also called the *minimum norm solution*. (Notice here that $\Sigma^{-1}$ has the obvious definition of just involving the relevant square matrix of nonzero singular values.)

For those who remember from EE16B, this is also closely connected to the idea of the Moore-Penrose Psuedo-inverse. This problem continues the trend of illustrating the utility of the SVD while reasoning about such problems. It really is a workhorse of machine learning reasoning.

(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

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

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

**Solution:** The objective is

$$\begin{aligned} f(w) &= \frac{1}{2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \|\Gamma \mathbf{w}\|_2^2 \\ &= \mathbf{w}^\top (\frac{1}{2}\mathbf{X}^\top \mathbf{X} + \lambda \Gamma^\top \Gamma)\mathbf{w} - \mathbf{y}^\top \mathbf{X} w + \frac{1}{2}\mathbf{y}^\top \mathbf{y}. \end{aligned}$$

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

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

Thus, the solution is now given by $w = (\mathbf{X}^\top \mathbf{X} + 2\lambda \Gamma^\top \Gamma)^{-1}\mathbf{X}^\top y$, and one can again verify that this is a minimizer by showing that the Hessian is positive definite since the matrix $\Gamma$ has full rank, implying that $\Gamma^\top \Gamma$ is itself positive definite.

# 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 $\mathbf{F}$ where

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

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

Parts (a)–(e) study the rank of the feature matrix $\mathbf{F}$ as a function of the points $x_i$'s. These parts are elementary and therefore bonus problems.

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

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

**Solution:**

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

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

(b) **(BONUS)** From parts (b) through (e), we work through different steps to establish that the columns of $\mathbf{F}$ 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 $\mathbf{F}$. Now as a first step, construct a matrix $\mathbf{F}'$ from the matrix $\mathbf{F}$ via this operation: subtract the first row of $\mathbf{F}$ from its rows 2 through $n$. **Is it true that** $\det(\mathbf{F}) = \det(\mathbf{F}')$**?**

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

**Solution:** Notice that $\mathbf{F}' = \mathbf{E}\mathbf{F}$, where $\mathbf{E}$ is an $n \times n$ lower triangular matrix with the form

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

where missing entries denote zeros. Also, the eigenvalues of a triangular matrix are given by its diagonal entries, and so the determinant of $\mathbf{E}$ is 1. Furthermore, since $\det(\mathbf{F}') = \det(\mathbf{E})\det(\mathbf{F})$, we have $\det(\mathbf{F}') = \det(\mathbf{F})$.

(c) **(BONUS)** Perform the following sequence of operations to $\mathbf{F}'$, and obtain the matrix $\mathbf{F}''$.

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

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

$\vdots$

n-1) Subtract $x_1 * \mathsf{column}_1$ from $\mathsf{column}_2$.

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

**Solution:** We now have $\mathbf{F}'' = \mathbf{F}'\mathbf{E}$, where $\mathbf{E}$ is an $n \times n$ upper triangular matrix taking the form

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

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

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

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

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

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

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

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

Hint: First show that

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

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.)

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

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

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

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

where

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

Now using the previous part and noticing that the lower right sub-matrix is given by

$$\mathbf{F}^{(n-1)} = \begin{bmatrix} 1 & x_2 & \ldots & x_2^{n-2} \\ 1 & x_3 & \ldots & x_3^{n-2} \\ 1 & \vdots & \ldots & \\ 1 & x_n & \ldots & x_n^{n-2} \end{bmatrix} = \begin{bmatrix} \mathbf{p}_{D-1}(x_2)^T \\ \mathbf{p}_{D-1}(x_3)^T \\ \vdots \\ \mathbf{p}_{D-1}(x_n)^T, \end{bmatrix}$$

where the superscript is used to denote that the matrix size is reduced to $n-1$. Note that $D-1=n-2$. For convenience, we also denote $\mathbf{F}$ by $\mathbf{F}^{(n)}$ to denote that it has size $n$. Using part (d), we obtain that $\det(\mathbf{F}''') = \det(\mathbf{F}^{(n-1)})$. Note that the matrix $\mathbf{F}^{(n-1)}$ has the same monomial form as $\mathbf{F}^{(n)}$. but with dimension $n-1$ instead of $n$. Performing the operations as above by using the sample $x_2$, we have

$$\begin{aligned}
\det(\mathbf{F}^{(n)}) = \det(\mathbf{F}) &= \left(\prod_{i=2}^{n}(x_i - x_1)\right)\left(\prod_{j=3}^{n}(x_j - x_2)\right)\det\left(\begin{bmatrix} 1 & x_3 & \dots & x_3^{n-3} \\ 1 & \vdots & \dots & \\ 1 & x_n & \dots & x_n^{n-3} \end{bmatrix}\right) \\
&= \left(\prod_{i=2}^{n}(x_i - x_1)\right)\left(\prod_{j=3}^{n}(x_j - x_2)\right)\det\left(\begin{bmatrix} \mathbf{p}_{D-2}(x_3)^T \\ \vdots \\ \mathbf{p}_{D-2}(x_n)^T, \end{bmatrix}\right) \\
&= \left(\prod_{i=2}^{n}(x_i - x_1)\right)\left(\prod_{j=3}^{n}(x_j - x_2)\right)\det(\mathbf{F}^{(n-2)})
\end{aligned}$$

