Polynomial Regression

Goals:

- To become more familiar with vector and matrix operations in code
- To investigate two different approaches to fitting regression models
- To think about the pros and cons of more flexible models

Submission

You should submit your modifications to the provided starter code, along with a file responding to the questions for each part. You can either use LaTeX and submit this in PDF form (i.e. `writeup.pdf`) or use markdown and submit it as part of the `README.md`. This writeup should include your names at the top of the first page, and it should clearly label all problems that require a response (look for the blue writeup!) Additionally, cite any collaborators and sources of help you received.

You do not need to write a lot of code for this assignment (mainly you will fill in key steps), but anything you write should be clear and commented as necessary. The most important things:

- You should include a header in triple quotes at the top of each file (include your names, date, and program description).
- Each function and method should have an appropriate doctsring.
- If anything is complicated, it should include some comments.
- When you are ready to submit, make sure that your code runs and remove any debugging print statements.

Introduction

In this exercise, you will work through linear and polynomial regression. Our data consists of inputs $x_i \in \mathbb{R}$ and outputs $y_i \in \mathbb{R}$, which are related through a target function $y = f(x)$. Your goal is to learn a linear predictor $h_{\boldsymbol{b}}(x)$ that best approximates $f(x)$.[1]

---

code and data

- code : `PolynomialRegression.py`, `run_regression.py`
- data : `regression_train.csv`, `regression_test.csv`

[1] *Adapted from course material by Jessica Wu.*

A note about ***numpy***: We used `numpy` before in Lab 1, but we'll use more of its functionality in this lab. It is a good skill to pick up since you will inevitably use `numpy` if you plan to do math in Python, e.g. for machine learning. 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, note 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 `np.dot` function. For example, `A*B` does element-wise multiplication while `np.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 simplicity in this lab (unless otherwise indicated in the code), both column and row vectors are rank-1 arrays of shape n, not rank-2 arrays of shape $n \times 1$ or shape $1 \times n$.

Useful `numpy` functions:

- `np.array([...])`: takes in a list (or list-of-lists, etc) and turns it into a numpy array. Note that after this step, you cannot `append`, etc – you have to use `numpy` operations to create new array objects.
- `np.dot(A,B)`, `np.matmul(A,B)`: perform matrix multiplication. Inner dimensions of A and B should match.
- `np.ones((p,q))`: creates a 2D array of 1's with the given shape (here it would be $p \times q$). Similarly, you can use `np.zeros((p,q))` to create an array of 0's. If you include only one dimension (i.e. `np.zeros(p)`), this will yield a vector.
- `np.concatenate((A, B), axis=0)`: concatenate two arrays along the given axis. Here it would be along the rows (axis 0), so A would be "on top", and B below. If we concatenate along axis 1 then A would be on the "left" and B would be on the right.
- `np.reshape(A, (p,q))`: will reshape array A to have dimensions $p \times q$ (or whatever dimensions you need, provided the values of A can actually be reshaped in this way).
- `np.linalg.pinv(A)`: returns the pseudo-inverse of matrix A (we will use this just in case we have issues with a singular matrix).

[2]Try out `SciPy`'s tutorial (`https://docs.scipy.org/doc/numpy/user/quickstart.html`). Those familiar with Matlab may find the "Numpy for Matlab Users" documentation (`https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html`) more helpful.

Visualization

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

(a) 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 and polynomial regression in predicting the data? Include your responses and plots of both training and test data in your writeup.

Hint: As you implement the remaining exercises, use this plotting function as a debugging tool.

Linear Regression

Recall that the objective of linear regression is to minimize the cost function

$$J(\boldsymbol{b}) = \frac{1}{2}\sum_{i=1}^{n}\left(h_{\boldsymbol{b}}(\boldsymbol{x}_i) - y_i\right)^2.$$

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

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \qquad \boldsymbol{X} = \begin{pmatrix} — & \boldsymbol{x}_1 & — \\ — & \boldsymbol{x}_2 & — \\ & \vdots & \\ — & \boldsymbol{x}_n & — \end{pmatrix}, \qquad \boldsymbol{b} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix}$$

where each example $\boldsymbol{x}_i = \left[1, x_{i1}, \ldots, x_{ip}\right]$. Our linear regression model is:

$$h_{\boldsymbol{b}}(\boldsymbol{x}) = \boldsymbol{b}^T\boldsymbol{x}$$

Beginning with the simple linear regression case ($p = 1$), we have:

$$h_{\boldsymbol{b}}(\boldsymbol{x}) = b_0 + b_1 x$$

(b) Note that to take into account the intercept term (b_0), we can add an additional "feature" to each example and set it to one, e.g. $x_{i0} = 1$. This is equivalent to adding an additional first column to $\boldsymbol{X}$ and setting it to all ones.

Modify `generate_polynomial_features(...)` in `PolynomialRegression` to create the matrix $\boldsymbol{X}$ for a simple linear model.

(c) Before tackling the harder problem of training the regression model, complete `predict(...)` in `PolynomialRegression` to predict $\boldsymbol{y}$ from $\boldsymbol{X}$ and $\boldsymbol{b}$.

(d) One way to solve linear regression is through stochastic gradient descent (SGD).

