Regression Week 2 – Multiple Regression

Polynomial Regression

From Wikipedia: Polynomial regression is a form of linear regression in which the relationship between the independent variable x and the dependent variable y is modeled as an nth degree polynomial. Although polynomial regression fits a nonlinear model to the data, as a statistical estimation problem it is linear, in the sense that the regression function E(y | x) is linear in the unknown parameters that are estimated from the data. For this reason, polynomial regression is considered to be a special case of multiple linear regression.

In polynomial regression, we have a single input with multiple features that are some power of the input. *x* is a feature of our input. Each *w* is the coefficient associated with that input. These are combined with a polynomial and an error term;

$$y = w_o + w_1 x + w_2 x^2 + \cdots + w_p x^p + \varepsilon$$

$x^0 = 1$ is called the constant feature
x^1 is feature 2
x^2 is feature 3
…
x^p is feature p+1

w^0 is parameter 1
w^1 is parameter 2
w^2 is parameter 3
…
w^p is parameter p+1

Polynomial Regression is just one case of the more general notion of multiple regression. There are other functions of x that can be modeled. So we need to rewrite the multiple regression model in more general terms using a basis expansion.

$$y = w_0 h_0(x) + w_1 h_1(x) + \cdots + w_D h_D(x) + \varepsilon$$

or more compactly, for data input *i*;

$$y_i = \sum_{j=0}^{D} w_j h_j(x_i) + \varepsilon_i$$

$h_j(x)$ is some function of the input that represents our *j-th* feature.
w_j is the coefficient or weight of the *j-th* feature.

For example
$h_0(x)$ feature 1 is usually a constant and often 1.
$h_1(x)$ feature 2 is commonly used for a linear component, eg *x*.
$h_2(x)$ feature 3 is commonly used for a quadratic component, eg x^2 or for some other arbitrary function in x, like sin(x).
…
$h_D(x)$ feature D+1

The *h(x)* vector is what comes our of the feature extraction part of the ML workflow.

Notation used in the rest of this course for specifying the input data (like a house) and the features on the input data (like square feet and number of bathrooms)

Output:	y	a scalar, it is the observed output value
Inputs:	$\vec{\boldsymbol{x}} = (\vec{\boldsymbol{x}}[1], \vec{\boldsymbol{x}}[2], \vec{\boldsymbol{x}}[3], \dots, \vec{\boldsymbol{x}}[d])$	the input values a d-dimensional **vector** of scalar

$\vec{\boldsymbol{x}}[j]$ The j-th input value (a scalar) like square feet of the house

$\overrightarrow{\boldsymbol{Bold}}$ is used to indicate a vector in the videos. In these notes I am indicating a vector using the arrow above the value.

Our data sets will have many output values and their associated input values. Given a dataset with N output values each associated with d input values;

$\vec{\boldsymbol{y}}$	The vector of N observed output values
$\overrightarrow{\boldsymbol{y_i}}$	The i-th observation in the data
$h_j(\vec{\boldsymbol{x}})$	The j-th feature (a scalar), Could be just the input value as in a simple hyperplane model, Or more generally this could be could be any function of one or more on the input values of **x**, for instance we could take the logarithm of one input and add it to another input.
$\overrightarrow{\boldsymbol{x_i}}$	The input vector for the i-th data point in the data set, like the i-th house in a dataset of houses
$\overrightarrow{\boldsymbol{x_i}}[j]$	The j-th input of the i-th data point in the dataset, like the number of bathrooms of the i-th house in the dataset

More Notation

N	Capital-N will represent that number of observations in the dataset.
d	small-d represents the number of inputs, so it is the length of the input vector $\overrightarrow{\boldsymbol{x_i}}$.
D	Capital-D represents the number of features we extract from those inputs (and since there is a coefficient for each feature, it determines the number of coefficients in the model, which is D+1 because there is the constant feature implied).

