CM146, Winter 2024
Problem Set 2: Perceptron and Regression
Due February 19, 2024 at 11:59 pm

1 Perceptron [2 pts]

Design (specify $\boldsymbol{\theta}$ for) a two-input perceptron (with an additional bias or offset term) that computes the following boolean functions. Assume $T = 1$ and $F = -1$. If a valid perceptron exists, show that it is not unique by designing another valid perceptron (with a different hyperplane, not simply through normalization). If no perceptron exists, state why.

Solution: For $\boldsymbol{x} = (1, x_1, x_2)$, a valid perceptron defined by $\boldsymbol{\theta} = (\theta_0, \theta_1, \theta_2)$ requires that $y = \text{sign}(\boldsymbol{\theta}^T \boldsymbol{x})$.

[OR]

x_1	x_2	y
-1	-1	-1
-1	$+1$	$+1$
$+1$	-1	$+1$
$+1$	$+1$	$+1$

[XOR]

x_1	x_2	y
-1	-1	-1
-1	$+1$	$+1$
$+1$	-1	$+1$
$+1$	$+1$	-1

(a) **(1 pts)** OR

Solution: One possible solution is $\boldsymbol{\theta} = (1, 1, 1)$. Another possible solution is $\boldsymbol{\theta} = (0.5, 1, 1)$.

(b) **(1 pts)** XOR

Solution: No solution exists because the data is not linearly separable.

2 Logistic Regression [10 pts]

Consider the objective function that we minimize in logistic regression:

$$J(\boldsymbol{\theta}) = -\sum_{n=1}^{N} [y_n \log h_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + (1 - y_n) \log (1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}_n))],$$

Parts of this assignment are adapted from course material by Andrew Ng (Stanford), Jenna Wiens (UMich) and Jessica Wu (Harvey Mudd).

where $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(\boldsymbol{\theta}^T \boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{\theta}^T \boldsymbol{x}}}$.

Solution: Recall that we have $\sigma\prime(z) = \sigma(z)(1-\sigma(z))$, and thus for $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(\boldsymbol{\theta}^T \boldsymbol{x})$, we have:

$$\begin{aligned}
\frac{\partial h_{\boldsymbol{\theta}}(\boldsymbol{x})}{\partial \theta_k} &= \frac{\partial \sigma(\boldsymbol{\theta}^T \boldsymbol{x})}{\partial \theta_k} \\
&= \frac{\partial \sigma(\boldsymbol{\theta}^T \boldsymbol{x})}{\partial \boldsymbol{\theta}^T \boldsymbol{x}} \frac{\partial \left(\boldsymbol{\theta}^T \boldsymbol{x}\right)}{\partial \theta_k} \\
&= \sigma(\boldsymbol{\theta}^T \boldsymbol{x})(1-\sigma(\boldsymbol{\theta}^T \boldsymbol{x})) \frac{\partial \left(\sum_j \theta_j x_j\right)}{\partial \theta_k} \\
&= \sigma(\boldsymbol{\theta}^T \boldsymbol{x})(1-\sigma(\boldsymbol{\theta}^T \boldsymbol{x})) x_k \\
&= h_{\boldsymbol{\theta}}(\boldsymbol{x})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x})) x_k
\end{aligned} \tag{1}$$

This latter fact is very useful to make the following derivations.

(a) **(2 pts)** Find the partial derivatives $\frac{\partial J}{\partial \theta_j}$.

Solution:

$$\begin{aligned}
\frac{\partial J}{\partial \theta_j} &= \frac{\partial \left(-\sum_{n=1}^N \left[y_n \log h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right) + \left(1-y_n\right) \log\left(1-h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\right)\right]\right)}{\partial \theta_j} \\
&= -\sum_{n=1}^N \frac{\partial \left[y_n \log h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right) + \left(1-y_n\right) \log\left(1-h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\right)\right]}{\partial \theta_j} \\
&= -\sum_{n=1}^N \left(\frac{\partial \left[y_n \log h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\right]}{\partial \theta_j} + \frac{\partial \left[\left(1-y_n\right) \log\left(1-h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\right)\right]}{\partial \theta_j} \right) \\
&= -\sum_{n=1}^N \left(y_n \frac{\partial \log h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)}{\partial \theta_j} + \left(1-y_n\right) \frac{\partial \log\left(1-h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\right)}{\partial \theta_j} \right)
\end{aligned}$$

To compute the partial derivatives, we use the chain rule and identity 1

$$\begin{aligned}
\frac{\partial \log h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)}{\partial \theta_j} &= \frac{\partial \log h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)}{\partial h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)} \frac{\partial h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)}{\partial \theta_j} \\
&= \frac{1}{h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)} h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)(1-h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) x_{n,j} \\
&= (1-h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)) x_{n,j}
\end{aligned} \tag{2}$$