Unrolling each of these determinants then yields the required answer

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

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

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

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

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

**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 $D_0 + D_1 + \dots + D_\ell = D$. 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 $\ell$ 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 $i$th bars.

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

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

**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 $i$th entry is given by $\alpha_i^2$. The columns are thus linearly dependent no matter how many samples we take.

Thus we see that although all the points are distinct, the feature matrix is not full rank. This is different from the univariate case, where the $x_i$'s being distinct was necessary and sufficient for the feature matrix $\mathbf{F}$ to have full column rank.

(h) Now **design a set of sampling points $\mathbf{x}_i$ such that the corresponding multivariate polynomial feature matrix $\mathbf{F}_\ell$ 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 $\mathbf{F}_\ell$ 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.

**Solution:** We describe two possible solutions.

The first possible 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 $j = 1, 2, \ldots, \ell$. Now, we can verify that every distinct monomial $x_{i,1}^{D_1} x_{i,2}^{D_2} \ldots x_{i,\ell}^{D_\ell} = (\alpha_i)^{D_1 + D_2(D+1)^1 + \cdots + D_\ell (D+1)^{\ell-1}}$. Thus, every choice of the tuple $(D_0, D_1, \ldots, D_\ell)$ results in a different power of $\alpha_i$. In order to see this, note that $D_\ell D_{\ell-1} \ldots D_0$ can be thought of as a $D$-ary number. (Think of $D = 9$.)

Evaluating $\mathbf{F}_\ell$ 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 $D^\ell$ scales like $\binom{D+\ell}{\ell}$ when $D$ grows.)

The second possible solution:

In the previous solution, we made the rows of $F$ looks like univariate polynomials evaluated at distinct points. In this case, we make the columns look like univariate polynomials evaluated at distinct points. Let $x_{i,j} = p_j^i$, where $p_j$ is the $j^{\text{th}}$ prime number. Consider a monomial term in row $i$ of the matrix $x_{i,1}^{D_1} x_{i,2}^{D_2} \ldots x_{i,\ell}^{D_\ell}$:

$$x_{i,1}^{D_1} x_{i,2}^{D_2} \ldots x_{i,\ell}^{D_\ell} = x_{1,1}^{iD_1} x_{1,2}^{iD_2} \ldots x_{1,\ell}^{iD_\ell} = (x_{1,1}^{D_1} x_{1,2}^{D_2} \ldots x_{1,\ell}^{D_\ell})^i.$$

This monomial is the $i^{\text{th}}$ power the same monomial in row 1, so columns are univariate polynomials. Furthermore, by the fundamental theorem of arithmetic,

$$x_{1,1}^{D_1} x_{1,2}^{D_2} \ldots x_{1,\ell}^{D_\ell}$$

will give distinct values for each distinct choice of $\{D_1, \ldots, D_\ell\}$. Thus, the univariate polynomials are evaluated at distinct points, resulting in a full rank for this matrix.

# 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

$$f(x) = \sum_{i=0}^{m} \frac{1}{i!} f^{(i)}(x_0)(x-x_0)^i + \frac{1}{(m+1)!} f^{(m+1)}(a(x))(x-x_0)^{m+1}. \tag{9}$$

Here, $f^{(m)}$ denotes the $m$th 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 $\phi_m$ 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 $I$ around $x_0$ of interest, if we have $|f^{(m)}(x)| \leq T$ for $x \in I$, we know that for $x \in I$ 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) $\sin x$

**Solution:**

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

$$\begin{aligned}
\phi_1(x) &\approx 1 + (x-0)^1 \\
\phi_2(x) &\approx 1 + (x-0)^1 + \frac{1}{2}(x-0)^2 \\
\phi_3(x) &\approx 1 + (x-0)^1 + \frac{1}{2}(x-0)^2 + \frac{1}{6}(x-0)^3 \\
\phi_4(x) &\approx 1 + (x-0)^1 + \frac{1}{2}(x-0)^2 + \frac{1}{6}(x-0)^3 + \frac{1}{24}1(x-0)^4
\end{aligned}$$

