Universidade Estadual de Campinas
École Centrale de Nantes
Coursera / Pier.

Machine Learning

Vinícius Cordeiro Pereira

March 8, 2020

Contents

1. Introduction

What is Machine Learning?

Two definitions of Machine Learning are offered.

1. Arthur Samuel described it as: "**the field of study that gives computers the ability to learn without being explicitly programmed.**" This is an older, informal definition.
2. Tom Mitchell provides a more modern definition: "**A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.**"

Example: playing checkers.

- E = the experience of playing many games of checkers
- T = the task of playing checkers.
- P = the probability that the program will win the next game.

In general, any machine learning problem can be assigned to one of two broad classifications: *Supervised learning and Unsupervised learning.*

Supervised Learning

In supervised learning, we are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and the output.

Supervised learning problems are categorized into "**regression**" and "**classification**" problems. In a regression problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function. In a classification problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.

Example :

Given data about the size of houses on the real estate market, try to predict their price. Price as a function of size is a continuous output, so this is a regression problem.

We could turn this example into a classification problem by instead making our output about whether the house "sells for more or less than the asking price." Here we are classifying the houses based on price into two discrete categories.

Unsupervised Learning

Unsupervised learning allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables.

We can derive this structure by clustering the data based on relationships among the variables in the data.

With unsupervised learning there is no feedback based on the prediction results.

Example:

Clustering: Take a collection of 1,000,000 different genes, and find a way to automatically group these genes into groups that are somehow similar or related by different variables, such as lifespan, location, roles, and so on.

Non-clustering: The "Cocktail Party Algorithm", allows you to find structure in a chaotic environment. (i.e. identifying individual voices and music from a mesh of sounds at a cocktail party).

2. Linear Regression with One Variable

To establish notation for future use, we'll use $x^{(i)}$ to denote the "input" variables (living area in this example), also called input features, and $y^{(i)}$ to denote the "output" or target variable that we are trying to predict (price). A pair ($x^{(i)}$, $y^{(i)}$) is called a training example, and the dataset that we'll be using to learn-a list of m training examples ($x^{(i)}$, $y^{(i)}$); $i = 1, ..., m$ - is called a **training set**.

Note that the superscript **"(i)"** in the notation is simply an index into the training set, and has nothing to do with exponentiation. We will also use X to denote the space of input values, and Y to denote the space of output values. In this example, X = Y = $\mathbb{R}$.

To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function h : X $\rightarrow$ Y so that h(x) is a "good" predictor for the corresponding value of y. For historical reasons, this function h is called a hypothesis. Seen pictorially, the process is therefore like this:

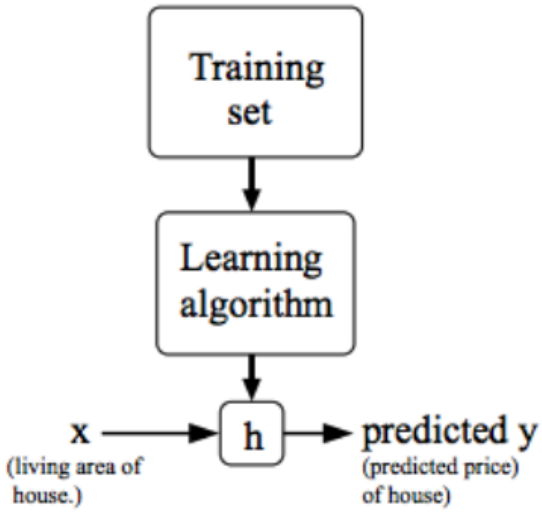

Figure 2.1: Supervised learning

2.1 Cost Function

We can measure the accuracy of our hypothesis function by using a cost function. This takes an average difference (actually a fancier version of an average) of all the results of the hypothesis with inputs from x's and the actual output y's.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2 \tag{2.1}$$

Where:

$$h_\theta(x_i) = \theta_0 + \theta_1 \cdot x_i$$

This function is otherwise called the "Squared error function", or "Mean squared error". The mean is halved $\left(\frac{1}{2}\right)$ as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the $\frac{1}{2}$ term.

2.2 Gradient Descent

So we have our hypothesis function and we have a way of measuring how well it fits into the data. Now we need to estimate the parameters in the hypothesis function. That's where gradient descent comes in.

Imagine that we graph our hypothesis function based on its fields θ_0 and θ_1 (actually we are graphing the cost function as a function of the parameter estimates). We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting a particular set of parameters.

We put θ_0 on the x axis and θ_1 on the y axis, with the cost function on the vertical z axis. The points on our graph will be the result of the cost function using our hypothesis with those specific theta parameters. The graph below depicts such a setup.

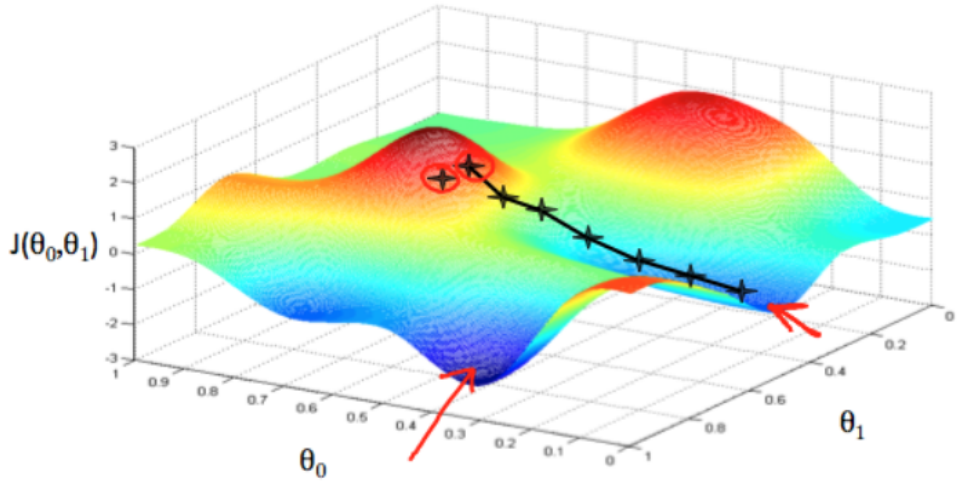

Figure 2.2: Supervised learning

We will know that we have succeeded when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum. The red arrows show the minimum points in the graph.

The way we do this is by taking the derivative (the tangential line to a function) of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent. The size of each step is determined by the parameter α, which is called the learning rate.

For example, the distance between each 'star' in the graph above represents a step determined by our parameter α. A smaller α would result in a smaller step and a larger α results in a larger step. The direction in which the step is taken is determined by the partial derivative of J(θ_0,θ_1). Depending on where one starts on the graph, one could end up at different points. The image above shows us two different starting points that end up in two different places.

The gradient descent algorithm is (repeat until convergence):

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x_j^{(i)} \tag{2.2}$$

Where, j=0,1 represents the feature index number.

Remark : $\mathbf{x_0^{(i)}}$ is ALWAYS equal 0.

