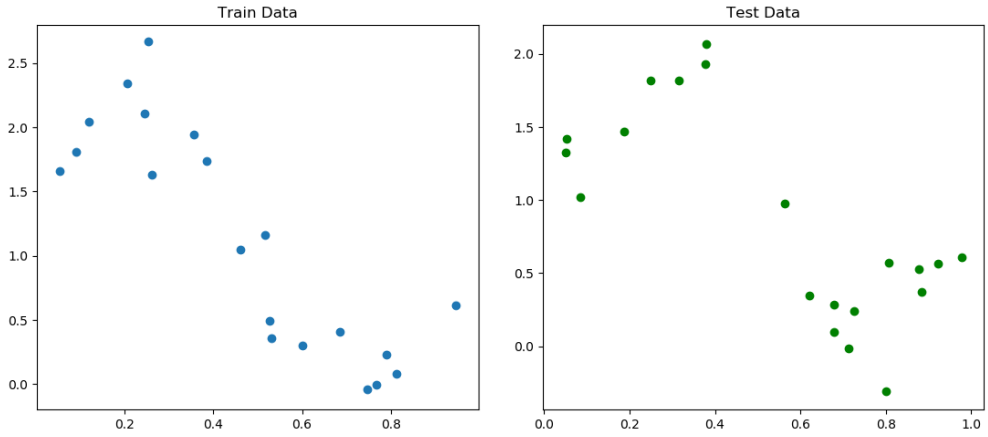
Problem Set 2: Implementation of Polynomial Regression

Rodrigo Valle

10 November 2018

NOTE: For the code samples in this assignment I've used Python 3.

Visualization



a) The training data looks like it could be estimated well by a linear function, but it looks like fitting a 3rd or 5th degree polynomial function might yield even better results. Actually, if I had to guess, this looks like a sine wave with some noise added to it!

Linear Regression

b) Modify `PolynomialRegression.generate_polynomial_features(...)` to create the matrix X for a simple linear model.

```
def generate_polynomial_features(self, X):
    N, d = X.shape  # X is of shape (N,1)
    Phi = np.concatenate([np.ones((N,1)), X], axis=1)
    return Phi
```

c) Complete `PolynomialRegression.predict(...)` to predict $\mathbf{y}$ from X and $\mathbf{w}$.

```
def predict(self, X):
    ...
    X = self.generate_polynomial_features(X)
    y = np.matmul(X, self.coef_)
    return y
```

d) Gradient Descent

- Implement `PolynomialRegression.cost(...)` to calculate $J(\mathbf{w})$.

```
def cost(self, X, y):
    X = self.generate_polynomial_features(X)
    cost = np.sum(np.power(np.matmul(X, self.coef_) - y, 2))
    return cost
```

- Implement the gradient descent step in `PolynomialRegression.fit_GD(...)`.

```
def fit_GD(self, X, y, eta=None, eps=0, tmax=10000, verbose=False):
    ...
    X = self.generate_polynomial_features(X)
    n, d = X.shape
    self.coef_ = np.zeros(d)
    err_list = np.zeros((tmax, 1))

    for t in range(tmax):
        ...
        y_pred = np.matmul(X, self.coef_)
        gradient = 2 * eta * np.matmul(X.T, y_pred - y)
        self.coef_ = self.coef_ - gradient

        # track error
        # hint: you cannot use self.predict(...)
        # to make the predictions
        err_list[t] = np.sum(np.power(y - y_pred, 2)) / n
        ...
```

```
        ...
        return self
```

- Try varying the learning rate (η) and compare the learned coefficients, number of iterations until convergence, and the final value of the objective function.

η	Iterations	Coefficients ($\mathbf{w}$)	$J(\mathbf{w})$
10^{-4}	10000	$\begin{pmatrix}2.270 & -2.461\end{pmatrix}^T$	4.086
10^{-3}	7021	$\begin{pmatrix}2.446 & -2.816\end{pmatrix}^T$	3.912
10^{-2}	765	$\begin{pmatrix}2.446 & -2.816\end{pmatrix}^T$	3.912
0.0407	10000	$\begin{pmatrix}-9.405 \times 10^{18} & -4.652 \times 10^{18}\end{pmatrix}^T$	2.711×10^{31}

The coefficients, when GD was given the opportunity to converge, are around $\begin{pmatrix}2.45 & -2.82\end{pmatrix}^T$.

At $\eta = 10^{-4}$, the coefficients that are close to optimal, but GD still requires more iterations/steps to finish converging and get closer to the optimum cost.

At $\eta = 10^{-3}$ and $\eta = 10^{-2}$, GD converges to very similar weight vectors, equivalent to 5 decimal places. $\eta = 10^{-3}$, predictably, uses an order of magnitude more iterations to converge.