Simple Hyperplane Model

This is like the simple linear model, but in multiple dimensions. We assume that each observation is a function of the inputs directly, not of some function of the inputs. (the following presumes one-based indexing of $\vec{\boldsymbol{x}}$).

$$y_i = w_0 + w_1\overrightarrow{\boldsymbol{x_i}}[1] + w_2\overrightarrow{\boldsymbol{x_i}}[2] + \cdots + w_d\overrightarrow{\boldsymbol{x_i}}[d] + \varepsilon_i$$

Using a more general D-dimensional curve to model the general notion of multiple regression;

General Multiple Regression Model

$$y = w_0 h_0(\vec{x_i}) + w_1 h_1(\vec{x_i}) + \cdots + w_D h_D(\vec{x_i}) + \varepsilon$$

$$y_i = \sum_{j=0}^{D} w_j h_j(\vec{x_i}) + \varepsilon_i$$

Interpreting the Coefficients in the Multiple Regression Model

Multiple Linear Regression

For a model with **multiple linear features**, we can look at any single coefficient by fixing the other coefficients and only letting the single coefficient vary. In effect, we are looking at that single coefficient as a simple linear regression because everything else is fixed. So we can interpret it as a line and it's slope is the change in the output per unit change of the input for that term in the model, but only for a given set of fixed inputs for the other terms.

So for instance, if we think of a model for predicting house prices that uses floor space in square feet and the number of bathrooms as features, then if we fix the square feet at say 2500 square feet, we can interpret the number of bathrooms coefficient as the change in house price for unit change in the number of bathrooms for a house of 2500 square feet. Note that we might think the coefficient should be positive; as the number of bathrooms increase the house value would increase, we might be wrong. Because we have fixed the square feet, too many bathrooms is a negative aspect of the house because most of the floor space in the house ends up in bathrooms. What this implies is that square feet and number of bathrooms are related and affect each other, so fixing one and varying the other can yield unexpected results.

Polynomial Regression

Here we have a single input and our features are different powers of the input. In this case, we can't interpret the coefficients as in a linear model, because there is only one feature to fix. If we fix the one feature then the model produces a scalar. So we can't interpret individual coefficients of a polynomial model in the same way we can with a linear model.

Multiple Regression Algorithms

As with the simple linear regression, we will use two algorithms; the closed form solution and gradient descent.

Matrix Notation

First, we can rewrite the summation for a single observation as the inner product of two vectors;

$$y_i = \sum_{j=0}^{D} w_j h_j(\vec{x_i}) + \varepsilon_i$$

Generally when we have a vector we think of it as a column vector. So if we want an inner product, we want a row x a column vector, so we must transpose the first vector to get the row vector. So for observation *i* the model is;

$$y_i = [w_0 \quad \dots \quad w_D] \begin{bmatrix} h_0(\vec{x_i}) \\ \dots \\ h_D(\vec{x_i}) \end{bmatrix} + \varepsilon_i$$

or more compactly

$$y_i = \overrightarrow{w^T h(\vec{x_i})} + \varepsilon_i$$

as you can see the vector notation over w and h is actually getting in the way. So we will drop it, but remember that when we use w without a subscript we actually mean the vector $\vec{w}$ with D+1 elements, from w_0 to w_D, and when we use $h(x_i)$ we meant the vector $\vec{h}$ with D+1 elements blah blah blah.

So simply

$$y_i = w^T h(\vec{x_i}) + \varepsilon_i$$

And we can write it equivalently by commuting terms;

$$y_i = h(\vec{x_i})^T w + \varepsilon_i$$

That will produce the same result, which of course is a scalar.