Similarly

$$\frac{\partial \log\left(1-h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\right)}{\partial \theta_j} = -h_{\boldsymbol{\theta}}(\boldsymbol{x}_n) x_{n,j} \tag{3}$$

Using Equations 2 and 3

$$\begin{aligned}\frac{\partial J}{\partial \theta_j} &= -\sum_{n=1}^{N}\left(y_n(1-h_{\boldsymbol{\theta}}(\boldsymbol{x}_n))x_{n,j}-(1-y_n)\,h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)x_{n,j}\right)\\ &= -\sum_{n=1}^{N}\left(y_n-h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\right)x_{n,j}\\ &= \sum_{n=1}^{N}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)-y_n\right)x_{n,j}\end{aligned}$$

(b) **(3 pts)** Find the partial second derivatives $\frac{\partial^2 J}{\partial\theta_j\partial\theta_k}$ and show that the Hessian (the matrix $\boldsymbol{H}$ of second derivatives with elements $H_{jk}=\frac{\partial^2 J}{\partial\theta_j\partial\theta_k}$) can be written as $\boldsymbol{H}=\sum_{n=1}^{N}h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\left(1-h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\right)\boldsymbol{x}_n\boldsymbol{x}_n^T$.
Solution:

$$\begin{aligned}\frac{\partial^2 J}{\partial\theta_j\partial\theta_k} &= \frac{\partial}{\partial\theta_j}\left(\frac{\partial J}{\partial\theta_k}\right)\\ &= \frac{\partial}{\partial\theta_j}\left(\sum_{n=1}^{N}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)-y_n\right)x_{n,k}\right)\\ &= \sum_{n=1}^{N}\frac{\partial\left(\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)-y_n\right)x_{n,k}\right)}{\partial\theta_j}\\ &= \sum_{n=1}^{N}\frac{\partial\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)x_{n,k}\right)}{\partial\theta_j}\\ &= \sum_{n=1}^{N}\frac{\partial h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)}{\partial\theta_j}x_{n,k}\end{aligned}$$

Using identity 1

$$\frac{\partial^2 J}{\partial\theta_j\partial\theta_k} = \sum_{n=1}^{N}h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\left(1-h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\right)x_{n,j}x_{n,k} \tag{4}$$

$\boldsymbol{H}$ is defined as:

$$\boldsymbol{H}=\sum_{n=1}^{N}h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\left(1-h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\right)\boldsymbol{x}_n\boldsymbol{x}_n^T$$

Entry j,k of $\boldsymbol{H}$ is given by:

$$\begin{aligned}H_{j,k} &= \sum_{n=1}^{N}h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\left(1-h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\right)\left(\boldsymbol{x}_n\boldsymbol{x}_n^T\right)_{j,k}\\ &= \sum_{n=1}^{N}h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\left(1-h_{\boldsymbol{\theta}}\left(\boldsymbol{x}_n\right)\right)x_{n,j}x_{n,k}\\ &= \frac{\partial^2 J}{\partial\theta_j\partial\theta_k}, \qquad \text{from Equation 4}\end{aligned}$$

Thus, $\boldsymbol{H}$ is the Hessian.

(c) **(5 pts)** Show that J is a convex function and therefore has no local minima other than the global one.
Hint: A function J is convex if its Hessian is positive semi-definite (PSD), written $\boldsymbol{H} \succeq 0$. A matrix is PSD if and only if

$$\boldsymbol{z}^T \boldsymbol{H} \boldsymbol{z} \equiv \sum_{j,k} z_j z_k H_{jk} \geq 0.$$

for all real vectors $\boldsymbol{z}$.
Solution: So we have for the Hessian matrix $\boldsymbol{H}$ (using that for $\boldsymbol{X} = \boldsymbol{x}\boldsymbol{x}^T$ if and only if $X_{ij} = x_i x_j$):

$$\boldsymbol{H} = \sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\right) \boldsymbol{x}_n {\boldsymbol{x}_n}^T.$$

And to prove $\boldsymbol{H}$ is positive semidefinite, we show $\boldsymbol{z}^T \boldsymbol{H} \boldsymbol{z} \geq 0$ for all real vectors $\boldsymbol{z}$.

$$\begin{aligned}
\boldsymbol{z}^T \boldsymbol{H} \boldsymbol{z} &= \boldsymbol{z}^T \left(\sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\right) \boldsymbol{x}_n {\boldsymbol{x}_n}^T \right) \boldsymbol{z} \\
&= \sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\right) \boldsymbol{z}^T \boldsymbol{x}_n {\boldsymbol{x}_n}^T \boldsymbol{z} \\
&= \sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\right) \left(\boldsymbol{z}^T \boldsymbol{x}_n\right)^2 \\
&\geq 0
\end{aligned}$$

with the last inequality holding because $0 \leq h(\boldsymbol{x}) \leq 1$, which implies that $h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\right) \geq 0$, and $\left(\boldsymbol{z}^T \boldsymbol{x}_n\right)^2 \geq 0$.