Recall that the parameters of our model are the b_j values. These are the values we will adjust to minimize cost $J(\boldsymbol{b})$. In SGD, each iteration runs through the training set and performs the update

$$b_j \leftarrow b_j - \alpha \left(h_{\boldsymbol{b}}(\boldsymbol{x}_i) - y_i\right) x_{ij} \quad \text{(simultaneously update } b_j \text{ for all } j\text{)}.$$

With each step of gradient descent, our parameters b_j come closer to the optimal values that will achieve the lowest cost $J(\boldsymbol{b})$.

- As we perform gradient descent, it is helpful to monitor the convergence by computing the cost. Complete `cost(...)` in `PolynomialRegression` to calculate $J(\boldsymbol{b})$.
- Next, implement the gradient descent step in `fit_SGD(...)` in `PolynomialRegression`. The loop structure has been written for you, so you only need to supply the updates to $\boldsymbol{b}$ and the new predictions $\hat{y}$ within each iteration. In your writeup, include a plot of the final fit to the training data for $\alpha = 0.01$.

  Hint: A good way to verify that gradient descent is working correctly is to look at the value of $J(\boldsymbol{b})$ and check that it is decreasing with each step. If you set `verbose=True` when calling `fit_SGD(...)`, then, assuming you have implemented gradient descent and `cost(...)` correctly, your value of $J(\boldsymbol{b})$ should never increase and should converge to a steady value by the end of the algorithm.

  Hint: With `verbose=True`, you may also find it useful to look at the 2D plots of the training data and the output of the trained regression model to verify that the model looks reasonable and is improving on each step.
- So far, we have used a default learning rate (or step size) of $\alpha = 0.01$. Try different $\alpha = 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}$, and make a table of the coefficients and number of iterations until convergence (in your writeup). How do the coefficients compare? How quickly does each algorithm converge?

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

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

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 `fit(...)` in `PolynomialRegression`.
- writeup: How do the coefficients compare to those obtained by SGD? How does your comparison change as α changes?
- Use Python's `time` module to measure the run time of the two approaches. You can capture the current time with `time.time()` (differences between these values are measured in seconds). How do the run times compare? writeup

Polynomial Regression

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

$$h_{\boldsymbol{b}}(\boldsymbol{x}) = \boldsymbol{b}^T\boldsymbol{x} = b_0 + b_1 x + b_2 x^2 + \ldots + b_d x^d.$$

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

$$\boldsymbol{\Phi} = \begin{pmatrix} — & \phi(x_1) & — \\ — & \phi(x_2) & — \\ & \vdots & \\ — & \phi(x_n) & — \end{pmatrix},$$

where $\phi(x)$ is a function such that $\phi_k(x) = x^k$ for $k = 0, \ldots, d$.

Update `generate_polynomial_features(...)` in `PolynomialRegression` to create an $d + 1$ dimensional feature vector for each example (okay to assume $p = 1$ for this part).

(g) Given n training examples, 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 $n - 1$ (for example, in Handout 2 we had $n = 2$ points and a polynomial of deg $d = 1$ fit them perfectly). 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, d. To measure overfitting, we will use the Root-Mean-Square (RMS) error, defined as

$$E_{RMS} = \sqrt{2\mathbb{E}[\boldsymbol{b}]/n},$$

where

$$\mathbb{E}[\boldsymbol{b}] = \frac{1}{2}\sum_{i=1}^{n}\left(\sum_{k=0}^{d} b_k\phi_k(\boldsymbol{x}_i) - y_i\right)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(h_{\boldsymbol{b}}(\boldsymbol{x}_i) - y_i\right)^2$$

and n is the number of examples.

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

Implement `rms_error(...)` in `PolynomialRegression`.

(h) For $d = 0, \ldots, 10$, 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 defend your answer. writeup

Finally, for the degree you chose, show a plot of the final fit to the training data (i.e. use the `verbose=True` flag). writeup

Regularization (optional extension)

Finally, we will explore the role of regularization. For this problem, we will use L_2-regularization so that our regularized objective function is

$$J(\boldsymbol{b}) = \frac{1}{2}\sum_{i=1}^{n}\left(h_{\boldsymbol{b}}(\boldsymbol{x}_i) - y_i\right)^2 + \frac{\lambda}{2}\sum_{k=1}^{d} b_k^2,$$

again optimizing for the parameters $\boldsymbol{b}$.

(i) Modify `fit(...)` in `PolynomialRegression` to incorporate L_2-regularization.

(j) Use your updated solver to find the coefficients that minimize the error for a tenth-degree polynomial ($d = 10$) given regularization factor $\lambda = 0, 10^{-8}, 10^{-7}, \ldots, 10^{-1}, 10^{0}$. Now use these coefficients to calculate the RMS error (unregularized) on both the training data and test data as a function of λ. Generate a plot depicting how RMS error varies with λ (for your x-axis, let $x = [0, 1, 2, \ldots, 10]$ correspond to $\lambda = [0, 10^{-8}, 10^{-7}, \ldots, 10^{0}]$ so that λ is on a logistic scale, with regularization increasing as x increases). How does regularization affect training and test error? Which λ value appears to work best? writeup