At $\eta = 0.0407$, we see that GD actually diverges from the solution by taking steps too large to descend towards the optimum. The coefficients contain whatever point the algorithm was currently bouncing from when it hit the hard upper limit of 10,000 iterations.

e) Closed Form Solution

- Implement the closed form solution `PolynomialRegression.fit(...)`

```
def fit(self, X, y):
    # note: '@' is shorthand for np.matmul(...)
    X = self.generate_polynomial_features(X)
    self.coef_ = np.linalg.pinv(X.T @ X) @ X.T @ y
```

- 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?

  The closed form solution for linear regression gives the values of the coefficients as:

$$\mathbf{w} \approx \begin{pmatrix} 2.446 \\ -2.816 \end{pmatrix}$$

Which are the same values to which gradient descent converged previously. The cost is predictably identical to the final cost computed by gradient descent: 3.912.

A timed comparison of the two methods was also performed using python's `timeit` module, taking the average wall execution time out of 10,000 trials.

Closed Form	Iterative (Gradient Descent)
$9.653 \times 10^{-5}\,\text{s}$	$1.25 \times 10^{-2}\,\text{s}$

f) Update `PolynomialRegression.fit_GD(...)` with the learning rate $\eta_k = \frac{1}{1+k}$. How long does it take the algorithm to converge with this learning rate?

```
def fit_GD(self, X, y, eta=None, eps=0, tmax=10000, verbose=False):
    eta_input = eta
    ...
    for t in range(tmax):
       if eta_input is None:
           eta = 1 / (1 + t)
       ...
    ...
    return self
```

With this dynamic learning rate, gradient descent takes 1719 iterations to converge, which is about 1000 more iterations than setting $\eta = 0.01$ from before, but roughly 5000 fewer iterations than setting $\eta = 0.001$.

The advantage of this dynamic learning rate is that GD can spend a little more time scanning for an appropriate learning rate automatically as it iterates instead of a human trying to find the problem-specific learning rate by hand.

Polynomial Regression

g) Update `PolynomialRegression.generate_polynomial_features(...)` to create an $m + 1$ dimensional features vector for each instance.

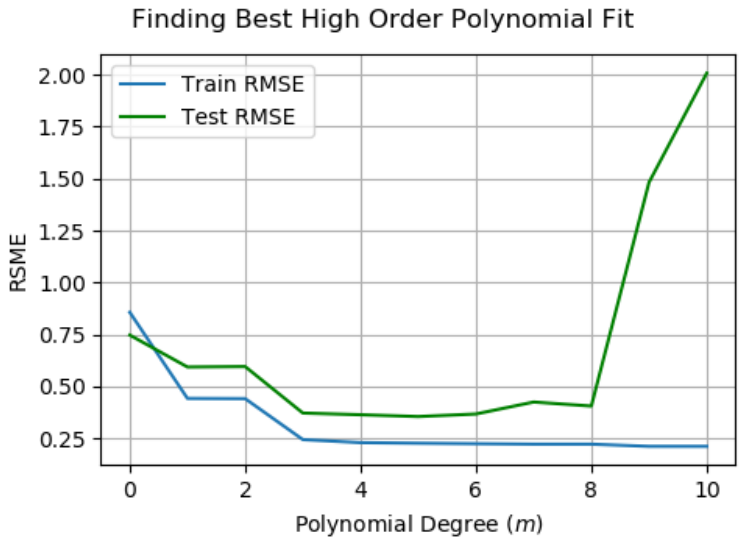
```
def generate_polynomial_features(self, X):
    phi = np.concatenate([X**i for i in range(self.m_ + 1)], axis=1)
    return phi
```

h) Why do you think we might prefer RMSE as a metric over $J(\mathbf{w})$? Implement `PolynomialRegression.rms_error(...)`.

RMSE is essentially calculating the standard deviation of the distribution of every example's distance from the model's prediction. Optimizing the RMSE will yield the same result as optimizing the variance, which is what the cost function J computes, so from a training perspective there's not much difference. However, the key advantage is that the standard deviation is in the same units as the mean, so it's much easier for a human to conceptualize and compare.

```
def rms_error(self, X, y):
    n, d = X.shape
    error = np.sqrt(self.cost(X, y) / n)
    return error
```

i) Comparing different choices of the polynomial degree m yields the following plot:



From the plot, fitting a polynomial of degree 5 reported the lowest error on the test data with a test error of 0.3551 and a train error of 0.2268.

We can clearly see on the plot that the model begins severely overfitting past an 8th degree polynomial as training error continues to decline but a sharp increase in test error is observed.