Regression

Summary

Linear Regression

1. What is linear regression?
2. Computing θ using ordinary least squares
3. Optimizing θ using gradient descent and the least means squares method
4. The normal equation

Logistic Regression

1. Classification
2. Determining the decision boundaries
3. Computing θ using Log Loss
4. Optimizing θ using gradient descent and the least means squares method

Regularization

Linear Regression

What is linear regression?

The goal of linear regression (and any machine learning outcome) is to model the relationship between the data you have collected (independent variables / predictors) and the outcome you observed (the dependent variable / response).

Given our input data $X = (x_1, x_2, \ldots, x_n)$ and the response$y$, the general form of a linear regression problem is

$$y = \theta^T X$$

where X is the input data with the size of n observations and m features, θ are the learning coefficients that best describe the linear trend of the data. If there are three features, this model could be written out explicitly as:

$$y = {\theta_0}^T + {\theta_1}^T x_1 + {\theta_2}^T x_2 + {\theta_3}^T x_3$$

In machine learning, we train the linear model on a larger subset of data (called the **training set**), and fit the model to compute θ. Then we use θ on the remaining data that the model has not been fitted to (called the **test set**) to see how well the model can predict the response variable. To split the dataset into a training and test set, we can use the trainTestSplit function shown below. The default training set size in this function is 80% of the entire dataset.

```
function [Xtrain, Ytrain, Xtest, Ytest] = trainTestSplit(X, Y, trainSize)
    trainSize = 0.8;
    allVals = 1:size(Y, 1);
    vectorLength = round(trainSize * size(Y, 1));
    trainIndex = datasample(1:size(Y, 1), vectorLength, 'Replace', false)';
    idx = ismember(allVals, trainIndex);
    testIndex = allVals(~idx);

    Xtest = X(testIndex, :);
    Xtrain = X(trainIndex, :);
    Ytest = Y(testIndex, 1);
    Ytrain = Y(trainIndex, 1);
end
```

The model's predictions on the test set are generally called **hypotheses**, and our linear regression expression when referring to a machine learning problem will be written as

$$h(x_i) = \theta^T X$$

Computing the cost of θ using ordinary least squares and the Normal equation

Given a training set, how does an algorithm learn the parameters θ?

A common method is to see how close our hypothesis $h(x)$ is to the real y in the test set. **In other words, we'll compute the difference between our prediction and the real observation, and aim to minimize the error to find the best model fit.**

The difference for any prediction with respect to its response is known as the **cost**, and to compute the costs, we can input the predictions and test data into a **cost function**.

The formalized version of the linear regression cost function is formulated below:

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

This method of measuring the error between the model's predictions and the real value is known as **oridinary least squares**, and is a common measure of a linear model's error.

The MATLAB code for the linear regression cost function is written below:

```
function cost = costFunction(X, y, theta)
    m = length(y);
    cost = 0;

    predictions =  X * theta;
    error = (predictions - y) .^ 2;
    cost =  sum(error)/(2*m);
end
```

Let's get some intuition for how the cost changes in relation to the model's predictions with a toy dataset that is represented as

$$y = \theta^T x$$

```
x = [1, 2, 3];
y = [1, 2, 3];

% Make a scatter plot of the raw data
```

```
figure
scatter(x, y)
xlabel('X')
ylabel('Y')
lsline
```

The scatter plot shows a perfect linear line, and thus we can see that $\theta = 1$. If we compute the cost of this perfect linear relationship, we find that the cost is 0:

$$J(\theta) = \frac{1}{2 \cdot 3}(1-1)^2 + (2-2)^2 + (3-3)^2 = 0$$

However, what if we inject some noise into this perfect linear trend?

```
x = [1, 1.2, 2, 2.5, 3];
y = [1, 1.4, 2, 2.3, 3];
theta = y/x
cost = costFunction(x, y, theta)

% Make a scatter plot of the raw data
figure
scatter(x, y)
xlabel('X')
ylabel('Y')
lsline
```

We find that the updated $\theta = 0.9880$ and $J(\theta) = 0.0077$.

{INSERT FIGURE FOR RELATIONSHIP BETWEEN COST AND THETA}

If we fully explore the parameter space for θ and $J(\theta)$, we'll find that there is a square relationship between θ and $J(\theta)$. **Thus, the more we deviate from the expected value of θ, the higher we will penalize our linear regression model for choosing that value of θ.**

Optimizing θ using gradient descent

So we now know how the linear model ultimately decides on the value of the learning coefficient θ - by minimizing the cost of θ. However, out of all the possible values for θ, how do we choose the best one?

A common approach is **to start with an initial random value for θ, then iteratively make changes to θ to reduce the cost until θ converges to a value that minimizes $J(\theta)$.**

This algorithm is called **gradient descent** or more formally, **stochastic gradient descent,** and is formulated below:

$$\theta = \theta_j - \alpha \frac{\vartheta}{\vartheta \theta_j} J(\theta)$$

The parameter α is the learning rate, which will impact both the speed and the final solution of our estimation of θ. We will discuss how to choose α later in this lecture.

To work out the specific expression for the linear regression gradient descent formula, we need to solve the partial derivative of the cost function. We won't go over the actual derivation of the partial derivative of the cost function, but the final cost function for the linear model turns out to be relatively simple:

$$\frac{\vartheta}{\vartheta\theta_j}J(\theta) = (h_\theta(x) - y)x_j$$

And the final rule for determining the best coefficient becomes:

$$\theta = \theta_j - \alpha(h_\theta(x) - y)x_j$$

This specific rule is also known as the **Least Means Square** update rule, and its properties are intuitive. The magnitude of the update is porportional to the error within the model. As the error decreases with newly computed $h_\theta(x)$, the coefficients θ converge to a final stable value.

To implement the gradient descent function in MATLAB:

```
function [theta, cost] = gradientDescent(X, y, theta, alpha, num_iters)
    m = length(y);
    cost = zeros(num_iters, 1);

    for iter = 1:num_iters
        predictions =  X * theta;
        updates = X' * (predictions - y);
        theta = theta - alpha * (1/m) * updates;
        cost(iter) = costFunction(X, y, theta);
    end
end
```

Again, let's get some intuition for gradient descent. Let's start with a coefficient θ that does not describe the data all that well, and has a high cost associated with it. We will iterate the algorithm several times until we get a stable value for θ:

```
[theta, cost] = gradientDescent(X, y, 3, 0.05, 100)
plot(theta, cost)
```

{Insert figure}

Now gradient descent will take the initial approximation for theta, and update the value for theta with respect

The normal equation

The previous method estimates the best parameter θ by minimizing the error of our prediction with respect to the actual value in an iterative fashion using gradient descent. However, there is an analytical way to compute θ without gradient descent. This approach is called the Normal equation:

$$\theta = (X^TX)^{-1}X^Ty$$

While this equation is powerful on its own, the weakness is computing $(X^TX)^{-1}$, which becomes incredibly slow as the number of features increases.

Logistic Regression

Classification

Determining the decision boundaries

Computing θ using Log Loss

Optimizing θ using gradient descent and the least means squares method