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 a 4-th order approximation (and the lower order approximations can be read off immediately).

$$\phi_4(x) \approx 0 + (x-0)^1 + \frac{1}{2}0(x-0)^2 + \frac{1}{6}(-1)(x-0)^3 + \frac{1}{24}0(x-0)^4$$

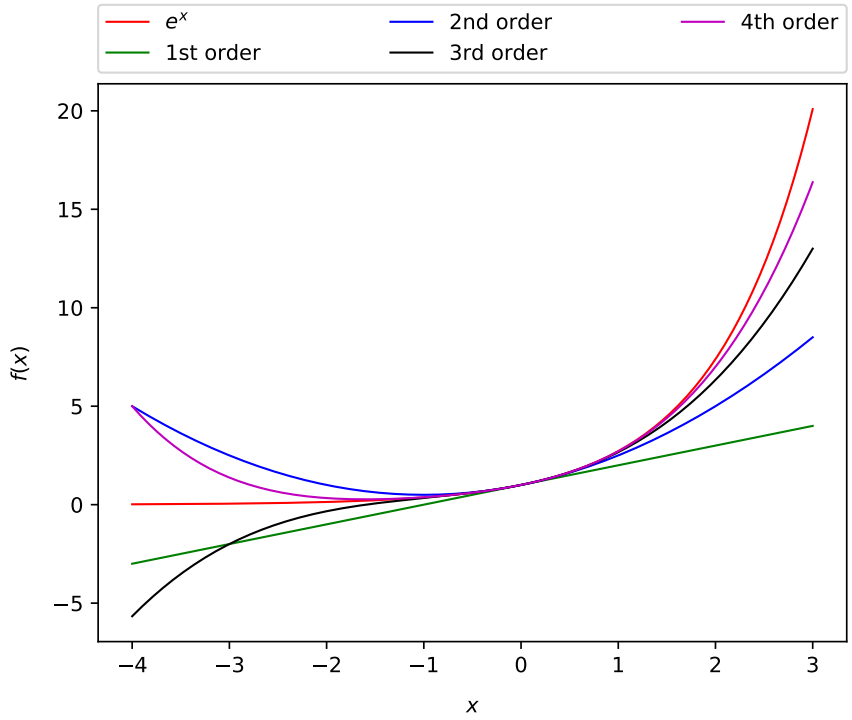
(b) For $f(x) = e^x$, **plot the Taylor approximation from order 1 through 4 at $x_0 = 0$ for $x$ in the domain $I := [-4, 3]$.** If you are unfamiliar with plotting in Python, please refer to the starter code for this question which could save you time.

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 $m = 2$. 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 $I$?**

**Solution:** The following plot shows the original function and its approximations




```
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', '+']

    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()

    # Get arrays
    x_vec = np.linspace(-4,3,1000)
    y_mat = np.zeros((5,1000))

    # Actual function and approximations applied on x
    y_mat[0,:] = list(map(lambda x: np.exp(x), x_vec))
    y_mat[1,:] = list(map(lambda x: 1+x, x_vec))
    y_mat[2,:] = list(map(lambda x: 1+x+0.5*x**2, x_vec))
    y_mat[3,:] = list(map(lambda x: 1+x+0.5*x**2+1/6*x**3, x_vec))
    y_mat[4,:] = list(map(lambda x: 1+x+0.5*x**2+1/6*x**3+1/24*x**4,
    ↪ x_vec))

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

    # Plot and save
    filename = 'approx_plot.pdf'
    plotmatnsave(y_mat, x_vec, labels, '.', filename)
```

We know that $r_2(x) = 0$ at $x = 0$ by definition of $\phi_2(x) = 0$, therefore $x = 0$ is a minimum. Note that

$$|r_2(x)| = \begin{cases} e^x - (1 + x + \frac{1}{2}x^2) & \text{for } x \in [0,3] \\ 1 + x + \frac{1}{2}x^2 - e^x & \text{for } x \in [-4,0] \end{cases}$$

We look at the intervals $I_1 = [0,3]$ and $I_2 = [-4,0]$ separately. (i) For $x \in I_1$, we have that $|r_2(x)|$ is monotonically increasing with $x$, since the derivative reads $e^x - x - 1$ which is positive for all $x > 0$ (think about why). Therefore, the maximum value is at the boundary $x = 3$ with $r_2(3) = 11.5855$. (ii) For $x \in I_2$, we see that $|r_2(x)| = -r_2(x)$ is monotically

decreasing and thus the maximum is again at the boundary $x = -4$ with $-r_2(-4) = 4.9817$. Using the fact that at $x = 0$, $r_2(x) = -r_2(x)$ and comparing the two boundary values reveals that

$$\|f - \phi_2\|_\infty = \sup_{x \in I = I_1 \cup I_2} |r_2(x)| = r_2(3) = 11.5855$$

