CM146, Fall 2017
Problem Set 2: Perceptron and regression
Due Nov 6,2017 at 11:59 pm

Submission

- Submit your solutions electronically on the course Gradescope site as PDF files.
- If you plan to typeset your solutions, please use the LaTeX solution template. If you must submit scanned handwritten solutions, please use a black pen on blank white paper and a high-quality scanner app.

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

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.

(a) AND

(b) XOR

2 Logistic Regression [10 pts]

Consider the objective function that we minimize in logistic regression:

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

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

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

(c) 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 $\boldsymbol{z}$.

3 Locally Weighted Linear Regression [10 pts]

Consider a linear regression problem in which we want to "weight" different training instances differently because some of the instances are more important than others. Specifically, suppose we want to minimize

$$J(\theta_0, \theta_1) = \sum_{n=1}^{N} w_n \left(\theta_0 + \theta_1 x_{n,1} - y_n\right)^2.$$

Here $w_n > 0$. In class, we worked out what happens for the case where all the weights (the w_n's) are the same. In this problem, we will generalize some of those ideas to the weighted setting.

(a) Calculate the gradient by computing the partial derivatives of J with respect to each of the parameters (θ_0, θ_1).

(b) Set each partial derivatives to 0 and solve for θ_0 and θ_1 to obtain values of (θ_0, θ_1) that minimize J.

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

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.[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 $1 \times N$.

Visualization [1 pts]

As we learned last week, 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) 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?

[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

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:

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

`regression.py` 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, $\left(1, x_{n,1}, x_{n,1}^2, \ldots, x_{n,1}^m\right)^T$. Setting $m = 1$ instantiates an object where the feature vector for instance n, $\left(1, x_{n,1}\right)^T$.

(b) 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. $x_{i,0} = 1$. 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.

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

(d) 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 $J(\boldsymbol{\theta})$. In gradient descent, each iteration performs the update

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

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')
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 ealier if the value of the objective function is unchanged across consecutive iterations.
  - We will use a fixed step size.

- So far, you have used a default learning rate (or step size) of $\eta = 0.01$. Try different $\eta = 10^{-4}, 10^{-3}, 10^{-2}, 0.0407$, 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?

(e) 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? How do the coefficients and the cost compare to those obtained by GD? How quickly does the algorithm run compared to GD?

(f) 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 long does it take the algorithm to converge with your proposed learning rate?

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) 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.

(h) 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.[3]

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

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

(i) 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 justify your answer.

[3]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, $k = m+1$.