Introduction to Artificial Intelligence
Homework 4 — Regression

Instructions

Submission

Submission will be handled using Gradescope, as will announcements regarding due dates or any changes to this assignment. This assignment will contain some coding and some written questions. In order to keep things reasonable for your TAs to grade, please submit **one** zipfile containing a folder with **all** your code, data files, and a **single** pdf with your answers to the written questions. This zipfile should have the following format: `hw4-username.zip`, replacing "username" with your username (for example "hw4-bhrolenok3.zip").

For this assignment, your submission should include:

- `hw4-answers.pdf` A PDF of your answers to the written questions. You can use whatever software you like to create your report, but what you turn in must be a PDF (no .docx, .txt, .rtf...). We recommend LaTeX (which is how this handout was created) because it generates professional looking reports, and is a good tool to learn. For a simple example, the source that generated this file is included with the template code.

- `hw4.py` The template file, modified to include your code.

- `README` Any special instructions for running your code, (library versions, network access, etc) **plus** any external sources you used to help you with this assignment. This includes other students and websites. When in doubt, cite it!

Collaboration

Please see the collaboration note on the syllabus.

Coding

Linear regression is probably one of the simplest and yet most widely used learning algorithms that we'll discuss in this class. Partly, this is because it is an extremely simple algorithm, both to implement and to analyze. In the rest of this exercise, you'll implement linear regression several different ways, and explore a simple but powerful extension.

Setting up and code template

A short template has been provided for you in the file `hw4.py`. The only dependencies are NumPy[1] and matplotlib[2], and these are fairly straightforward libraries to install on any system, Linux, Mac, and Windows.

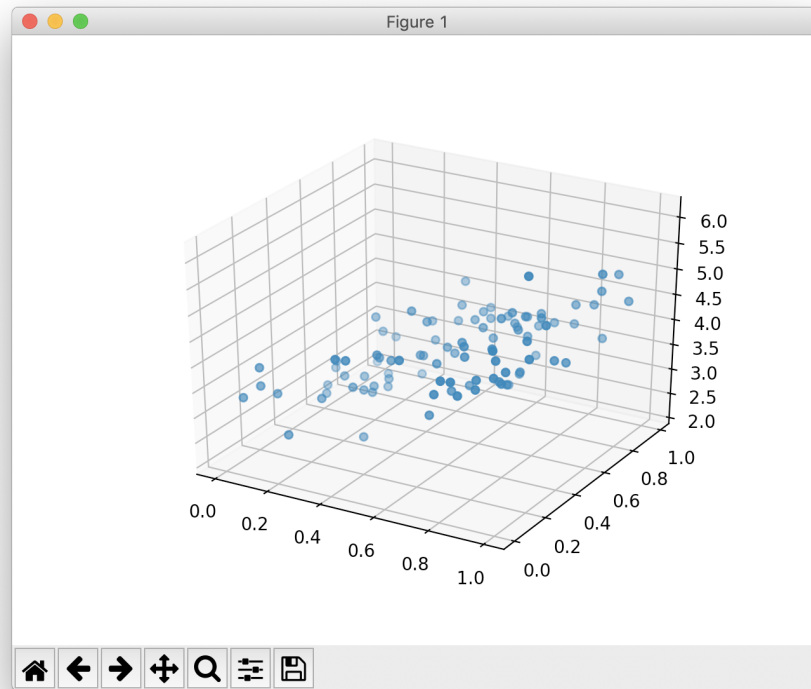[1] `https://numpy.org/`
[2] `https://matplotlib.org`

Figure 1: The what the demo output should look like.

You can use whatever development environment you wish, but we recommend something simple, like your favorite text editor, the command line, and `virtualenv`[3].

Once you think you've got everything set up, you can test by running `hw4.py` from the command line. This should display the `data/2D-noisy-lin.txt` as a 3D scatter plot, which should look like Figure 1. If you encounter any errors, or the figure looks significantly different, debug your installation first since the rest of this assignment depends on these libraries.

After getting everything working, start digging in to the code in the template. Most functions are thoroughly commented, even those you don't need to modify to complete the assignment. You might find some snippets that are useful elsewhere in the assignment.