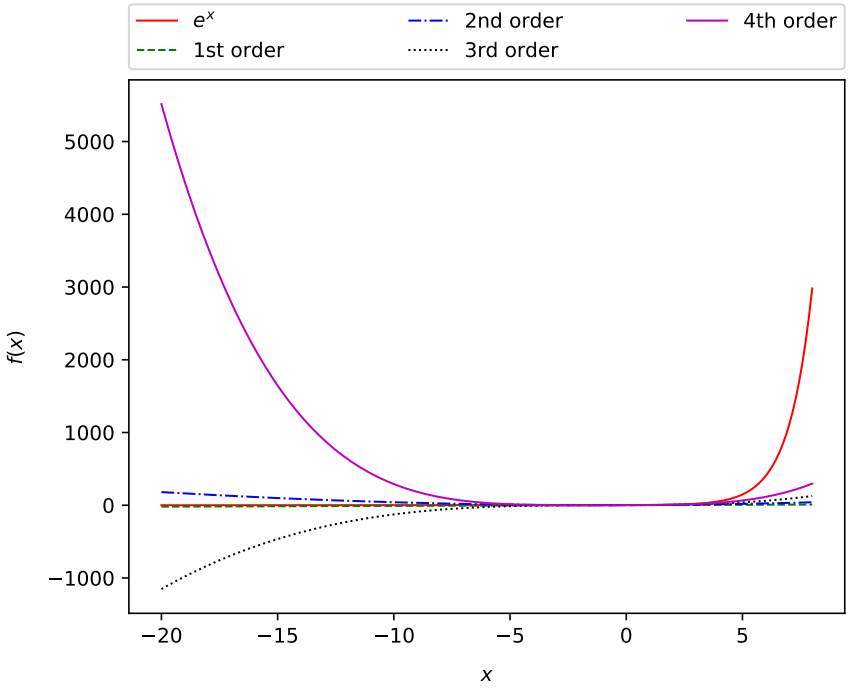
As in the description, in order to upper bound for $|r_m(x)|$ for all $x \in I$, we need to find an upper bound on the absolute value of the derivative in the interval, that is $\sup_{x \in I} |f^{(m)}(x)| = \sup_{x \in I} |e^x| = e^3$. Therefore, $\|f - \phi_m\|_\infty \leq \frac{e^3 4^{m+1}}{(m+1)!}$.

Using Stirling's formula, we obtain that

$$\frac{e^3 4^{m+1}}{(m+1)!} \approx \frac{e^3}{\sqrt{2\pi(m+1)}} \left(\frac{4e}{m+1}\right)^{m+1}$$

Both terms individually converge to zero (for the second term since there exists $m$ such that for all $N \geq m$ the term $\frac{4e}{m+1} \leq 1$) and thus their product also converges to zero.

The following figure demonstrates that although the 4th order approximation is reasonably close to the original function within the bounded interval $I = [-4, 3]$, the maximum approximation error quickly becomes very large for $x \notin I$




Mathematically speaking, given a fixed interval $I$ even though the uniform approximation error (taking the supremum inside the interval $I$) is bounded by some $\varepsilon$, this does not guarantee anything about the error for $x \notin I$. Although for any interval $[-a, a]$, you may find an $D(\varepsilon, a)$ for an arbitrary $\varepsilon$ such that $\sup_{x \in [-a,a]} |f(x) - \phi_{D(\varepsilon,a)}| < \varepsilon$, this degree $D(\varepsilon, a)$ is potentially

different for all $a$. One way to see it is that the upper bound of the approximation error really reads

$$\frac{e^a}{\sqrt{2\pi(m+1)}}\left(\frac{a\cdot e}{m+1}\right)^{m+1}$$

as a function of $a$, i.e. the larger $a$, the larger of an $m$ we might need. Note here that the actual error $e^a\frac{a^{m+1}}{(m+1)!}$ is not in fact mononotically decreasing with $m$ across all integers but eventually does for large $m$. This is the reason the error at $x=-10$ in the plot above actually increases with the degree.

(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 $|f(x)-\phi_D(x)| \leq \varepsilon$ for all $x \in I$. **What is an upper bound of $\|f-\phi_m\|_\infty$ for arbitrary non-zero integers $m$? Using this upper bound, show that if $D$ is larger than $O(\log(1/\varepsilon))$, we can guarantee that $\|f-\phi_D\|_\infty \leq \varepsilon$ for both choices of $f(x)$ in part (a).** Note that constant factors are not relevant here, and assume sufficiently small positive $\varepsilon \ll 1$.