3 Maximum Likelihood Estimation [15 pts]

Suppose we observe the values of n independent random variables $X_1, \ldots, X_n$ drawn from the same Bernoulli distribution with parameter θ[1]. In other words, for each X_i, we know that

$$P(X_i = 1) = \theta \quad \text{and} \quad P(X_i = 0) = 1 - \theta.$$

Our goal is to estimate the value of θ from these observed values of X_1 through X_n.

For any hypothetical value $\hat{\theta}$, we can compute the probability of observing the outcome $X_1, \ldots, X_n$ if the true parameter value θ were equal to $\hat{\theta}$. This probability of the observed data is often called the *likelihood*, and the function $L(\theta)$ that maps each θ to the corresponding likelihood is called the *likelihood function*. A natural way to estimate the unknown parameter θ is to choose the θ that maximizes the likelihood function. Formally,

$$\hat{\theta}_{MLE} = \arg\max_{\theta} L(\theta).$$

[1]This is a common assumption for sampling data. So we will denote this assumption as iid, short for Independent and Identically Distributed, meaning that each random variable has the same distribution and is drawn independent of all the other random variables

(a) **(3 pts)** Write a formula for the likelihood function, $L(\theta) = P(X_1, \ldots, X_n; \theta)$. Your function should depend on the random variables $X_1, \ldots, X_n$ and the hypothetical parameter θ. Does the likelihood function depend on the order in which the random variables are observed ?

Solution: Since the X_i are independent, we have

$$\begin{aligned} L(\theta) &= P_\theta(X_1, \ldots, X_n) \\ &= \prod_{i=1}^{n} P_\theta(X_i) \\ &= \prod_{i=1}^{n} \theta^{X_i}(1-\theta)^{1-X_i} \\ &= \theta^{\#\{X_i=1\}}(1-\theta)^{\#\{X_i=0\}}, \end{aligned}$$

where $\#\{\cdot\}$ counts the number of X_i for which the condition in braces holds true. In the third line, we used the trick $X_i = \mathbb{I}\{X_i = 1\}$. The likelihood function does not depend on the order of the data.

Common mistakes:

- For this problem, many people wrote the definition of the likelihood function $L(\theta) = P_\theta(X_1, \ldots, X_n) = \prod_{i=1}^{n} P_\theta(X_i)$. Since you knew the distribution of the X_i, we expected you to write out an explicit formula.

(b) **(6 pts)** Since the log function is increasing, the θ that maximizes the *log likelihood* $\ell(\theta) = \log(L(\theta))$ is the same as the θ that maximizes the likelihood. Find $\ell(\theta)$ and its first and second derivatives, and use these to find a closed-form formula for the MLE. For this part, assume we use the natural logarithm (i.e. logarithm base e).

Solution: Introduce the shorthand $n_1 = \#\{X_i = 1\}$ and $n_0 = \#\{X_i = 0\}$. Using the properties of the log function, we can rewrite ℓ as follows:

$$\begin{aligned} \ell(\theta) &= \log(\theta^{n_1}(1-\theta)^{n_0}) \\ &= n_1 \log(\theta) + n_0 \log(1-\theta). \end{aligned}$$

The first and second derivatives of ℓ are given by

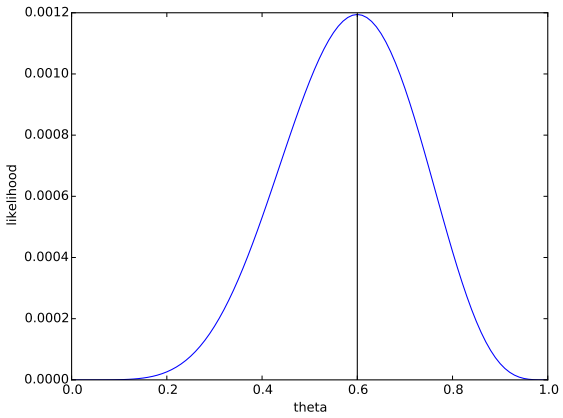
$$\ell'(\theta) = \frac{n_1}{\theta} - \frac{n_0}{1-\theta} \quad \text{and} \quad \ell''(\theta) = -\left(\frac{n_1}{\theta^2} + \frac{n_0}{(1-\theta)^2}\right).$$