On a side note, we should adjust our parameter α to ensure that the gradient descent algorithm converges in a reasonable time. Failure to converge or too much time to obtain the minimum value imply that our step size is wrong.

Correct: Simultaneous update

$temp0 := \theta_0$ - $\alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$
$temp1 := \theta_1$ - $\alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$
$\theta_0 := temp0$
$\theta_1 := temp1$

3. Linear Regression with Multiple Variables

Linear regression with multiple variables is also known as "multivariate linear regression".

We now introduce notation for equations where we can have any number of input variables.

- $x_j^{(i)}$ = value of feature j in the i^{th} training example
- $x^{(i)}$ = the input (features) of the ith training example
- m = the number of training examples
- n = the number of features

The multivariable form of the hypothesis function accommodating these multiple features is as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$$

In order to develop intuition about this function, we can think about θ_0 as the basic price of a house, θ_1 as the price per square meter, θ_2 as the price per floor, etc. x_1 will be the number of square meters in the house, x_2 the number of floors, etc.
Using the definition of matrix multiplication, our multivariable hypothesis function can be concisely represented as:

$$h_\theta(x) = \begin{bmatrix}\theta_1 & \theta_2 & \cdots & \theta_n\end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x \tag{3.1}$$

3.1 Gradient Descent for Multiple Variables

The gradient descent equation itself is generally the same form; we just have to repeat it for our 'n' features:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x_j^{(i)} \qquad for\ j := 0 \ldots n \tag{3.2}$$

3.2 Gradient Descent in Practice - Feature Scaling

We can speed up gradient descent by having each of our input values in roughly the same range. This is because θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven.

The way to prevent this is to modify the ranges of our input variables so that they are all roughly the same. Ideally:

$$\text{-1} \leq x_{(i)} \leq 1 \text{ or } \text{-0.5} \leq x_{(i)} \leq 0.5$$

These aren't exact requirements; we are only trying to speed things up. The goal is to get all input variables into roughly one of these ranges, give or take a few.

Two techniques to help with this are feature scaling and mean normalization. Feature scaling involves dividing the input values by the range (i.e. the maximum value minus the minimum value) of the input variable, resulting in a new range of just 1. Mean normalization involves subtracting the average value for an input variable from the values for that input variable resulting in a new average value for the input variable of just zero. To implement both of these techniques, adjust your input values as shown in this formula:

$$x_i := \frac{x_i - \mu_i}{\sigma_i}$$

Where μ_i is the average of all the values for feature (i) and σ_i is the standard deviation.

3.3 Features and Polynomial Regression

We can improve our features and the form of our hypothesis function in a couple different ways.

We can combine multiple features into one. For example, we can combine x_1 and x_2 into a new feature x_3 by taking $x_1 \cdot x_2$.

3.3.1 Polynomial Regression

Our hypothesis function need not be linear (a straight line) if that does not fit the data well.

We can change the behavior or curve of our hypothesis function by making it a quadratic, cubic or square root function (or any other form).

For example, if our hypothesis function is $h_\theta(x) = \theta_0 + \theta_1 x_1$ then we can create additional features based on x_1 to get the quadratic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$ or the cubic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$.

One important thing to keep in mind is, if you choose your features this way then feature scaling becomes very important.

3.4 Normal Equation

Gradient descent gives one way of minimizing **J**. Let's discuss a second way of doing so, this time performing the minimization explicitly and without resorting to an iterative algorithm. In the "*Normal Equation*" method, we will minimize J by explicitly taking its derivatives with respect to the $\theta_j's$, and setting them to zero. This allows us to find the optimum theta without iteration. The normal equation formula is given below:

$$\theta = \left(X^T X\right)^{-1} X^T y \tag{3.3}$$

There is no need to do feature scaling with the normal equation.

Gradient Descent	**Normal Equation**
Need to choose alpha	No need to choose alpha
Needs many iterations	No need to iterate
$\mathcal{O}\ (kn^2)$	$\mathcal{O}\ (n^3)$, need to calculate inverse of X^T
Works well when n is large	Slow if n is very large

With the normal equation, computing the inversion has complexity $\mathcal{O}(n\hat{}3)$. So if we have a very large number of features, the normal equation will be slow. In practice, when n exceeds 10,000 it might be a good time to go from a normal solution to an iterative process.

3.4.1 Normal Equation Noninvertibility

When implementing the normal equation in octave we want to use the **'pinv'** function rather than **'inv.'** The **'pinv'** function will give you a value of θ even if $X^T X$ is not invertible.
If $X^T X$ is noninvertible, the common causes might be having :

- Redundant features, where two features are very closely related (i.e. they are linearly dependent)
- Too many features (e.g. m $\leq$ n). In this case, delete some features or use "regularization" (to be explained in a later lesson).

Solutions to the above problems include deleting a feature that is linearly dependent with another or deleting one or more features when there are too many features.

4. Logistic Regression

4.1 Classification

The classification problem is just like the regression problem, except that the values we now want to predict take on only a small number of discrete values. For now, we will focus on the binary classification problem in which **y** can take on only two values, 0 and 1. (Most of what we say here will also generalize to the multiple-class case.) For instance, if we are trying to build a spam classifier for email, then $x^{(i)}$ may be some features of a piece of email, and y may be 1 if it is a piece of spam mail, and 0 otherwise. Hence, y $\in$ 0,1. 0 is also called the negative class, and 1 the positive class, and they are sometimes also denoted by the symbols "-" and "+." Given $x^{(i)}$, the corresponding $y^{(i)}$ is also called the label for the training example.

4.1.1 Hypothesis Representation

We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old linear regression algorithm to try to predict y given x. However, it is easy to construct examples where this method performs very poorly. Intuitively, it also doesn't make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we know that y $\in$ 0, 1. To fix this, let's change the form for our hypotheses $h_\theta(x)$ to satisfy $0 \leq h_\theta(x) \leq 1$. This is accomplished by plugging $\theta^T x$ into the Logistic Function (also called Sigmoid Function).

$$
\begin{aligned}
h_\theta(x) &= g(\theta^T x) \\
z &= \theta^T x \\
g(z) &= \frac{1}{1+e^{-z}}
\end{aligned}
$$

The following image shows us what the sigmoid function looks like:

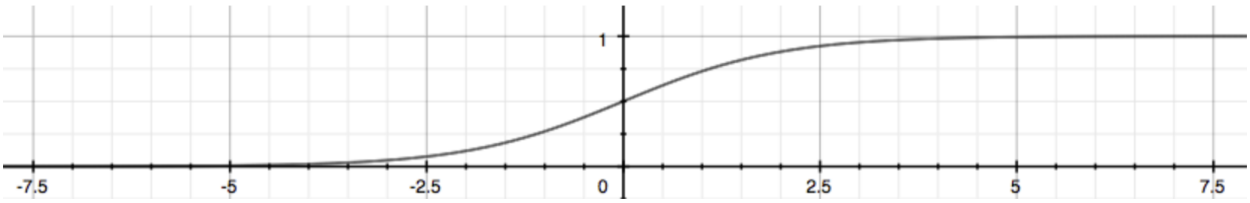

Figure 4.1: Sigmoid function

The function $g(z)$, shown here, maps any real number to the (0, 1) interval, making it useful for transforming an arbitrary-valued function into a function better suited for classification.