**Solution:** Note: The upper bound question, originally asked for in question (b), is moved here because its relevance really enters here. You may give full credit to question (b) if you managed to find an upper bound which allowed you to show (c).

Using the argument and approximation from (b), it suffices to require

$$u(D) := \frac{e^3}{\sqrt{2\pi(D+1)}}\left(\frac{4e}{D+1}\right)^{D+1} \leq \varepsilon. \tag{10}$$

Ignoring constants for large $D$, we can directly upper bound $u$ by $u(D) \leq C\left(\frac{4e}{D}\right)^D$ and thus it suffices to require $D\log D \geq \log\frac{1}{\varepsilon}$. Note that with $\varepsilon \ll 1$, we require $D$ to be "large enough". This question does not require to give precise bounds but aims at learning to quickly grasp the scaling of model or sample complexity with respect to the error, ignoring constants.

Plugging in $D = O(\log\frac{1}{\varepsilon})$ on the right hand side and ignoring constants, we obtain

$$-\log u(D) = \log\left(\log\left(\frac{1}{\varepsilon}\right)^{\log\frac{1}{\varepsilon}}\right) = \log\left(\frac{1}{\varepsilon}\right)\log\log\left(\frac{1}{\varepsilon}\right) \geq \log 1/\varepsilon$$

so that inequality (10) holds.

(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 $\varepsilon > 0$, there exists a degree $D \geq 1$ such that $\|f-\phi_D\|_\infty < \varepsilon$, 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.)

**Solution:**

Given any function $f$ that is $i$ times continuously differentiable, we can bound the magnitude of the derivatives of the function on a compact domain $I$ as $|f^{(i)}(x)| \leq T$ for some constant $T < \infty$ and all $i > 0$. The maximum approximation error is now bounded by

$$\|f(x) - \phi_m(x)\|_\infty \leq T \frac{(\sup_{x \in I} |x - x_0|)^{m+1}}{(m+1)!}.$$

Because $T$ and $\sup_{x \in I} |x - x_0|$ are constants independent of $m$, we again have $\|f(x) - \phi_m(x)\|_\infty \to 0$ as in question (b). 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.)

(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

$$\begin{aligned} f(x,y) = f(x_0,y_0) + f_x(x_0,y_0)(x-x_0) + f_y(x_0,y_0)(y-y_0) + \\ \frac{1}{2!}[f_{xx}(x_0,y_0)(x-x_0)^2 + f_{xy}(x_0,y_0)(x-x_0)(y-y_0) + \\ f_{yx}(x_0,y_0)(x-x_0)(y-y_0) + f_{yy}(x_0,y_0)(y-y_0)^2] + \ldots \end{aligned} \tag{11}$$

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

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

**Solution:**

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

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

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

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

$$f(\mathbf{v}) \approx f(\mathbf{v}_0) + \nabla f(\mathbf{v_0})^\top (\mathbf{v} - \mathbf{v}_0) + \frac{1}{2!}(\mathbf{v} - \mathbf{v}_0)^\top \mathbf{H} (\mathbf{v} - \mathbf{v}_0).$$

For the example $f(\mathbf{v}) = e^x y^2$, we obtain $\nabla f(\mathbf{v}_0) = \begin{pmatrix} e^{x_0} y_0^2 \\ e^{x_0} 2y_0 \end{pmatrix}$ and Hessian $\mathbf{H} = \begin{pmatrix} e^{x_0} y_0^2 & 2y_0 e^{x_0} \\ 2y_0 e^{x_0} & 2e^{x_0} \end{pmatrix}$.

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

By considering all such paths $\mathbf{v}(t)$ over different directions $\mathbf{v}$, we can reduce the multidimensional 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.

**Solution:** We first write $g$ explicitly in terms of $t$ using $\mathbf{v} = (x, y)$ and take its first and second order derivatives with respect to $t$:

$$\begin{aligned} g(t) &= e^{tx}(ty)^2 \\ g'(t) &= xe^{tx}(ty)^2 + 2ty^2 e^{tx} \\ g''(t) &= (x^2t^2 + 4xt + 2)y^2 e^{tx} \end{aligned}$$

with which we find that $g(0) = 0$, $g'(0) = 0$ and $g''(0) = 2y^2$. The final second order Taylor expansion of $g(t)$ around $t_0 = 0$ thus reads

$$g(t) = y^2 t^2.$$

# 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.**

**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:

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

(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` 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.