Since the second derivative is always negative, the function ℓ is concave, and we can find its maximizer by solving $\ell'(\theta) = 0$. The solution to this equation is obtained by straight-forward algebra and is given by

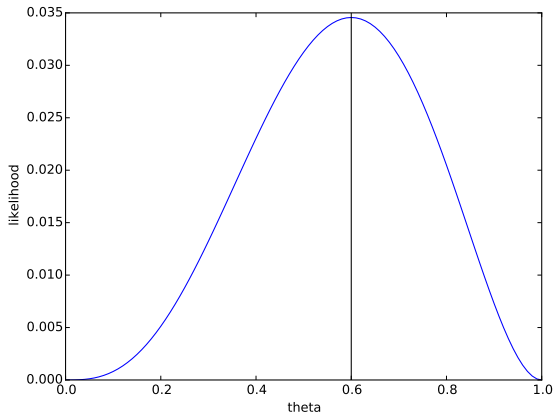
$$\hat{\theta}_{MLE} = \frac{n_1}{n_1 + n_0}.$$

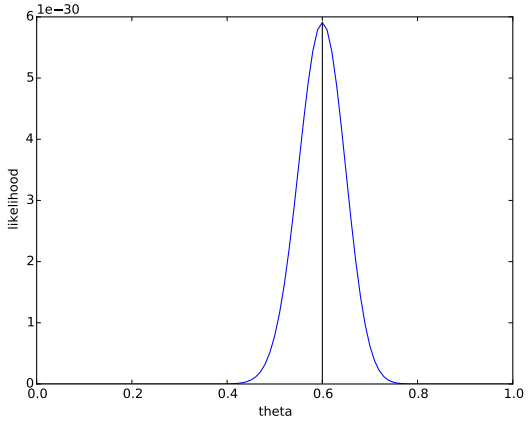
(c) **(3 pts)** Suppose that $n = 10$ and the data set contains six 1s and four 0s. Write a short program `likelihood.py` that plots the likelihood function of this data for each value of θ in $\{0, 0.01, 0.02, \ldots, 1.0\}$ (use `np.linspace(...)` to generate this spacing). For the plot, the x-axis should be θ and the y-axis $L(\theta)$. Scale your y-axis so that you can see some variation in its value. Include the plot in your writeup (there is no need to submit your code). Estimate $\hat{\theta}_{MLE}$ by marking on the x-axis the value of θ that maximizes the likelihood. Does the answer

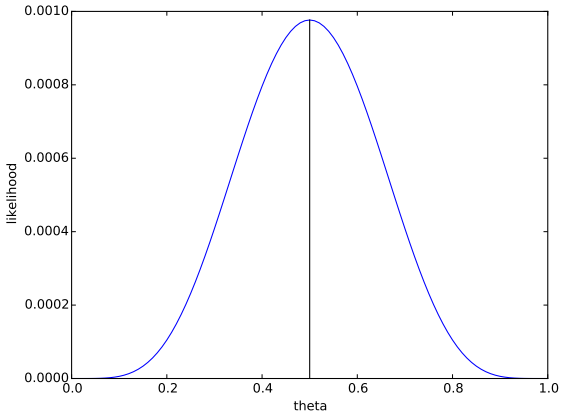
agree with the closed form answer ?
Solution:



(d) **(3 pts)** Create three more likelihood plots: one where $n = 5$ and the data set contains three 1s and two 0s; one where $n = 100$ and the data set contains sixty 1s and forty 0s; and one where $n = 10$ and there are five 1s and five 0s. Include these plots in your writeup, and describe how the likelihood functions and maximum likelihood estimates compare for the different data sets.
Solution:







The MLE is equal to the proportion of 1s observed in the data, so for the first three plots the MLE is always at 0.6 and for the last plot it is at 0.5. As the number of samples n increases, the likelihood function gets more peaked at its maximum value, and the values it takes on decrease.

4 Implementation: Polynomial Regression [20 pts]

In this exercise, you will work through linear and polynomial regression. Our data consists of inputs $x_n \in \mathbb{R}$ and outputs $y_n \in \mathbb{R}, n \in \{1, \ldots, N\}$, which are related through a target function $y = f(x)$. Your goal is to learn a linear predictor $h_{\boldsymbol{\theta}}(x)$ that best approximates $f(x)$. But this time, rather than using `scikit-learn`, we will further open the "black-box", and you will implement the regression model!