$h_\theta(x)$ will give us the probability that our output is 1. For example, $h_\theta(x) = 0.7$ gives us a probability of 70% that our output is 1. Our probability that our prediction is 0 is just the complement of our probability that it is 1 (e.g. if probability that it is 1 is 70%, then the probability that it is 0 is 30%).

4.1.2 Decision Boundary

In order to get our discrete 0 or 1 classification, we can translate the output of the hypothesis function as follows:

$$\begin{aligned} When\ \ z \geq 0 \ \ &\Rightarrow \ \ h_\theta(x) \geq 0.5 \rightarrow y = 1 \\ When\ \ z \leq 0 \ \ &\Rightarrow \ \ h_\theta(x) \leq 0.5 \rightarrow y = 0 \end{aligned}$$

So if our input to g is $\theta^T X$, then that means:

$$\begin{aligned} &h_\theta(x) = g(\theta^T x) \geq 0.5 \\ &When\ \ \theta^T x \geq 0 \end{aligned}$$

From these statements we can now say:

$$\begin{aligned} &\theta^T x \geq 0 \Rightarrow y = 1 \\ &\theta^T x < 0 \Rightarrow y = 0 \end{aligned}$$

The decision boundary is the line that separates the area where $\mathbf{y = 0}$ and where $\mathbf{y = 1}$. It is created by our hypothesis function.

4.1.3 Cost Function

We cannot use the same cost function that we use for linear regression because the Logistic Function will cause the output to be **wavy**, causing many local optima. In other words, *it will not be a convex function.*

Instead, our cost function for logistic regression looks like:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost\left(h_\theta(x^{(i)}), y^{(i)}\right) \tag{4.1}$$

Where:

$$Cost\left(h_\theta(x^{(i)}), y^{(i)}\right) = -log(h_\theta(x)) \qquad if \quad y = 1 \tag{4.2}$$

$$Cost\left(h_\theta(x^{(i)}), y^{(i)}\right) = -log(1 - h_\theta(x)) \qquad if \quad y = 0 \tag{4.3}$$

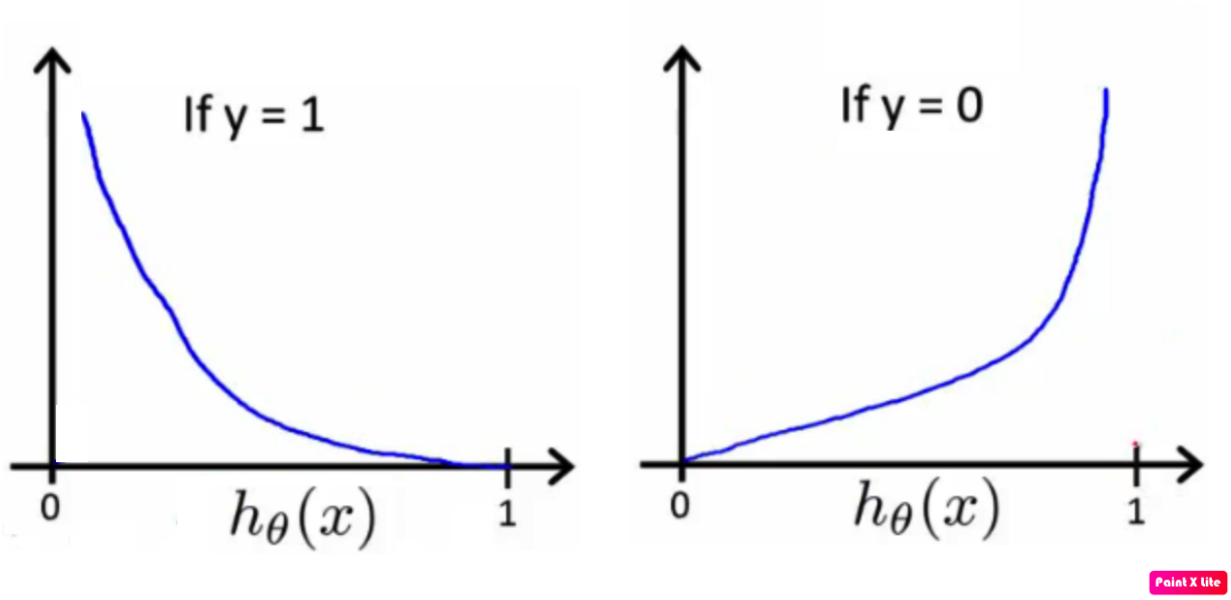

Figure 4.2: When **y = 1 (left)**, **y = 0 (right)** we get the following plot for J(θ) vs $h_\theta(x)$

If our correct answer 'y' is 0, then the cost function will be 0 if our hypothesis function also outputs 0. If our hypothesis approaches 1, then the cost function will approach infinity.

If our correct answer 'y' is 1, then the cost function will be 0 if our hypothesis function outputs 1. If our hypothesis approaches 0, then the cost function will approach infinity.

We can compress our cost function's two conditional cases into one case:

$$Cost(h_\theta(x^{(i)}), y^{(i)}) = -y \cdot log(h_\theta(x)) - (1-y) \cdot log(1-h_\theta(x)) \tag{4.4}$$

We can fully write out our entire cost function as follows:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \cdot log(h_\theta(x^{(i)})) + (1-y^{(i)}) \cdot log(1-h_\theta(x^{(i)}))\right]$$

A vectorized implementation is:

$$\begin{aligned} h &= g(X\theta) \\ J(\theta) &= \frac{1}{m} \cdot (-y^T \cdot log(h) - (1-y)^T \cdot log(1-h)) \end{aligned}$$

4.1.4 Gradient Descent

The general form of gradient descent is:

$$\begin{aligned} &Repeat: \{ \\ &\quad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \\ &\} \end{aligned}$$

We can work out the derivative part using calculus to get:

$$\begin{aligned} &Repeat: \{ \\ &\quad \theta_j := \theta_j - \alpha \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot x_j^{(i)} \\ &\} \end{aligned}$$

Notice that this algorithm is identical to the one we used in linear regression. We still have to simultaneously update all values in theta.

A vectorized implementation is:

$$\theta := \theta - \tfrac{\alpha}{m} X^T \left(g(X\theta) - \overrightarrow{y}\right)$$

4.2 Multiclass Classification: One-vs-all

Now we will approach the classification of data when we have more than two categories. Instead of y = 0,1 we will expand our definition so that y = 0,1...n.

Since y = 0,1...n, we divide our problem into n+1 (+1 because the index starts at 0) binary classification problems; in each one, we predict the probability that 'y' is a member of one of our classes.

$$
\begin{aligned}
&y \in \{0, 1 \dots n\} \\
&h_\theta^{(0)}(x) = P(y = 0|x; \theta) \\
&h_\theta^{(1)}(x) = P(y = 1|x; \theta) \\
&\dots \\
&h_\theta^{(n)}(x) = P(y = n|x; \theta) \\
&prediction = \max_i(h_\theta^{(i)}(x))
\end{aligned}
$$

We are basically choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that returned the highest value as our prediction.