**Solution:**

```
#!/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):
    np.linalg.lstsq
    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
    for d in range(n - 1):
        D = d + 1
```

```
        for i in range(D + 1):
            if i == 0:
                Xf = np.array([1] * x_train.size)
            else:
                Xf = np.vstack([np.power(x_train, i), Xf])
        Xf = Xf.T

        w = lstsq(Xf, y_train)
        y_predicted = Xf @ w
        err[d] = (np.linalg.norm(y_train - y_predicted)**2) / n

    plt.plot(err)
    plt.xlabel('Degree of Polynomial')
    plt.ylabel('Training Error')
    plt.show()


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

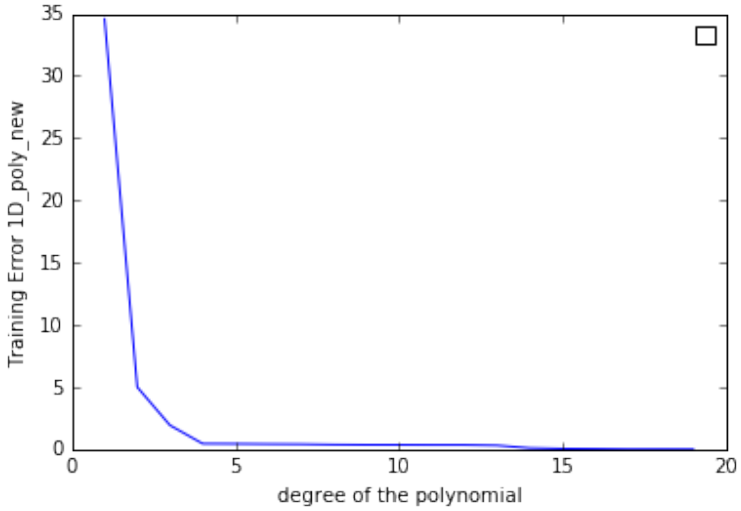



Figure 1: Training Error: **Result for 1D_poly.mat**

(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 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.

```python
#!/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
    for d in range(n - 1):
        D = d + 1
        for i in range(D + 1):
            if i == 0:
                Xf = np.array([1] * n)
            else:
                Xf = np.vstack([np.power(x_train, i), Xf])
        Xf = Xf.T

        w = lstsq(Xf, y_train)
        err_train[d] = (np.linalg.norm(y_train - Xf @ w)**2) / n
        err_fresh[d] = (np.linalg.norm(y_fresh - Xf @ w)**2) / n

    plt.figure()
    plt.ylim([0, 6])
    plt.plot(err_train, label='train')
    plt.plot(err_fresh, label='fresh')
    plt.legend()
    plt.show()

```

```
44
45 if __name__ == "__main__":
46     main()
```

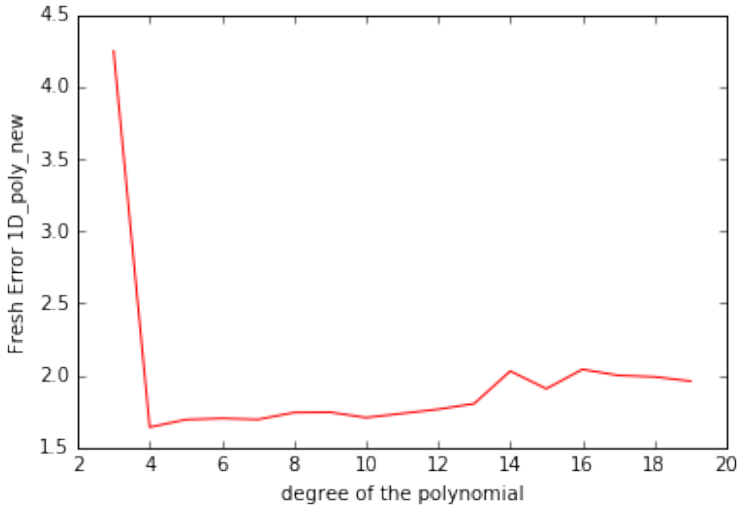



Figure 2: Fresh Error yfresh: **Result for 1D_poly.mat**

(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. 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. A $D$-multivariate polynomial function looks like

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

where $\forall j : \sum_i p_{ji} \leq D$. Here $\alpha_j$ is the scale constant for $j$th term and $p_{ji}$ is the exponent of $x_i$ in $j$th term. The data in `polynomial_regression_samples.mat` ($100000 \times 5$) with columns corresponding to the 5 attributes of the peach. **Use 4-fold cross-validation to decide which of $D \in \{0,1,2,3,4,5,6\}$ 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`.

**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 $D \in \{1,2,3,4,5,6\}$ and $\lambda \in \{0.05, 0.1, 0.15, 0.2\}$** - this is referred to as a grid search. **Find the best $D$ and $\lambda$ that best explains the data using ridge regression. Print the average training/validation error per sample for all $D$ and $\lambda$.**

**Solution:** Minimum of cross validation error happens at $D = 4$ and $\lambda = 0.15$.