---

code and data

- code: `CS146_Winter2024_PS2.ipynb`
- data: `regression_train.csv`, `regression_test.csv`

Similar to *PS1*, copy the colab notebook to your drive and make the changes to the notebook. Mount the drive appropriately and copy the data to your drive to access via colab.

The notebook has marked blocks where you need to code.

$$\#\#\# \text{ ========= } TODO : START \text{ ========= } \#\#\#$$

$$\#\#\# \text{ ========= } TODO : END \text{ ========== } \#\#\#$$

Note: For parts b, c, g, and h (and only these parts), you are expected to copy-paste/screenshot your code snippet as a part of the solution in the submission pdf. Tip: If you are using LaTeX, check out the Minted package (example) for code highlighting.

This is likely the first time that many of you are working with `numpy` and matrix operations within a programming environment. For the uninitiated, you may find it useful to work through a `numpy` tutorial first.[2] Here are some things to keep in mind as you complete this problem:

- If you are seeing many errors at runtime, inspect your matrix operations to make sure that you are adding and multiplying matrices of compatible dimensions. Printing the dimensions of variables with the `X.shape` command will help you debug.
- When working with `numpy` arrays, remember that `numpy` interprets the `*` operator as element-wise multiplication. This is a common source of size incompatibility errors. If you want matrix multiplication, you need to use the `dot` function in Python. For example, `A*B` does element-wise multiplication while `dot(A,B)` does a matrix multiply.
- Be careful when handling `numpy` vectors (rank-1 arrays): the vector shapes $1 \times N$, $N \times 1$, and N are all different things. For these dimensions, we follow the the conventions of `scikit-learn`'s `LinearRegression` class[3]. Most importantly, unless otherwise indicated (in the code documentation), both column and row vectors are rank-1 arrays of shape N, not rank-2 arrays of shape $N \times 1$ or shape $1 \times N$.

Visualization [1 pts]

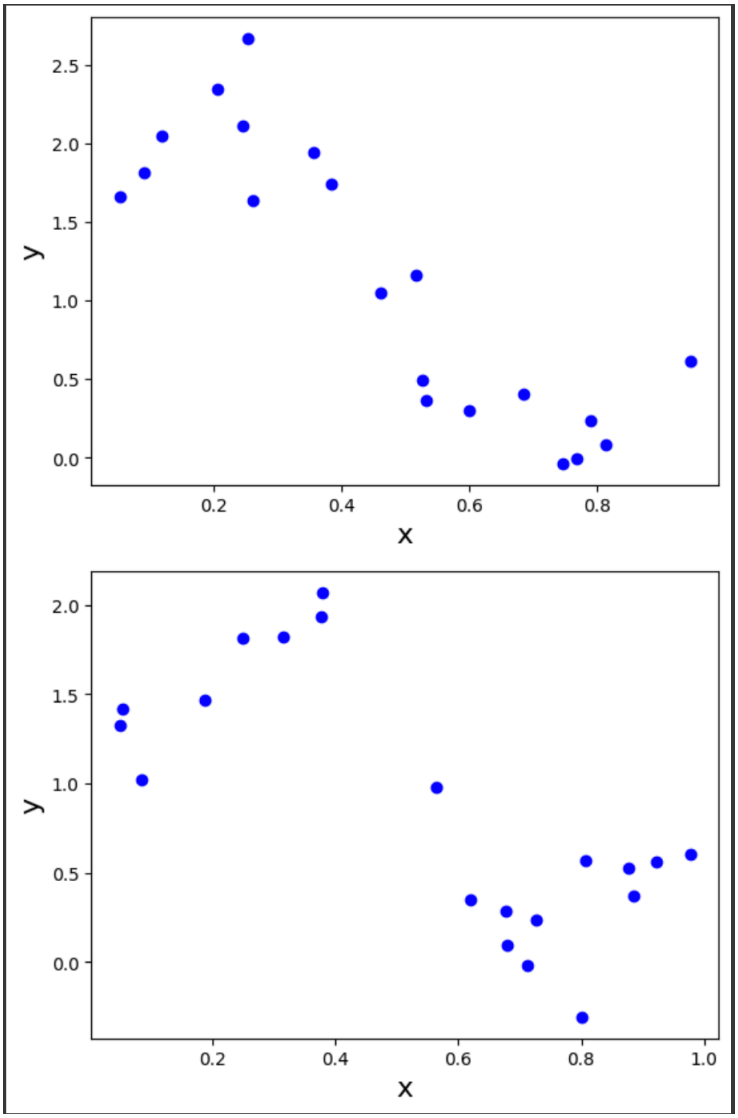
[2] Try out `SciPy`'s tutorial (https://numpy.org/doc/stable/user/quickstart.html), or use your favorite search engine to find an alternative. Those familiar with Matlab may find the "Numpy for Matlab Users" documentation (https://numpy.org/doc/stable/user/numpy-for-matlab-users.html) more helpful.