The following image shows how one could classify 3 classes:

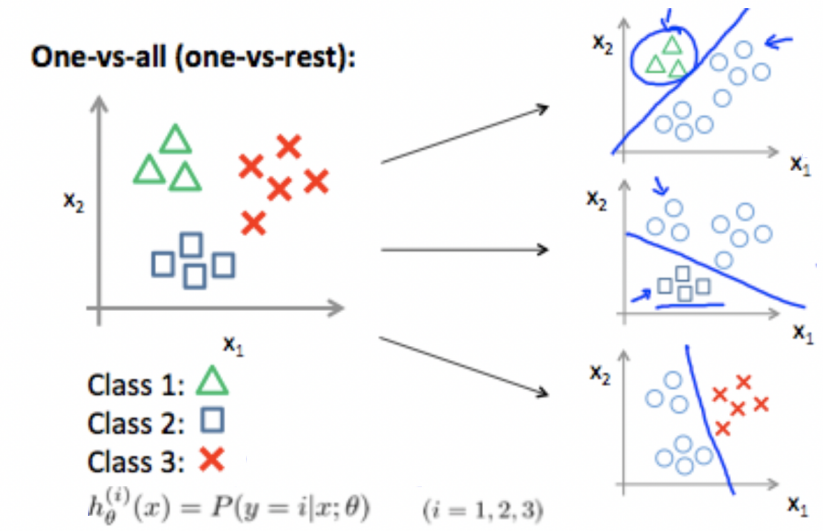

Figure 4.3: One-vs-all or One-vs-Rest

5. Regularization

5.1 The Problem of Overfitting

Consider the problem of predicting y from x $\in \mathbb{R}$. The leftmost figure below shows the result of fitting a $y = \theta_0 + \theta_1 x$ to a dataset. We see that the data doesn't really lie on straight line, and so the fit is not very good.

Instead, if we had added an extra feature x^2, and fit $y = \theta_0 + \theta_1 x + \theta_2 x^2$, then we obtain a slightly better fit to the data (See middle figure). Naively, it might seem that the more features we add, the better. However, there is also a danger in adding too many features: The rightmost figure is the result of fitting a 5^{th}order polynomial

$$y = \sum_{j=0}^{5} \theta_j x^j.$$

We see that even though the fitted curve passes through the data perfectly, we would not expect this to be a very good predictor of, say, housing prices (y) for different living areas (x). Without formally defining what these terms mean, we'll say the *figure on the left* shows an instance of **underfitting**—in which the data clearly shows structure not captured by the model—and the *figure on the right* is an example of **overfitting**.

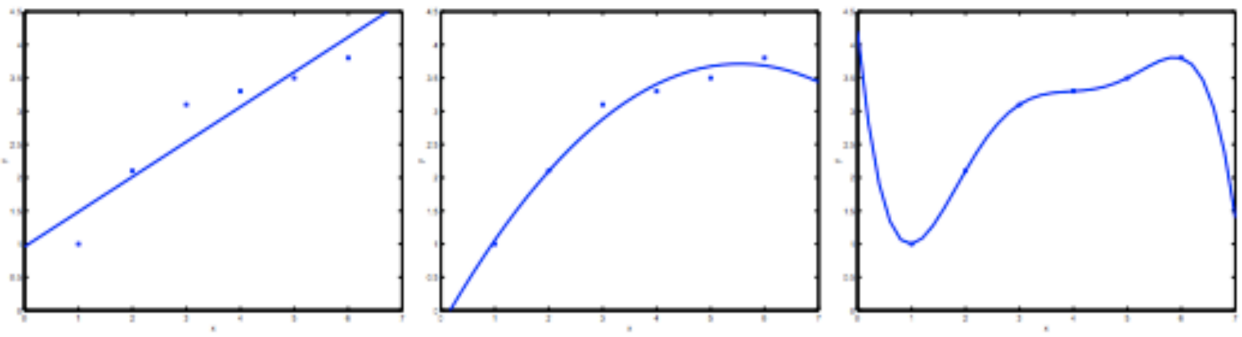

Figure 5.1: One-vs-all or One-vs-Rest

Underfitting, or high bias, is when the form of our hypothesis function h maps poorly to the trend of the data.

Overfitting, or high variance, is caused by a hypothesis function that fits the available data but does not generalize well to predict new data.

This terminology is applied to both linear and logistic regression. There are two main options to address the issue of overfitting:

1. Reduce the number of features: Manually select which features to keep; Use a model selection algorithm.
2. Regularization: Keep all the features, but reduce the magnitude of parameters θ_j. Regularization works well when we have a lot of slightly useful features.

5.2 Cost Function

Say we wanted to make the following function more quadratic:

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

We'll want to eliminate the influence of $\theta_3 x^3$and $\theta_4 x^4$ without actually getting rid of these features or changing the form of our hypothesis, we can instead modify our cost function:

$$min_\theta \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 + 1000 \cdot \theta_3 x^3 + 1000 \cdot \theta_4 x^4$$

We've added two extra terms at the end to inflate the cost of $\theta_3$and θ_4. Now, in order for the cost function to get close to zero, we will have to reduce the values of $\theta_3$and θ_4 to near zero. This will in turn greatly reduce the values of $\theta_3 x^3$and $\theta_4 x^4$ in our hypothesis function.

5.2.1 Linear Regression

We could also regularize all of our theta parameters in a single summation as:

$$min_\theta \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 + \lambda \sum_{i=1}^{n} \theta_j^2$$

The lambda is the regularization parameter. It determines how much the costs of our theta parameters are inflated.

5.2.2 Logistic Regression

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \cdot log(h_\theta(x^{(i)})) + (1-y^{(i)}) \cdot log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{i=1}^{n}\theta_j^2$$

The second sum,

$$\sum_{i=1}^{n}\theta_j^2$$

means to explicitly exclude the bias term, θ_0. I.e. the θ vector is indexed from 0 to n (holding n+1 values, θ_0 through θ_n), and this sum explicitly skips θ_0, by running from 1 to n, skipping 0. (**Identically for linear regression**).

5.3 Gradient Descent

We will modify our gradient descent function to separate out θ_0 from the rest of the parameters because we do not want to penalize θ_0.

$$\begin{aligned}
&Repeat: \{ \\
&\quad \theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot x_0^{(i)} \\
&\quad \theta_j := \theta_j - \alpha\left[\left(\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j\right] \qquad j \in \{1,2,\ldots,n\} \\
&\quad \}
\end{aligned}$$

5.4 Normal Equation

Now let's approach regularization using the alternate method of the non-iterative normal equation.

To add in regularization, the equation is the same as our original, except that we add another term inside the parentheses:

$$\theta = (X^TX + \lambda \cdot L)^{-1}X^Ty$$

Where:

$$L = \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}$$

L is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension (n+1)x(n+1).