Our data contains many observations. We can take the vector notation and extend it into matrix notation to get a compact equation that represents all of our observations and inputs;

$$\begin{bmatrix} y_1 \\ \dots \\ y_N \end{bmatrix} = \begin{bmatrix} h_0(x_1) & \dots & h_D(x_1) \\ \dots & \dots & \dots \\ h_0(x_N) & \dots & h_D(x_N) \end{bmatrix} \begin{bmatrix} w_0 \\ \dots \\ w_D \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \dots \\ \varepsilon_N \end{bmatrix}$$

which is written as

$$\vec{y} = H\vec{w} + \vec{\varepsilon}$$

Or just

$$y = Hw + \varepsilon$$

N	capital-N is the number of observations in the dataset
D	capital-D is the number of features extracted from the dataset
y	lowercase-y is the vector of all N output values
H	capital H represents the matrix of the D feature terms for all N observations. So it is an NxD matrix with N rows (one for each observation) and D columns (one for each feature).
w	lowercase w is the vector of coefficients for the model.
ε	epsilon is the vector of the N error terms, one for each observation.

The Cost of the Fit for Multiple Regression

The cost of fit we have been using is the Residual Sum of Squares (RSS), which is the difference between the observed value and the predicted value squared for each observation and then all of those squared differences summed. The predicted value for the i-th observation in a multiple regression is given by;

$$\hat{y}_i = h(\vec{x_i})^T w$$

So the residual sum of squares for all observations using the coefficients in $\vec{w}$ is given by;

$$RSS(w) = \sum_{i=1}^{N} (y_i - h(\vec{x_i})^T w)^2 = (y_i - Hw)^2 = (y_i - Hw)^T (y_i - Hw)$$

That final piece reflects the fact that the square of a vector is equal to the inner product of the transpose of the vector and itself. You can draw it out to see why.

Gradient of the Cost of Fit for Multiple Regression

$$\nabla RSS(w) = \nabla (y - Hw)^2 = -2H^T (y - Hw)$$

Approach 1: Closed Form Solution

Set $\nabla RSS(w) = 0$ and solve for w.

1)	$\nabla RSS(w) = -2H^T(y - H\hat{w}) = 0$	set gradient of RSS to zero
2)	$-2H^T y + 2H^T H\hat{w} = 0$	distribute the $-2H^T$
3)	$2H^T H\hat{w} = 2H^T y$	add $2H^T y$ to each side
4)	$H^T H\hat{w} = H^T y$	divide each size by 2
5)	$(H^T H)^{-1} H^T H\hat{w} = (H^T H)^{-1} H^T y$	multiply by inverse $H^T H$
6)	$\hat{w} = (H^T H)^{-1} H^T y$	closed form estimated coefficients

Aside

In step (5) we multiply both sides by the inverse of $H^T H$. $H^T H$ evaluates to a matrix. Multiplying a matrix by it's inverse results in the identity matrix:

$$A^{-1} A = I$$

The product of the Identity Matrix times a Matrix is the Matrix. Similarly, the product of the identity matrix times a vector is the vector.

$$IA = A$$
$$I\vec{v} = \vec{v}$$

So our closed form solution is:

$$\hat{w} = (H^T H)^{-1} H^T y$$

Examining the $(H^T H)^{-1}$, remember that H is the matrix of the D feature terms for all N observations. So it is an NxD matrix with N rows (one for each observation) and D columns (one for each feature). Thus its transpose has D rows and N columns. When we multiple a DxN matrix with an NxD matrix we get a square DxD matrix. (This takes N multiplications and N-1 additions for each of the results in the DxD matrix. So for a data set with 50 observations and 10 features, DxD = 10x10 = 100 results, N = 50 multiplications to calculate each result, so there are NxDxD = 50x100 = 5,000 multiplications and (N-1)xDxD = 4900 additions).

In general, we can take the inverse of a square matrix if it's rows and columns are linearly independent. In our case, this is true in most cases where the number of linearly independent observations is greater than the number of features.

So $(H^T H)^{-1}$, if it has an inverse, will be a DxD square matrix where D is the number of features in our model. The problem is that taking the **matrix inverse is O(D³) complex**, which means as the number of features grows, the number of operations necessary to take the inverse of the DxD matrix grows as it's cube. This can make the closed form solution **computational impractical** for large models.

Approach 2: Gradient Descent

Since inverting large matrices can be computationally expensive, we can choose to use gradient descent in these cases. Remember that gradient descent is an iterative algorithm that converges on the best estimated set of coefficients.