[3] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

It is often useful to understand the data through visualizations. For this data set, you can use a scatter plot to visualize the data since it has only two properties to plot (x and y).

(a) **(1 pts)** Visualize the training and test data using the `plot_data(...)` function. What do you observe? For example, can you make an educated guess on the effectiveness of linear regression in predicting the data?

Solution:



Top plot is for training data, bottom plot is for test data. We observe that the data is nonlinear, so linear regression will probably not be very effective.

Linear Regression [12 pts]

Recall that linear regression attempts to minimize the objective function

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} \left(h_{\boldsymbol{\theta}}(\boldsymbol{x}_n) - y_n\right)^2.$$

In this problem, we will use the matrix-vector form where

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, \qquad \boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_N^T \end{pmatrix}, \qquad \boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_D \end{pmatrix}$$

and each instance $\boldsymbol{x}_n = \left(1, x_{n,1}, \ldots, x_{n,D}\right)^T$.

In this instance, the number of input features $D = 1$.

Rather than working with this fully generalized, multivariate case, let us start by considering a simple linear regression model:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{\theta}^T \boldsymbol{x} = \theta_0 + \theta_1 x_1$$

The Colab notebook contains the skeleton code for the class `PolynomialRegression`. Objects of this class can be instantiated as `model = PolynomialRegression (m)` where m is the degree of the polynomial feature vector where the feature vector for instance n is $\left(1, x_{n,1}, x_{n,1}^2, \ldots, x_{n,1}^m\right)^T$. Setting $m = 1$ instantiates an object where the feature vector for instance n is $\left(1, x_{n,1}\right)^T$.

(b) **(1 pts)** Note that to take into account the intercept term (θ_0), we can add an additional "feature" to each instance and set it to one, e.g. $x_{n,0} = 1$. This is equivalent to adding an additional first column to $\boldsymbol{X}$ and setting it to all ones.

Modify `PolynomialRegression.generate_polynomial_features(...)` to create the matrix $\boldsymbol{X}$ for a simple linear model. Copy-paste/screenshot your code snippet.

Solution: Code is below. Note that this code works for a polynomial model, i.e. for part g. Any code that is correct for a linear model (for example, concatenating a column of ones to X) is fine.

```
def generate_polynomial_features(self, X) :
    """
    Maps X to an mth degree feature vector e.g. [1, X, X^2, ..., X^m].

    Parameters
    ----------------------
        X       -- numpy array of shape (n,1), features

    Returns
    ----------------------
        Phi     -- numpy array of shape (n,(m+1)), mapped features
```

```
    """

    n,d = X.shape

    ### ========== TODO : START ========== ###
    # part b: modify to create matrix for simple linear model
    # part g: modify to create matrix for polynomial model
    Phi = X
    m = self.m_

    Phi = np.ones((n, m+1))
    for i in range(1,m+1,1):
      Phi[:,i] = X.flatten()**(i)

    ### ========== TODO : END ========== ###

    return Phi
```

(c) **(1 pts)** Before tackling the harder problem of training the regression model, complete `PolynomialRegression.predict(...)` to predict $\boldsymbol{y}$ from $\boldsymbol{X}$ and $\boldsymbol{\theta}$. Copy-paste/screenshot your code snippet.

Solution: Code is below.

```
def predict(self, X) :
    """
    Predict output for X.

    Parameters
    --------------------
        X       -- numpy array of shape (n,d), features

    Returns
    --------------------
        y       -- numpy array of shape (n,), predictions
    """
    if self.coef_ is None :
        raise Exception("Model not initialized. Perform a fit first.")

    X = self.generate_polynomial_features(X) # map features

    ### ========== TODO : START ========== ###
    # part c: predict y
    y = np.dot(X, self.coef_)
    ### ========== TODO : END ========== ###

    return y
```

(d) **(5 pts)** One way to solve linear regression is through gradient descent (GD).

Recall that the parameters of our model are the θ_j values. These are the values we will adjust to minimize $J(\boldsymbol{\theta})$.