```
Average train error:
[[0.05856762013 0.05856762066 0.05856762225 0.05856762491 0.05856762862]
 [0.05848920002 0.05848948229 0.0584902034  0.05849122    0.05849243261]
 [0.05846063295 0.05848702853 0.05848893458 0.05849036454 0.05849178744]
 [0.05839260718 0.05848700898 0.05848892445 0.05849035741 0.05849178171]
 [0.05831118703 0.05848700861 0.05848892426 0.05849035728 0.05849178161]
 [0.05815215946 0.0584870086  0.05848892426 0.05849035728 0.05849178161]]
Average valid error:
[[0.05857468619 0.05857468536 0.0585746856  0.05857468691 0.05857468927]
 [0.05852250667 0.05852112813 0.05852038752 0.05852010745 0.05852016181]
 [0.05854617135 0.0585210268  0.05852032221 0.05852006042 0.0585201252 ]
 [0.05860046897 0.05852102354 0.05852032035 0.05852005896 0.05852012386]
 [0.05869725681 0.05852102357 0.05852032036 0.05852005896 0.05852012386]
 [0.05887534948 0.05852102357 0.05852032036 0.05852005896 0.05852012386]]
```

In the following implementation, we run cross-validation to find errors for each $D$ separately, by varying the value of $\lambda$. 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.

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

data = spio.loadmat('polynomial_regression_samples.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]

feat_x = 0


def lstsq(A, b, lambda_=0):
    return np.linalg.solve(A.T @ A + lambda_ * np.eye(A.shape[1]), A.T @
    ↪ b)


def assemble_feature(x, D):
    n_feature = x.shape[1]
    Q = [(np.ones(x.shape[0]), 0, 0)]
    i = 0
    while Q[i][1] < D:
        cx, degree, last_index = Q[i]
        for j in range(last_index, n_feature):
            Q.append((cx * x[:, j], degree + 1, j))
        i += 1
    return np.column_stack([q[0] for q in Q])


def fit(D, lambda_):
    Ns = int(data_x.shape[0] * (Kc - 1) / Kc)  # training
    Nv = int(Ns / (Kc - 1))  # validation
```

```
34
35      Etrain = np.zeros(4)
36      Evalid = np.zeros(4)
37      for c in range(4):
38          valid_x = feat_x[c * Nv:(c + 1) * Nv]
39          valid_y = data_y[c * Nv:(c + 1) * Nv]
40          train_x = np.delete(feat_x, list(range(c * Nv, (c + 1) * Nv)),
    ↪ axis=0)
41          train_y = np.delete(data_y, list(range(c * Nv, (c + 1) * Nv)))
42
43          w = lstsq(train_x, train_y, lambda_=lambda_)
44          Etrain[c] = np.mean((train_y - train_x @ w)**2)
45          Evalid[c] = np.mean((valid_y - valid_x @ w)**2)
46
47      return np.mean(Etrain), np.mean(Evalid)
48
49
50  def main():
51      np.set_printoptions(precision=11)
52      Etrain = np.zeros((KD, len(LAMBDA)))
53      Evalid = np.zeros((KD, len(LAMBDA)))
54      for D in range(KD):
55          global feat_x
56          feat_x = assemble_feature(data_x, D + 1)
57          for i in range(len(LAMBDA)):
58              Etrain[D, i], Evalid[D, i] = fit(D + 1, LAMBDA[i])
59
60      print('Average train error:', Etrain, sep='\n')
61      print('Average valid error:', Evalid, sep='\n')
62
63      D, i = np.unravel_index(Evalid.argmin(), Evalid.shape)
64      print("D =", D + 1)
65      print("lambda =", LAMBDA[i])
66
67
68  if __name__ == "__main__":
69      main()
```