6. Neural Networks

Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. They interpret sensory data through a kind of machine perception, labeling or clustering raw input. The patterns they recognize are numerical, contained in vectors, into which all real-world data, be it images, sound, text or time series, must be translated.

Neural networks help us cluster and classify. You can think of them as a clustering and classification layer on top of the data you store and manage. They help to group unlabeled data according to similarities among the example inputs, and they classify data when they have a labeled dataset to train on.

6.1 Model Representation

Let's examine how we will represent a hypothesis function using neural networks. At a very simple level, neurons are basically computational units that take inputs (dendrites) as electrical inputs (called "spikes") that are channeled to outputs (axons). In our model, our dendrites are like the input features $x_1 \dots x_n$, and the output is the result of our hypothesis function. In this model our x_0 input node is sometimes called the "bias unit." It is always equal to 1. In neural networks, we use the same logistic function as in classification, $\frac{1}{1+e^{-\theta^T x}}$, yet we sometimes call it a sigmoid (logistic) activation function. In this situation, our "theta" parameters are sometimes called "weights".

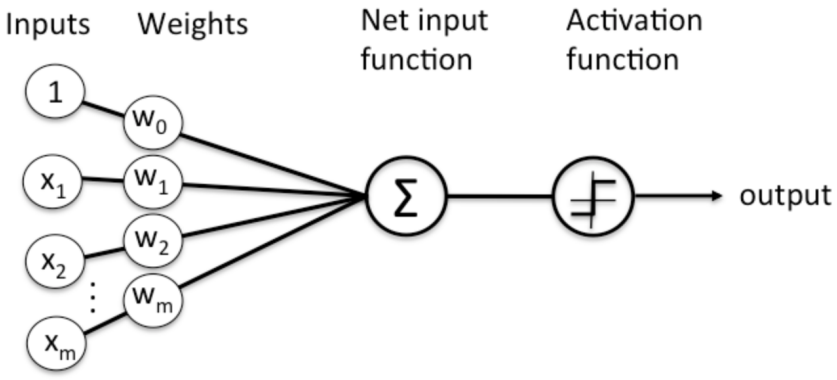

Figure 6.1: Neural diagram

Visually, a simplistic representation looks like:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \rightarrow [\quad] \rightarrow h_\theta(x)$$

Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer".We can have intermediate layers of nodes between the input and output layers called the "hidden layers."

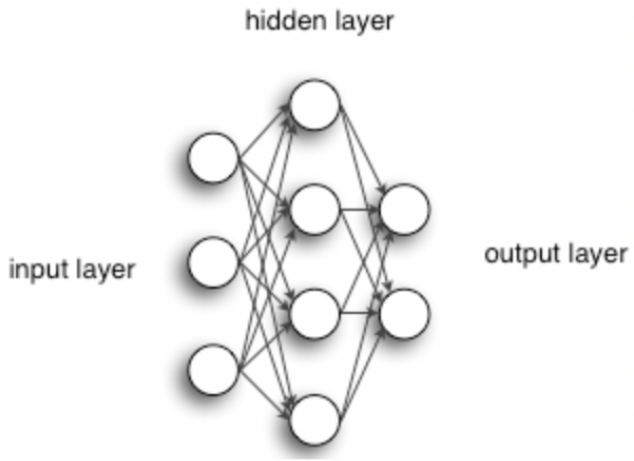

Figure 6.2: Neural layers

Where:

- $a_i^{(j)}$ = "**activation**" of unit i in layer j
- $\Theta^{(j)}$=matrix of **weights** controlling function mapping from layer j to layer $j+1$

If we had one hidden layer, with **3** activations, it would look like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} \rightarrow h_\theta(x)$$

The values for each of the "activation" nodes is obtained as follows:

$$\begin{aligned}
a_1^{(2)} &= g\left(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\right) \\
a_2^{(2)} &= g\left(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3\right) \\
a_3^{(2)} &= g\left(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3\right) \\
h\Theta^(x_) = a_1^{(3)} &= g\left(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}\right)
\end{aligned}$$

This is saying that we compute our activation nodes by using a *3x4* matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by yet another parameter matrix $\Theta\hat{}(2)$ containing the weights for our second layer of nodes. And each layer gets its own matrix of weights, $\Theta^{(j)}$.

The dimensions of these matrices of weights is determined as follows:

> If network has s_j units in layer j and s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$.

The $+\mathbf{1}$ comes from the addition in $\Theta^{(j)}$ of the "**bias nodes**", x_0 and $\Theta_0^{(j)}$. In other words the output nodes will not include the bias nodes while the inputs will.

6.1.1 Vectorized implementation

We're going to define a new variable $z_k^{(j)}$ that encompasses the parameters inside our **g** function. In our previous example if we replaced by the variable **z** for all the parameters we would get:

$$\begin{aligned}
a_1^{(2)} &= g\left(z_1^{(2)}\right) \\
a_2^{(2)} &= g\left(z_2^{(2)}\right) \\
a_3^{(2)} &= g\left(z_3^{(2)}\right)
\end{aligned}$$

In other words, for layer $j = 2$ and node **k**, the variable **z** will be:

$$z_k^{(2)} = g\left(\Theta_{k,0}^{(1)}x_0 + \Theta_{k,1}^{(1)}x_1 + \cdots + \Theta_{k,n}^{(1)}x_n\right)$$

Setting the input $x = a^{(1)}$, we can rewrite the equation as:

$$z^{(j)} = \Theta^{(j-1)} a^{(j-1)} \tag{6.1}$$

Now we can get a vector of our activation nodes for layer j as follows:

$$a^{(j)} = g(z^{(j)}) \tag{6.2}$$

6.2 Cost Function

Let's first define a few variables that we will need to use:

- L = total number of layers in the network
- s_l = number of units (not counting bias unit) in layer l
- K = number of output units/classes

Our cost function for neural networks is going to be a generalization of the one we used for logistic regression, but it is going to be slightly more complicated:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} \left[y_k^{(i)} \cdot log((h_\theta(x^{(i)}))_k) + (1 - y_k^{(i)}) \cdot log(1 - (h_\theta(x^{(i)}))_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_j} \sum_{j=1}^{s_{j+1}} \left(\Theta_{j,i}^{(l)} \right)^2$$

We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

Note:

- the double sum simply adds up the logistic regression costs calculated for each cell in the output layer
- the triple sum simply adds up the squares of all the individual Θs in the entire network.
- the i in the triple sum does not refer to training example i

6.3 Backpropagation Algorithm

"Backpropagation" is neural-network terminology for **minimizing our cost function**, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute:

$$min_{\Theta} J(\Theta)$$

To do so, we use the following algorithm:

Given training set $(x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)})$

Set $\Delta_{i,j}^{(l)} := 0$ for all (l, i, j), (hence you end up having a matrix full of zeros).