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} (h_{\boldsymbol{\theta}}(\boldsymbol{x}_n) - y_n)^2$$

In gradient descent, each iteration performs the update

$$\theta_j \leftarrow \theta_j - 2\eta \sum_{n=1}^{N} (h_{\boldsymbol{\theta}}(\boldsymbol{x}_n) - y_n)\, x_{n,j} \quad \text{(simultaneously update } \theta_j \text{ for all } j\text{)}.$$

With each step of gradient descent, we expect our updated parameters θ_j to come closer to the parameters that will achieve the lowest value of $J(\boldsymbol{\theta})$.

- As we perform gradient descent, it is helpful to monitor the convergence by computing the cost, *i.e.*, the value of the objective function J. Complete `PolynomialRegression.cost(...)` to calculate $J(\boldsymbol{\theta})$.

  If you have implemented everything correctly, then the following code snippet should return 40.234.

```
train_data = load_data('regression_train.csv') # Use your own path
model = PolynomialRegression()
model.coef_ = np.zeros(2)
model.cost(train_data.X, train_data.y)
```

- Next, implement the gradient descent step in `PolynomialRegression.fit_GD(...)`. The loop structure has been written for you, and you only need to supply the updates to $\boldsymbol{\theta}$ and the new predictions $\hat{y} = h_{\boldsymbol{\theta}}(\boldsymbol{x})$ within each iteration.

  We will use the following specifications for the gradient descent algorithm:

  - We run the algorithm for $10,000$ iterations.
  - We terminate the algorithm earlier if the value of the objective function is unchanged across consecutive iterations.
  - We will use a fixed learning rate.

- Experiment with different values of learning rate $\eta = 10^{-4}, 10^{-3}, 10^{-2}, 0.1$, and make a table of the coefficients, number of iterations until convergence (this number will be $10,000$ if the algorithm did not converge in a smaller number of iterations) and the final value of the objective function. How do the coefficients compare? How quickly does each algorithm converge?

  Solution: We provide the table below. We note that the coefficients for $\eta = 0.001, 0.01$ are the same, and the coefficients for $\eta = 0.0001$ are slightly different. $\eta = 0.01$ converges the fastest, followed by $\eta = 0.001$; $\eta = 0.0001$ does not fully converge in 10,000 iterations (learning rate is too small); $\eta = 0.1$ diverges (learning rate is too large).

	eta	theta_0	theta_1	iterations	cost	time
0	0.0001	2.270448	-2.460648	10000	4.086397	1.017170
1	0.0010	2.446407	-2.816353	7021	3.912576	0.499244
2	0.0100	2.446407	-2.816353	765	3.912576	0.045830
3	0.1000	NaN	NaN	10000	NaN	0.708435

(e) **(4 pts)** In class, we learned that the closed-form solution to linear regression is

$$\boldsymbol{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}.$$

Using this formula, you will get an exact solution in one calculation: there is no "loop until convergence" like in gradient descent.

- Implement the closed-form solution `PolynomialRegression.fit(...)`.
- What is the closed-form solution coefficients? How do the coefficients and the cost compare to those obtained by GD? Use the 'time' module to compare its runtime to GD.

  Solution:

```
model parameter by close form: [ 2.44640709 -2.81635359]
train cost close form: 3.9125764057914636
close form time: 0.0012502670288085938
```

The closed-form solution coefficients and cost are the same as those obtained using GD with $\eta = 0.001, 0.01$ and are slightly different (slightly lower cost) from those obtained using GD with $\eta = 0.0001$. The closed-form solution is significantly faster (in terms of runtime) than gradient descent (even with $\eta = 0.01$, the fastest value) because the number of data points and number of features are small for our dataset.

(f) **(1 pts)** Finally, set a learning rate η for GD that is a function of k (the number of iterations) (use $\eta_k = \frac{1}{1+k}$) and converges to the same solution yielded by the closed-form optimization (minus possible rounding errors). Update `PolynomialRegression.fit_GD(...)` with your proposed learning rate. How many iterations does it take the algorithm to converge with your proposed learning rate?

Solution: We provide the coefficients, number of iterations, and training cost below; only the number of iterations is necessary for full credit.

```
auto step size model.coef_: [ 2.44640676 -2.81635292]
auto step size model.iter_: 1357
auto step size train cost: 3.9125764057920818
```

Gradient descent with this learning rate schedule takes 1357 iterations to converge.

Polynomial Regression [7 pts]

Now let us consider the more complicated case of polynomial regression, where our hypothesis is

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{\theta}^T\phi(\boldsymbol{x}) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_m x^m.$$

(g) **(1 pts)** Recall that polynomial regression can be considered as an extension of linear regression in which we replace our input matrix $\boldsymbol{X}$ with

$$\boldsymbol{\Phi} = \begin{pmatrix} \phi(x_1)^T \\ \phi(x_2)^T \\ \vdots \\ \phi(x_N)^T \end{pmatrix},$$

where $\phi(x)$ is a function such that $\phi_j(x) = x^j$ for $j = 0, \ldots, m$.