Linear Regression - closed form solution

Now you should be ready to implement Linear Regression. First, recall the equation for our general linear model in matrix/vector notation

$$h_\theta(\mathbf{x}) = \mathbf{x}^\top \theta$$

where we're using convention that $x_0 = 1$ (we prepend 1 to our feature vector). This tells us how to use θ to predict y for a given x. To actually find θ we can construct a system of linear equations out of our training

---

[3]`https://virtualenv.pypa.io/`

data like so:

$$\mathbf{X} = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} & \cdots & x_D^{(1)} \\ x_0^{(2)} & x_1^{(2)} & \cdots & x_D^{(2)} \\ \vdots & & \ddots & \\ x_0^{(N)} & x_1^{(N)} & \cdots & x_D^{(N)} \end{bmatrix} = \begin{bmatrix} - & \mathbf{x}^{(1)\top} & - \\ - & \mathbf{x}^{(2)\top} & - \\ & \vdots & \\ - & \mathbf{x}^{(N)\top} & - \end{bmatrix}$$

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix}$$

$$\mathbf{y} = \mathbf{X}\theta$$

which we can use to find an estimate for the parameters, $\hat{\theta}$:

$$\hat{\theta} = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y} \tag{1}$$

Note that since this is just a bunch of matrix multiplications, transposes, and one matrix inverse, we can actually code this as a single line using numpy, although you can break it out into multiple steps to make it obvious and easier to debug. Look at the documentation for `numpy.dot(...)`, `numpy.transpose(...)`, and `numpy.linalg.inv(...)`, play around with them for simple 2-by-2 matrices until you understand them, then use them to build up equation 1 in the function `linreg_closed_form(...)`. You can test your work on the provided datasets or data you make yourself, and compare with the results that `numpy.linalg.lstsq(...)` produces. (Note: to get credit for this part be sure to actually implement the equation, don't use the output of `numpy.linalg.lstsq(...)`). You can also visualize the model you are finding with the `vis_linreg_model(...)` function.

Gradient Descent

What happens when we have a lot of data points or a lot of features? Notice we're computing $\left(\mathbf{X}^\top \mathbf{X}\right)^{-1}$ which becomes computationally expensive as that matrix gets larger. In the section after this we're going to need to be able to compute the solution for some really large matrices, so we're going to need a method that doesn't compute this inverse.

Gradient Descent is one such method, one that's used in a variety of algorithms in ML. The basic outline of the gradient descent method is to "follow" the gradient of the loss function, $\nabla_\theta \mathcal{J}_S(h_\theta)$, to a minimum for the parameters, θ. First lets define the loss function as the squared-loss:

$$\mathcal{J}_S(h_\theta) = \frac{1}{2N} \sum_{i=1}^{N} \left(h_\theta(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}\right)^2 \tag{2}$$

Implement equation 2 in the function `loss(...)`. You can check that it works by using `numpy.linalg.lstsq(...)` on the provided datasets. Note that the residuals returned by `numpy.linalg.lstsq(...)` are the sum of squared errors, so you will have to scale appropriately.

We can use Gradient Descent on this loss function to find the optimal parameters for our linear model by iteratively updating θ, adding in a small bit of the negative of the gradient. Here's a formal description of the algorithm:

Data: initial estimate $\hat{\theta}^{(0)}$, loss function $\mathcal{J}_S(h_\theta)$, number of iterations K, learning rate α
Result: estimate of optimal parameters, $\hat{\theta}^{(K)}$
for $k \leftarrow 1$ **to** K **do**
$\quad \mid \quad \hat{\theta}^{(k)} \leftarrow \hat{\theta}^{(k-1)} - \alpha \nabla \mathcal{J}_S(\hat{\theta}^{(k-1)})$
end

3

As we showed in class, for our linear model and using equation 2 for the loss, we can write the gradient as:

$$\nabla \mathcal{J}_S(h_\theta) = \frac{1}{N} \left(\mathbf{X}^\top \mathbf{X} \theta - \mathbf{X}^\top \mathbf{Y} \right) \tag{3}$$

Implement Gradient Descent using equation 3 in the function `linreg_grad_desc(...)`. You can check that it works by comparing your answers for the closed form solution or `numpy.linalg.lstsq(...)`. The default values for `alpha` and `num_iters` should be enough for gradient descent to converge to the same answers for the linear datasets, but you may want to test for higher or lower values to see how it affects the results. Again, it will probably be helpful to use `vis_linreg_model(...)` to visualize the resulting models.

Random Fourier Features

Linear Regression, unsurprisingly enough, can only fit **linear** data with any degree of accuracy. However, as we discussed in the class it's fairly easy to extend Linear Regression by transforming the data using a set of K non-linear feature functions, which we'll denote $\Phi = \{\phi_k(\mathbf{x})\}_{k=1}^K$.