For training example t =1 to m:

1. Set $a^{(1)} := x^{(t)}$

2. Perform forward propagation to compute $a^{(l)}$, for l=2,3,...,L

3. Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$

   Where L is our total number of layers and $a^{(L)}$ is the vector of outputs of the activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in $\mathbf{y}$. To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left:

4. Compute $\delta^{(L-1)}$, $\delta^{(L-2)}$,...,$\delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) .* a^{(l)} .* (1 - a^{(l)})$

   The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values given by $z^{(l)}$

   The g-prime derivative terms can also be written out as:

$$g'(z^{(l)}) = a^{(l)} .* (1 - a^{(l)})$$

5. $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta^{(l+1)}$ or with vectorization, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

Hence we update our new Δ matrix.

- $D_{i,j}^{(l)} := \frac{1}{m} \left(\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)} \right), \quad if \;\; j \neq 0.$
- $D_{i,j}^{(l)} := \frac{1}{m} \Delta_{i,j}^{(l)}, \quad if \;\; j = 0.$

The capital-delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. Thus we get $\frac{\partial}{\partial \Theta}{}_{ij}^{(l)} J(\Theta) = D_{ij}^{(l)}$

7. Advice for Applying Machine Learning

7.1 Evaluating a Hypothesis

Once we have done some trouble shooting for errors in our predictions by:

- Getting more training examples
- Trying smaller sets of features
- Trying additional features
- Trying polynomial features
- Increasing or decreasing λ

We can move on to evaluate our new hypothesis.

A hypothesis may have a low error for the training examples but still be inaccurate (because of overfitting). Thus, to evaluate a hypothesis, given a dataset of training examples, we can split up the data into two sets: a training set and a test set. Typically, the training set consists of 70 % of your data and the test set is the remaining 30 %.

The new procedure using these two sets is then:

1. Learn Θ and minimize $J_{train}(\Theta)$ using the training set.
2. Compute the test set error $J_{test}(\Theta)$

7.1.1 The test set error

1. For linear regression:
$$J_{test}(\Theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_\Theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$$
2. For classification $\rightarrow$ Misclassification error (aka 0/1 misclassification error):

$$err(h_\Theta(x), y) = \begin{cases} 1, & \text{if } h_\Theta(x) \geq 0.5 \text{ and } y = 0 \text{ or } h_\Theta(x) < 0.5 \text{ and } y = 1 \\ 0, & \text{otherwise} \end{cases}$$

This gives us a binary 0 or 1 error result based on a misclassification. The average test error for the test set is:

$$\text{Test Error} = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err(h_\Theta(x_{test}^{(i)}) - y_{test}^{(i)})$$

This gives us the proportion of the test data that was misclassified.

7.2 Model Selection and Train/Validation/Test Sets

Just because a learning algorithm fits a training set well, that does not mean it is a good hypothesis. It could over fit and as a result your predictions on the test set would be poor. The error of your hypothesis as measured on the data set with which you trained the parameters will be lower than the error on any other data set.

Given many models with different polynomial degrees, we can use a systematic approach to identify the 'best' function. In order to choose the model of your hypothesis, you can test each degree of polynomial and look at the error result.

One way to break down our dataset into the three sets is:

- Training set: 60%
- Cross validation set: 20%
- Test set: 20%

We can now calculate three separate error values for the three different sets using the following method:

1. Optimize the parameters in Θ using the training set for each polynomial degree.
2. Find the polynomial degree d with the least error using the cross validation set.
3. Estimate the generalization error using the test set with $J_{test}(\Theta^{(d)})$, (d = theta from polynomial with lower error);

This way, the degree of the polynomial d has not been trained using the test set.

7.3 Diagnosing Bias vs. Variance

In this section we examine the relationship between the degree of the polynomial d and the underfitting or overfitting of our hypothesis.

We need to distinguish whether bias or variance is the problem contributing to bad predictions, given that **high bias is underfitting** and **high variance is overfitting**. Ideally, we need to find a golden mean between these two.

The training error will tend to decrease as we increase the degree d of the polynomial. At the same time, the cross validation error will tend to decrease as we increase d up to a point, and then it will increase as d is increased, forming a convex curve.

High bias (underfitting): both $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ will be high. Also, $J_{CV}(\Theta) \approx J_{train}(\Theta)$.

High variance (overfitting): $J_{train}(\Theta)$ will be low and $J_{CV}(\Theta)$ will be much greater than $J_{train}(\Theta)$.

The is summarized in the figure below:

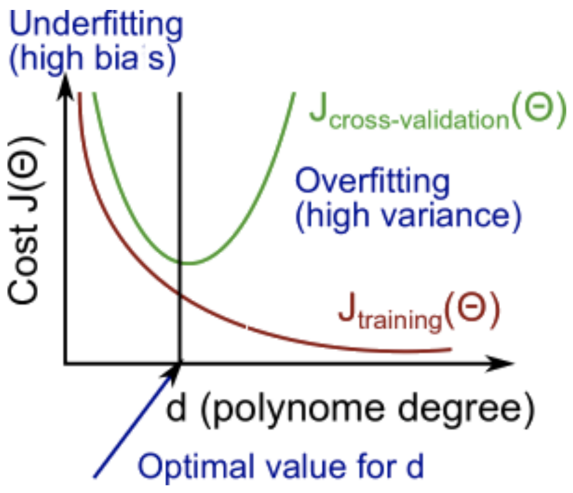

Figure 7.1: Optimal value for the degree d of the polynomial

7.4 Regularization and Bias/Variance

We can see that as λ increases, our fit becomes more rigid. On the other hand, as λ approaches 0, we tend to overfit the data. So how do we choose our parameter λ to get it 'just right' ? In order to choose the model and the regularization term λ, we need to:

1. Create a list of lambdas (i.e. $\lambda \int$ {0,0.01,0.02,0.04,0.08,0.16,0.32,0.64,1.28,2.56,5.12,10.24});
2. Create a set of models with different degrees or any other variants.
3. Iterate through the λ and for each λ go through all the models to learn some Θ.
4. Compute the cross validation error using the learned Θ (computed with λ) on the $J_{CV}(\Theta)$ without regularization or $\lambda = 0$.
5. Select the best combo that produces the lowest error on the cross validation set.
6. Using the best combo Θ and λ, apply it on $J_{test}(\Theta)$ to see if it has a good generalization of the problem.

7.5 Learning Curves

Training an algorithm on a very few number of data points (such as 1, 2 or 3) will easily have 0 errors because we can always find a quadratic curve that touches exactly those number of points. Hence:

- As the training set gets larger, the error for a quadratic function increases.
- The error value will plateau out after a certain m, or training set size.

Experiencing high bias:

Low training set size: causes $J_{train}(\Theta)$ to be low and $J_{CV}(\Theta)$ to be high.

Large training set size: causes both $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ to be high with $J_{train}(\Theta) \approx J_{CV}(\Theta)$.

If a learning algorithm is suffering from **high bias**, getting more training data will not (**by itself**) help much.