Update `PolynomialRegression.generate_polynomial_features(...)` to create an $m+1$ dimensional feature vector for each instance. Copy-paste/screenshot your code snippet.

Solution: Code is below. Any correct implementation (for example, using concatenation instead of manually setting columns of a new array) is fine.

```
def generate_polynomial_features(self, X) :
    """
    Maps X to an mth degree feature vector e.g. [1, X, X^2, ..., X^m].

    Parameters
    --------------------
        X       -- numpy array of shape (n,1), features

    Returns
    --------------------
        Phi     -- numpy array of shape (n,(m+1)), mapped features
    """

    n,d = X.shape

    ### ========== TODO : START ========== ###
    # part b: modify to create matrix for simple linear model
    # part g: modify to create matrix for polynomial model
    Phi = X
    m = self.m_

    Phi = np.ones((n, m+1))
    for i in range(1,m+1,1):
      Phi[:,i] = X.flatten()**(i)

    ### ========== TODO : END ========== ###

    return Phi
```

(h) **(2 pts)** Given N training instances, it is always possible to obtain a "perfect fit" (a fit in which all the data points are exactly predicted) by setting the degree of the regression to $N-1$. Of course, we would expect such a fit to generalize poorly. In the remainder of this problem, you will investigate the problem of overfitting as a function of the degree of the polynomial, m. To measure overfitting, we will use the Root-Mean-Square (RMS) error, defined as

$$E_{RMS} = \sqrt{J(\boldsymbol{\theta})/N},$$

where N is the number of instances.[4]

Why do you think we might prefer RMSE as a metric over $J(\boldsymbol{\theta})$?

[4]Note that the RMSE as defined is a biased estimator. To obtain an unbiased estimator, we would have to divide by $n-k$, where k is the number of parameters fitted (including the constant), so here, $k = m+1$.

Implement `PolynomialRegression.rms_error(...)`. Copy-paste/screenshot your code snippet.

Solution: We might prefer RMSE because 1) RMSE uses the average cost as it divides by N, the number of data points and 2) RMSE has the same units as our labels y_n while $J(\boldsymbol{\theta})$ is in squared units.

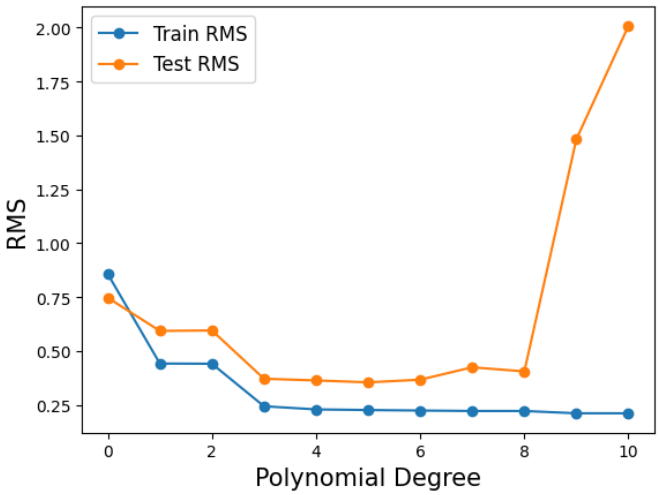
```
def rms_error(self, X, y) :
    """
    Calculates the root mean square error.

    Parameters
    --------------------
        X       -- numpy array of shape (n,d), features
        y       -- numpy array of shape (n,), targets

    Returns
    --------------------
        error   -- float, RMSE
    """
    ### ========== TODO : START ========== ###
    # part h: compute RMSE
    error = np.sqrt(self.cost(X, y) / X.shape[0])
    ### ========== TODO : END ========== ###
    return error
```

(i) **(4 pts)** For $m = 0, \ldots, 10$ (where m is the degree of the polynomial), use the closed-form solver to determine the best-fit polynomial regression model on the training data, and with this model, calculate the RMSE on both the training data and the test data. Generate a plot depicting how RMSE varies with model complexity (polynomial degree) – you should generate a single plot with both training and test error, and include this plot in your writeup. Which degree polynomial would you say best fits the data? Was there evidence of under/overfitting the data? Use your plot to justify your answer.

Solution:



The degree 5 polynomial best fits the data (lowest test error). From the plot, we see that low degree polynomials underfit (high train and test error) while high degree polynomials overfit (low train error but high test error).