HMC CS 158, Fall 2017
Problem Set 3 Programming: Regularized Polynomial Regression

Goals:

- To open up the "black-box" of `scikit-learn` and implement regression models.
- To investigate how adding polynomial features and regularization affect performance.

For this assignment, you can work individually though you are encouraged to work with a partner. You should sign up for partners on Canvas (People → PS3 Groups). If you are looking for a partner, try Piazza. If, after trying Piazza, you are having trouble finding a partner, e-mail Jessica.

Submission

You should submit any answers to the exercises in a single file `writeup.pdf`. This writeup should include your name and the assignment number at the top of the first page, and it should clearly label all problems. Additionally, cite any collaborators and sources of help you received (excluding course staff), and if you are using slip days, please also indicate this at the top of your document.

Your code should be commented appropriately (though you do not need to go overboard). The most important things:

- Your name and the assignment number should be at the top of each file.
- Each class and method should have an appropriate doctsring.
- If anything is complicated, it should include some comments.

There are many possible ways to approach the programming portion of this assignment, which makes code style and comments very important so that staff can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.

When you are ready to submit, make sure that your code compiles and remove any debugging statements. Then rename the top-level directory as `<username1>_<username2>_ps3`, with the usernames in alphabetical order (e.g. `hadas_yjw_ps3`). This directory should include the electronic version of your writeup and main code, any files necessary to run your code (including any code and data provided by staff), and follow the same structure as the assignment directory provided by staff. So, for this assignment, your directory should have the following structure:

- `<username1>_<username2>_ps3/`
  - `data/`
    - `regression_train.csv`
    - `regression_test.csv`
  - `source/`
    - `regression.py` (with your modifications)
  - `writeup.pdf`
  - `regression.pdf` (pdf printout of `regression.py`)

Package this directory as a single `<username1>_<username2>_ps3.zip` file, and submit the archive. Additionally, to aid the staff in grading, submit your `pdf`'s as separate files.

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

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 $h_{\boldsymbol{\theta}}(x)$ that best approximates $f(x)$.

---

code and data

- code : `regression.py`
- data : `regression_train.csv`, `regression_test.csv`

A note about ***numpy***: This is likely the first time that many of you are working with `numpy` and matrix operations within a programming environment. Every semester, some students find figuring out `numpy` to be the hardest part of this assignment! It is a good skill to pick up though since you will inevitably use `numpy` if you plan to do math in Python, e.g. for machine learning.

For the uninitiated, you may find it useful to work through a `numpy` tutorial first.[1] 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[2]. 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 $1 \times N$.

1 Visualization [1 pts]

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) **(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 and polynomial regression in predicting the data?

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

[1] Try out `SciPy`'s tutorial (http://wiki.scipy.org/Tentative_NumPy_Tutorial), or use your favorite search engine to find an alternative. Those familiar with Matlab may find the "Numpy for Matlab Users" documentation (http://wiki.scipy.org/NumPy_for_Matlab_Users) more helpful.

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

2 Linear Regression [11 pts]

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

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

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

$$\boldsymbol{y} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{pmatrix}, \qquad \boldsymbol{X} = \begin{pmatrix} — & (\boldsymbol{x}^{(1)})^T & — \\ — & (\boldsymbol{x}^{(2)})^T & — \\ & \vdots & \\ — & (\boldsymbol{x}^{(n)})^T & — \end{pmatrix}, \qquad \boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{pmatrix}$$

and each example $\boldsymbol{x}^{(i)} = \left(1, x_1^{(i)}, \ldots, x_d^{(i)}\right)^T$.

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

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

(b) **(1 pts)** Note that to take into account the intercept term (θ_0), we can add an additional "feature" to each example and set it to one, e.g. $x_0^{(i)} = 1$. 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.

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

(d) **(4 pts)** One way to solve linear regression is through stochastic gradient descent (SGD).

Recall that the parameters of our model are the θ_j values. These are the values we will adjust to minimize cost $J(\boldsymbol{\theta})$. In SGD, each iteration runs through the training set and performs the update[3]

$$\theta_j \leftarrow \theta_j - \alpha\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)}\right)x_j^{(i)} \quad \text{(simultaneously update } \theta_j \text{ for all } j\text{)}.$$

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

- **(1 pts)** As we perform gradient descent, it is helpful to monitor the convergence by computing the cost. Complete `PolynomialRegression.cost(...)` to calculate $J(\boldsymbol{\theta})$.
- **(2 pts)** Next, implement the gradient descent step in `PolynomialRegression.fit_SGD(...)`. The loop structure has been written for you, and you only need to supply the updates to $\boldsymbol{\theta}$ and the new predictions $\hat{y}$ within each iteration.

[3]The handout uses α as in class, but the code uses η (`eta`) as in `scikit-learn`.

Hint: A good way to verify that gradient descent is working correctly is to look at the value of $J(\boldsymbol{\theta})$ 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 $J(\boldsymbol{\theta})$ 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.

- **(1 pts)** So far, you 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. How do the coefficients compare? How quickly does each algorithm converge?

(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.

- **(2 pts)** Implement the closed-form solution `PolynomialRegression.fit(...)`.
- **(1 pts)** How do the coefficients compare to those obtained by SGD?
- **(1 pts)** Use Python's `time` module to measure the run time of the two approaches. How do the run times compare?

(f) **(1 pts)** Finally, find a learning rate η for SGD that is a function of k (the number of iterations) and converges to the same solution yielded by the closed-form optimization. For this problem, we care more about your learning rate than about whether you match the closed-form solution (that is, your solution may not match exactly, but you should be able to get pretty close).

Update `PolynomialRegression.fit_SGD(...)` with your proposed learning rate. How long does it take the algorithm to converge with your proposed learning rate?

3 Polynomial Regression [6 pts]

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

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

(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 example.

(h) **(2 pts)** 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 $n-1$. 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

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

where

$$\mathbb{E}[\boldsymbol{\theta}] = \frac{1}{2}\sum_{i=1}^{n}\left(\sum_{j=0}^{m}\theta_j\phi_j(\boldsymbol{x}^{(i)}) - y^{(i)}\right)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)}\right)^2$$

and n is the number of examples.[4]

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

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

(i) **(3 pts)** For $m = 0, \ldots, 10$, 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.

[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, $k = m+1$.

4 Extra Credit: Regularization [+2 pts]

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

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2}||\boldsymbol{\theta}_{[1:m]}||^2,$$

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

(j) **(+1 pts)** Modify `PolynomialRegression.fit(...)` to incorporate L_2-regularization.

(k) **(+1 pts)** Use your updated solver to find the coefficients that minimize the error for a tenth-degree polynomial ($m = 10$) 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?