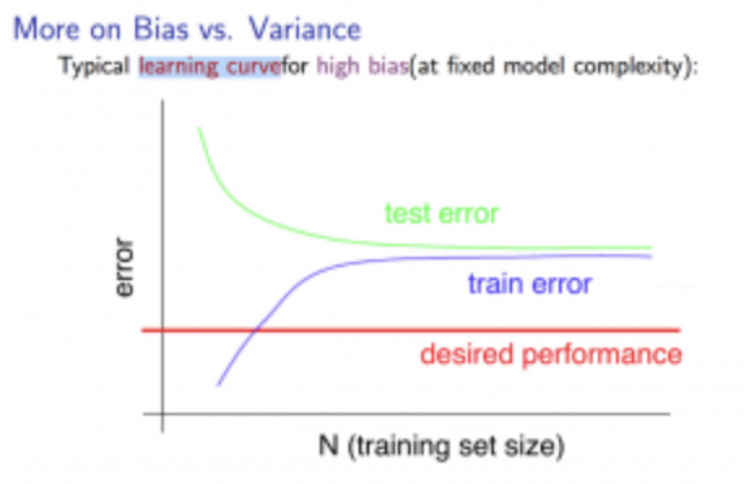

Figure 7.2: High Bias

Experiencing high variance:

Low training set size: $J_{train}(\Theta)$ will be low and $J_{CV}(\Theta)$ to be high.

Large training set size: $J_{train}(\Theta)$ increases with training set size and $J_{CV}(\Theta)$ continues to decrease without leveling off. Also, $J_{train}(\Theta) < J_{CV}(\Theta)$ but the difference between them remains significant. .

If a learning algorithm is suffering from **high variance**, getting more training data is **likely** to help.

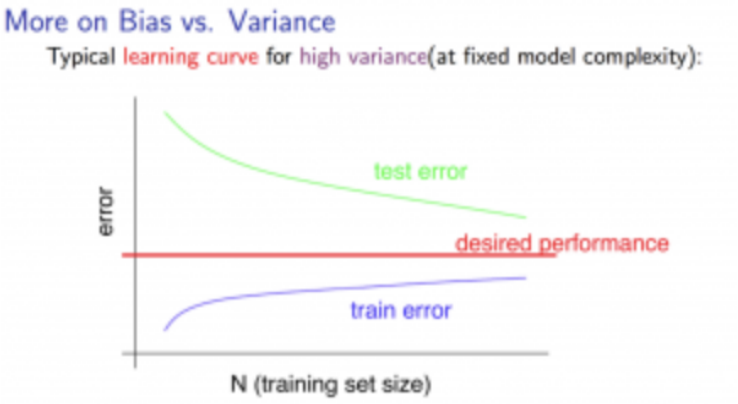

Figure 7.3: High Variance

7.6 Deciding What to Do Next

Table 7.1: Decision process

Fixes High BIAS	**Fixes High VARIANCE**
Adding features	Getting more training examples
Adding polynomial features	Trying smaller sets of features
Decreasing λ	Increasing λ

7.6.1 Diagnosing Neural Networks

- A neural network with fewer parameters is prone to underfitting. It is also computationally cheaper.
- A large neural network with more parameters is prone to overfitting. It is also computationally expensive. In this case you can use regularization (increase λ) to address the overfitting.

Using a single hidden layer is a good starting default. You can train your neural network on a number of hidden layers using your cross validation set. You can then select the one that performs best.

7.6.2 Model Complexity Effects

- Lower-order polynomials (low model complexity) have high bias and low variance. In this case, the model fits poorly consistently.
- Higher-order polynomials (high model complexity) fit the training data extremely well and the test data extremely poorly. These have low bias on the training data, but very high variance.
- In reality, we would want to choose a model somewhere in between, that can generalize well but also fits the data reasonably well.

7.7 Prioritizing What to Work On

7.7.1 System Design Example

Given a data set of emails, we could construct a vector for each email. Each entry in this vector represents a word. The vector normally contains 10,000 to 50,000 entries gathered by finding the most frequently used words in our data set. If a word is to be found in the email, we would assign its respective entry a 1, else if it is not found, that entry would be a 0. Once we have all our x vectors ready, we train our algorithm and finally, we could use it to classify if an email is a spam or not.

So how could you spend your time to improve the accuracy of this classifier?

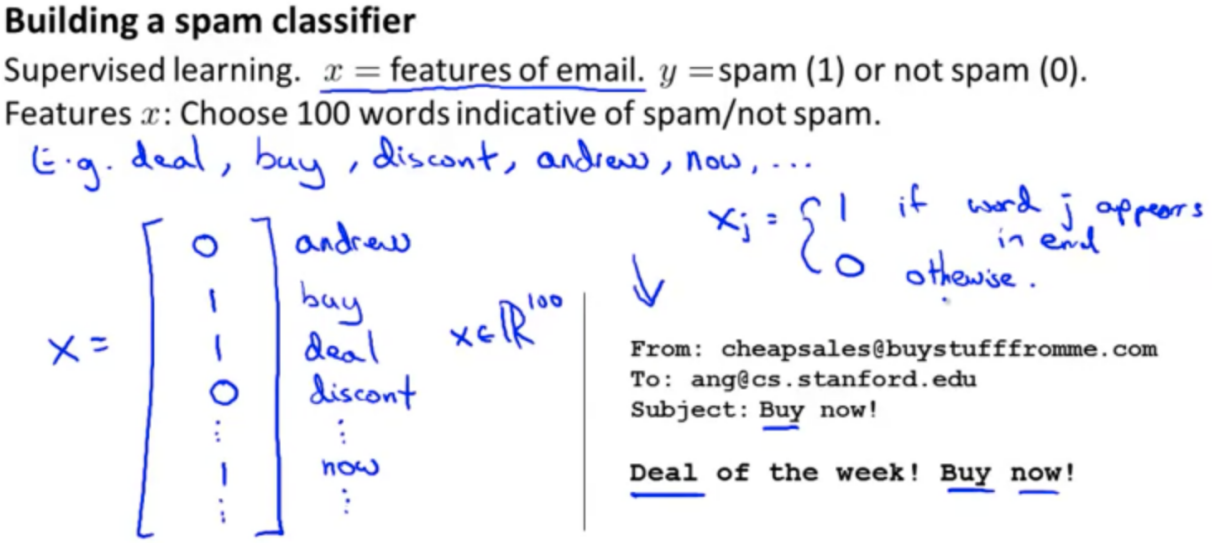

Figure 7.4: Spam vector explanation

- Collect lots of data (for example "honeypot" project but doesn't always work)
- Develop sophisticated features (for example: using email header data in spam emails)
- Develop algorithms to process your input in different ways (recognizing misspellings in spam).

It is difficult to tell which of the options will be most helpful.

7.8 Error Analysis

The recommended approach to solving machine learning problems is to:

1. Start with a simple algorithm, implement it quickly, and test it early on your cross validation data.
2. Plot learning curves to decide if more data, more features, etc. are likely to help.
3. Manually examine the errors on examples in the cross validation set and try to spot a trend where most of the errors were made.

8. Support Vector Machine (SVM)

Support vector machine is another simple algorithm that every machine learning expert should have in his/her arsenal. Support vector machine is highly preferred by many as it produces significant accuracy with less computation power. Support Vector Machine, abbreviated as SVM can be used for both regression and classification tasks. But, it is widely used in classification objectives.