# 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.

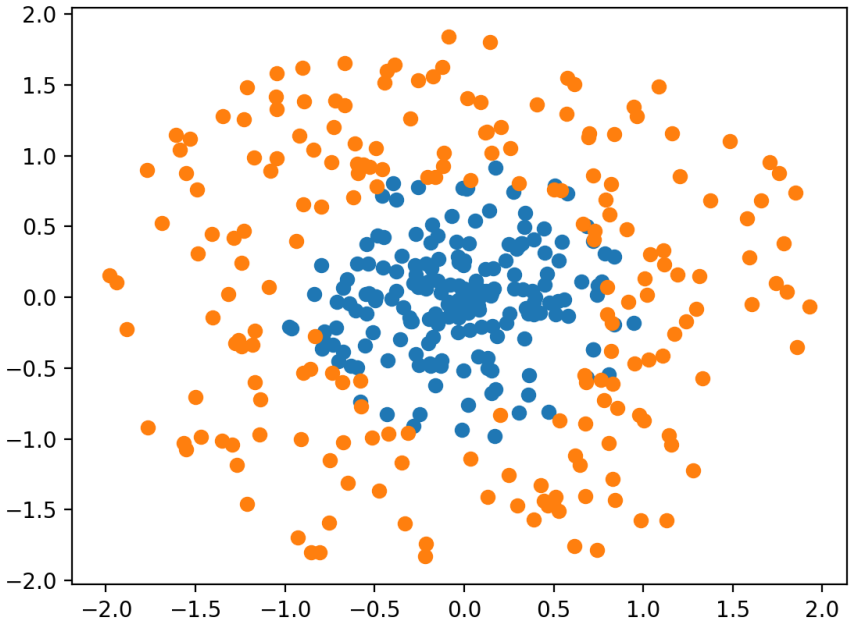
In Problem 7 on HW1, we found that linear regression can be quite effective for classification. We applied it in the setting where the training data points were *approximately linearly separable*. This means there exists a hyperplane such that most of the training data points in the first class are on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

**Solution:**




If we want to find a hyperplane that separates these points approximately, one of the half-spaces must contain most of the points from the outer ring (because they form a class). However due to the shape of the dataset, this half space will also contain most of the points in the inner circle and therefore the points cannot be classified with a linear classification boundary.

(b) **(BONUS)** Classify the points with the technique from Problem 7 on HW1 ("A Simple classification approach"). Use the feature matrix $\mathbf{X}$ 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 $\mathbf{b}$ consists of the class label $-1$ or $+1$. Perform the linear regression $\mathbf{w}_1 = \arg\min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{b}\|_2^2$. **Report the classification accuracy on the test set.**

**Solution:** The accuracy can be computed with the following code:

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

n = 400

X = np.load("Xtrain.npy")
y = np.load("ytrain.npy")

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()

# TODO: visualize the dataset
visualize_dataset(X, y)

w1 = np.linalg.solve(np.dot(X.T, X), np.dot(X.T, y))

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

ypred = -1.0*(Xtest.dot(w1) < 0.0) + 1.0*(Xtest.dot(w1) > 0.0)

print("prediction accuracy on the test set is ", 1.0*sum(ypred == ytest)
    ↪ / n)
```

The accuracy on the test set is 0.5075, so the classifier only gets about half of the predictions right. This is because the data is not linearly separable as seen in (a).

(c) **(BONUS)** Now augment the data matrix $\mathbf{X}$ 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 $\mathbf{w}_2 = \arg\min_{\mathbf{w}} \|\Phi\mathbf{w} - \mathbf{b}\|_2^2$. **Report the classification accuracy on the test set.**

**Solution:** Continuing from the code in the last part, the accuracy for the augmented data can be computed in the following way:

```
Phi = np.vstack([X[:,0], X[:,1], X[:,0]**2, X[:,0] * X[:,1], X[:,1]**2,
    ↪ 1.0 + 0.0*X[:,1]]).T

w2 = np.linalg.solve(np.dot(Phi.T, Phi), np.dot(Phi.T, y))

Phitest = np.vstack([Xtest[:,0], Xtest[:,1], Xtest[:,0]**2, Xtest[:,0] *
    ↪ Xtest[:,1], Xtest[:,1]**2, 1.0 + 0.0*Xtest[:,1]]).T
```

```
6
7  ypred2 = -1.0*(Phitest.dot(w2) < 0.0) + 1.0*(Phitest.dot(w2) > 0.0)
8
9  print("prediction accuracy on the test set is ", 1.0*sum(ypred2 == ytest)
       ↪  / n)
10
11 print("w2 = ", w2)
```

The accuracy on the test set is now 0.89 which shows that polynomial features are able to classify most of the points in the test set correctly.

(d) **(BONUS) 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 $\beta/\alpha$ here? Why did the classification in the augmented space work?**

**Solution:** The weight vector is

```
w2 =  array([ 0.08278755,  0.04680803,  0.65422045,
             -0.03610703,  0.71621139, -0.81494877])
```

so we have $\alpha \approx 0.7$ and $\beta \approx 0.8$. The interpretation of $\beta/\alpha$ is that $r^2 = \beta/\alpha$ is about the radius of the inner disk squared, so $r \approx 1$ which is the approximate radius of the inner disk. The classification in the polynomial feature space worked, because the polynomial features are rich enough to encode the exact classification rule ($x^2 + y^2 \leq r^2$).

Later on in the course, when we have access to nonlinear optimization techniques, we will see how the same ideas of using features can be combined with better loss functions for classification. But this problem should show you the power of features in classification problems as well as in regression ones.

# 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.