While $\left\|\nabla RSS(\widehat{w}^{(t)})\right\| < \epsilon$

$$\widehat{w}^{(t+1)} \leftarrow \widehat{w}^{(t)} - \eta \nabla RSS(\widehat{w}^{(t)})$$

The gradient for the residual sum of squares for the estimated parameters is given by;

$$\nabla RSS\left(\widehat{w}^{(t)}\right) = -2H^T\left(y - H\widehat{w}^{(t)}\right) = -2H^T\left(y - \hat{y}^{(t)}\right)$$

Also member that Hw is our estimated output, $\hat{y}$. So $y - \hat{y}$ is the residual. Plugging this back into the gradient descent algorithm;

$$\widehat{w}^{(t+1)} \leftarrow \widehat{w}^{(t)} - \eta \nabla RSS(\widehat{w}^{(t)})$$

this becomes our final **multiple regression gradient descent algorithm**

```
Multiple Regression Gradient Descent Algorithm
```

While $\left\|-2H^T\left(y - H\widehat{w}^{(t)}\right)\right\| < \epsilon$ do

$$\widehat{w}^{(t+1)} \leftarrow \widehat{w}^{(t)} + 2\eta H^T\left(y - H\widehat{w}^{(t)}\right)$$

We can look at the process to update a single feature *j* for some intuition;

$$\widehat{w}_j{}^{(t+1)} \leftarrow \widehat{w}_j{}^{(t)} + 2\eta \sum_{i=0}^{N} h_j(x_i)\left(y - \hat{y}\left(w^{(t)}\right)\right)$$

$y - \hat{y}\left(w^{(t)}\right)$ is the **residual** between the observed and predicted outputs at iteration t.
$h_j(x_i)$ is the value of **feature *j***

So we can see that the value of the feature is weighted by the residual. So if the residual is positive, the summation is positive and we increase the estimate of $\widehat{w}_j$. In other words, we increase the slope for feature j (the change in the output per unit change in feature j).

Another thing to note is that the gradient descent has the **partial derivative with respect to the j-th feature** $h_j(x_i)$ embedded in it. It is given by;

$2\sum_{i=0}^{N} h_j(x_i)\left(y - \hat{y}\left(w^{(t)}\right)\right) = 2\times(feature_i \times residual)$ = partial(j)

We can call that partial(j) when we use it in the gradient descent algorithm. Recall that twice the sum of the product of two vectors is just twice the dot product of the two vectors. Therefore the derivative for the weight for feature_i is just two times the dot product between the values of feature_i and the current residuals.

The **complete gradient descent algorithm** is given by:

- Choose initial values for the coefficients w. We can set the them to zero or we could be smarter if we wanted to converge faster.
- Choose a tolerance, ε
- t = 1, converted = false
- while not converged
    - sumOfSquares = 0
    - For j = 0 to D
        - $partial(j) \leftarrow 2\sum_{i=0}^{N} h_j(x_i)\left(y - \hat{y}\left(w^{(t)}\right)\right)$
        - $w^{(t+1)} \leftarrow w^{(t)} - \eta \times partial(j)$
        - $sumOfSquares \leftarrow sumOfSquares + partial(j) \times partial(j)$
    - $\left\|\nabla RSS\left(\widehat{w}^{(t)}\right)\right\| \leftarrow \sqrt{sumOfSquares}$
    - converged = $\left\|\nabla RSS\left(\widehat{w}^{(t)}\right)\right\| < \varepsilon$
    - t = t + 1

The magnitude of the residual sum of squares at iteration t is given by $\left\|\nabla RSS\left(\widehat{w}^{(t)}\right)\right\|$ and can be incrementally calculated in the while loop if we use the partial(j), or we can save partial(j) and calculated it after in its own loop.

- sumOfSquares = 0
- For j = 0 to D
    - sumOfSquares += partial(j)*partial(j)
- magnitudeSumOfSquares = sqrt(sumOfSquares)