If we look at the cost function of logistic regression, what we'll find is that each example (x,y) contributes a term to the overall cost function.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\Big[y^{(i)} \cdot log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \cdot log(1 - h_\theta(x^{(i)}))\Big]$$

When y is equal to one we get the left term:

$$-y \cdot log\frac{1}{1+e^{-\theta^T x}}$$

And if we plot this function as a function of z, what we find is that we get this curve shown on the lower left of the next figure. And thus, we also see that when z is equal to large, that is, when theta transpose x is large, that corresponds to a value of z that gives us a fairly small value, a very small contribution to the consumption. And this kinda explains why, when logistic regression sees a positive example, with y=1, it tries to set theta transport x to be very large because that corresponds to this term, in the cross function, being small.

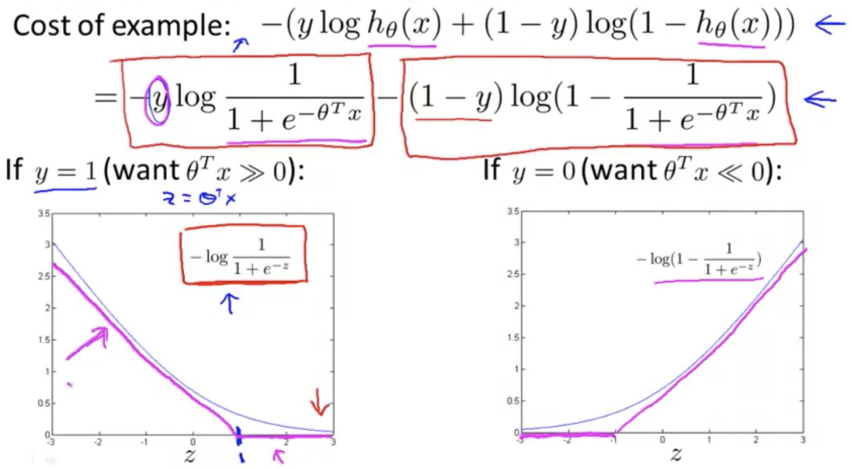

Figure 8.1: Logistic Cost example

Now, to fill the support vector machine, here's what we're going to do. We're gonna take this cross function, this on the left, and modify it a little bit. The new pass functions (drew in magenta) can be flat from here on out, and then we draw something that grows as a straight line, similar to logistic regression. But this is going to be a straight line at this portion. So the curve that I just drew in magenta, and the curve I just drew in purple and magenta, so it's pretty close approximation to the cross function used by logistic regression. Except it is now made up of two line segments, there's this flat portion on the right, and then there is this straight line portion on the left.

That's the new cost function we're going to use for when y is equal to one, and you can imagine it should do something pretty similar to logistic regression. But turns out, that this will give the support vector machine computational advantages and give us, later on, an easier optimization problem.

The other case is if y is equal to zero. In that case, if you look at the cost, then only the second term will apply because the first term goes away, right? If y is equal to zero, then you have a zero here, so you're left only with the second term of the expression. And for the support vector machine, once again, we're going to replace this blue line with something similar and at the same time we replace it with a new cost, this flat out here, this 0 out here. And that then grows as a straight line.

We're going to call this function on the left $Cost_1(z)$ and this function of the right we're gonna call $Cost_0(z)$.

The subscript just refers to the cost corresponding to when y is equal to 1, versus when y Is equal to zero.

So we have this SVM Cost function :

$$min_\theta \; C\sum_{i=1}^{m}\Big(y^{(i)}Cost_1(\theta^T x) + (1-y^{(i)})Cost_0(\theta^T x)\Big) + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2 \tag{8.1}$$

8.1 Large Margin Intuition

Let's think about what it takes to make these cost functions small. If you have a positive example, (y is equal to 1), then $Cost_1(z)$ is zero only when $z \geq 1$. So in other words, if you have a positive example, we really want theta transpose x to be greater than or equal to 1 and conversely if y is equal to zero, look this $Cost_0(z)$ then it's only in this region where $z \leq -1$ we have the cost is zero as z is equals to zero.

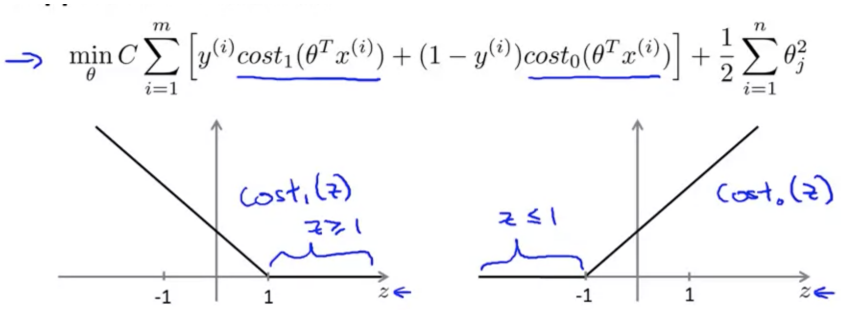

Figure 8.2: Large Marge Intuition

In logistic regression, we take the output of the linear function and squash the value within the range of [0,1] using the sigmoid function. If the squashed value is greater than a threshold value(0.5) we assign it a label 1, else we assign it a label 0. In SVM, we take the output of the linear function and if that output is greater than 1, we identify it with one class and if the output is -1, we identify is with another class. Since the threshold values are changed to 1 and -1 in SVM, we obtain this reinforcement range of values([-1,1]) which acts as margin.

Using these support vectors, we maximize the margin of the classifier. Deleting the support vectors will change the position of the hyperplane (hypotheses). These are the points that help us build our SVM.

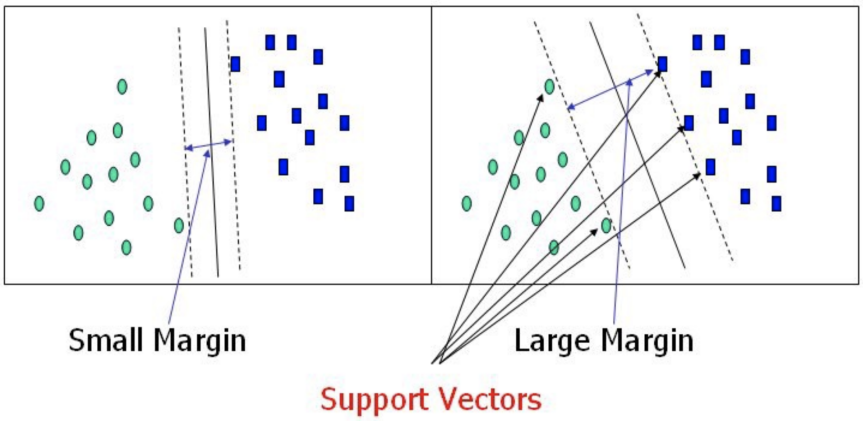

Figure 8.3: Large Marge Intuition