There are many possible ways to choose the set Φ^4, but there are actually a few methods that extend Linear Regression to fit models from almost any class of functions. One particularly simple (to implement) method is called *Random Fourier Features* which is described by Rahimi and Recht (2008). The key idea is that any "well behaved" function can be represented by its Fourier transform, which is basically a sum of an infinite number of sin and cos functions with different amplitudes and frequencies. The trick is that we can *approximate* this infinite sum with a *finite* number of samples, as long as we sample appropriately at randomly selected frequencies. The way this is implemented is by using the following set of basis functions:

$$\phi_k(\mathbf{x}) = \cos(\omega_k x + b_k), \qquad k = 1, \dots, K$$

So using using this set of basis functions, and randomly selecting ω_k and b_k, gives us a problem that's linear in the features (so we can use linear regression), and which should let us fit *almost* any function (where *almost* includes any function for which the Fourier inversion theorem holds). This also means that as we add more and more features (as we increase K), we should be able to more and more accurately estimate any crazy non-linear function we like! The proof that this is true, and the details of how exactly to do this sampling for ω_k and b_k are in the cited paper, which you'll need to read in order to implement the function `random_fourier_features(...)`. The paper gives different ways of sampling for different "kernels;" we'll talk about these in more detail later, but for now use the method for the Gaussian kernel.

The math in the paper may be a bit advanced, but the actual code that you have to write is minimal (less than 5 lines total), as a lot has been filled in for you, there and in the helper functions. Perhaps the most complicated part is getting the shape of the matrix right for the helper funciton `apply_RFF_transform(...)` to work with the code as written. The helper function is expecting a 1D array for **b** and a D-by-K matrix for the ω's:

$$\Omega = \left[\begin{array}{cccc} | & | & & | \\ \omega_1 & \omega_2 & \cdots & \omega_K \\ | & | & & | \end{array} \right]$$

The comments in that function also have good pointers for which `numpy` methods you'll want to use for the sampling.

A note about testing RFF

Since you are randomly sampling the transform for Random Fourier Features, your results may vary from one run to the next. Be sure to test your work, using the provided non-linear datasets like `1D-exp-*.txt` and

---

[4]and if you have any insight into the kinds of functions that generated your training data, it's a good idea to start with those.

`1D-quad-*.txt`, multiple times to get a good idea of how this method performs. As with gradient descent, the default parameters should produce reasonable results, but you will want to change these settings to get a good feel for what's going on, as well as to help with debugging.

Questions

1. Linear Regression

   (a) Use your closed form implementation to fit a model to the data in `1D-no-noise-lin.txt` and `2D-noisy-lin.txt`. What is the loss of the fit model in both cases? Include plots for both. Note: for some of these datasets the y-intercept will be non-zero, make sure you handle this case!

   (b) What happens when you're using the closed form solution and one of the features (columns of $\mathbf{X}$) is duplicated? Explain why. Note: you may want to test this out with your code as a useful *first step*, but you should think critically about what is happening and why.

   (c) Does the same thing happen if one of the training points (rows of $\mathbf{X}$) is duplicated? Explain why.

   (d) Does the same thing happen with Gradient Descent? Explain why.

2. Gradient Descent

   (a) Now use your Gradient Descent implementation to fit a model to `1D-no-noise-lin.txt` with `alpha=0.05`, `num_iters=10`, and `initial_Theta` set to the zero vector. What is the output (the full list of (θ, loss) tuples for each of the 10 iterations)?

   (b) Using the default parameters for `alpha` and `num_iters`, and with `initial_Theta` set to $\mathbf{0}$, do you get the same model parameters as you did with the closed form solution? the same loss? Report this for both `1D-no-noise-lin.txt` and `2D-noisy-lin.txt`.

   (c) Find a set of values of the learning rate and iterations where you get the same answers and a set of values where the answers are noticeably different. Explain what's going on in both cases, and for both datasets `1D-no-noise-lin.txt` and `2D-noisy-lin.txt`.

3. Random Fourier Features

   (a) Use your Random Fourier Features implementation to fit models for `1D-exp-samp.txt`, `1D-exp-uni.txt`, `1D-quad-uni.txt`, and `1D-quad-uni-noise.txt`, each with 3 different values for K: small, medium, and large. The small value should noticeably under-fit the data, the large value should fit almost perfectly, and the medium value should be somewhere in between. Report the values for K and include graphs for each of the four datasets (so you should report 12 values of K and have 12 graphs in total).

   (b) **EXTRA CREDIT**: What happens with Random Fourier Features if you try to predict a value far away from any points in the training dataset? Modify the plotting functions (or write your own) and show what the model looks like far away from the training points. Describe what you are seeing qualitatively, compare the model's output with the "true" output quantitatively, and explain why this is happening. You may want to run these experiments multiple times to get a better understanding of what is going on.

References

Rahimi, Ali, and Benjamin Recht. "Random features for large-scale kernel machines." Advances in neural information processing systems. 2008.