Machine Learning

Nagubandi Krishna Sai
MS20BTECH11014

2021-06-19

Contents

1 Fundamentals of Machine Learning **3**

1.1 Supervised Learning . . . 3

1.1.1 Linear Regression . . . 3

1.1.2 Hypothesis for Linear Regression . . . 4

1.1.3 Cost function for Linear Regression . . . 4

1.1.4 Gradient descent for Linear Regression . . . 5

1.1.5 Linear Regression for multivariables . . . 6

1.1.6 Polynomial regression . . . 6

1.1.7 Normal Equation . . . 7

1.1.8 Classification . . . 8

1.1.9 Logistic Regression Model . . . 9

1.1.10 Interpretation of hypothesis output for Logistic Regression 9

1.1.11 Cost function for Logistic Regression . . . 11

1.1.12 Gradient Descent for Logistic Regression . . . 14

1.1.13 Optimization algorithm . . . 14

1.1.14 Multiclass Classification : One-vs-All . . . 15

1.1.15 Problem of Overfitting, . . . 16

1.1.16 Regularization . . . 16

1.1.17 Regularized Logistic Regression . . . 17

1.2 Neural Networks . . . 18

1.2.1 Model representation I . . . 18

1.2.2 Model representation II . . . 19

1.2.3 Multiclass Classification . . . 22

1.2.4 Cost function for Neural networks . . . 25

1.2.5 Backpropagation Algorithm . . . 26

1.2.6 Gradient Checking . . . 29

1.2.7 Random Intialization . . . 29

1.2.8 Choosing Neural network . . . 30

1.2.9 Training a Neural Network . . . 30

1.2.10 Neural network Architecture . . . 31

2 Deciding whether the algorithm is perfect or not **32**
2.1 Evaluating the algorithm . . . 32
2.1.1 Evaluating a Hypothesis . . . 32
2.1.2 The test set error . . . 34
2.1.3 Model Selection and Train/ Validation/ Test Sets . . . 34
2.1.4 Diagnosing Bias vs Variance . . . 36
2.1.5 Regularization . . . 36
2.1.6 Learning Curves . . . 38
2.1.7 Deciding What to Do Next Revisited . . . 39
2.1.8 Example on Building a spam classifier . . . 41
2.1.9 Error Analysis . . . 42
2.1.10 Error metrics for skewed classes . . . 44
2.1.11 Trading of precision and recall . . . 48
2.1.12 F Score . . . 49

3 Support Vector Machine (SVM) **54**
3.0.1 How does SVM work? . . . 54
3.0.2 SVM for Non-linear data . . . 59
3.0.3 Hyperplane . . . 61
3.0.4 Tuning parameters . . . 62
3.0.5 Algorithm . . . 63
3.0.6 The Kernel . . . 67
3.0.7 Differences between SVM and Logistic Regression . . . 68
3.0.8 Optimization Objective . . . 70
3.0.9 Large Margin Intuition . . . 72

Chapter 1

Fundamentals of Machine Learning

1.1 Supervised Learning

Supervised Learning gives "correct answers", the output values are same as real life values.
In Supervised Learning, we are given a set of data and we know what our correct output should look like, having an idea that there is relationship between the input and the output.
Supervised Learning problems has two types of problems,

1. Regression.
2. Classification.

1.1.1 Linear Regression

In regression type of problems, we are trying to predict results within a continuous output, means that we are trying to map input variables to some continuous function.
Linear regression has real-valued output, but the output will be same or near valued to the actual output.

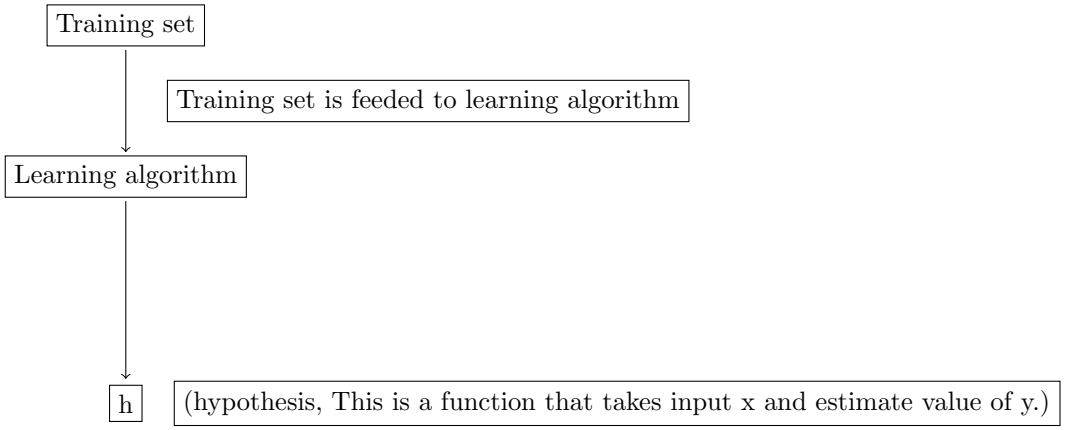
1. m = Number of training examples.
2. x's = "input" variable (or) feature.
3. y's = "output (or) target" variable.
4. (x,y) = one training example.
5. $(x^{(i)},y^{(i)}) = i^{th}$ training example.

Size of feet$^2(x)$	Price($) in 1000's (y)
2104	460
1416	232
1534	315
852	178
$\vdots$	$\vdots$

Table 1.1: Training set of housing prices.

Example :

1. m = 47.
2. $x^{(1)} = 2104$ and $y^{(1)} = 460$.
3. $x^{(2)} = 1416$ and $y^{(2)} = 232$.



1.1.2 Hypothesis for Linear Regression

$$h_\theta(x) = \theta_0 + \theta_1.x \tag{1.1}$$

θ_i's = parameters.

This type of hypothesis mode is called "Linear regression with one variable" (or) "Univariate linear regression."

1.1.3 Cost function for Linear Regression

Cost function helps us know that how well to fit the best possible straight line over the given data.

Q⟩ *How to choose* θ_i's ?

1. Choose θ_i 's so that $h_\theta(x)$ is close to y for our training example (x,y).
2. minimise θ_0,θ_1 so, that [$h_\theta(x)$ - y] is small.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} [h_\theta(x^{(i)}) - y^{(i)}]^2 \tag{1.2}$$

J(θ_0,θ_1) = Cost function (or) Squared error function.

1.1.4 Gradient descent for Linear Regression

Gradient descent is used to minimise cost function(J) in linear regression.
Gradient descent is used in many areas to minimise many functions in ML/AI.
Gradient descent algorithm,

$$\text{Repeat until convergence (minimum)} \left\{ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1),\ for\ j = 0, j = 1. \right. \tag{1.3}$$

1. := is Assignment operator.
2. α is learning rate.
3. $\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ is derivative.

Gradient descent is nothing but the derivative of the Cost function.

$$Slope\ of\ cost\ function\ curve = \frac{\partial J(\theta_1)}{\partial \theta_1},\ when\ \theta_0 = 0. \tag{1.4}$$

Learning rate,

1. If α is too small, gradient descent can be slow. After many such operations(can be infinite times), the 'θ_1' could reach "global minimum".
2. If α is too large, gradient descent can overshoot the minimum. It may "fail to converge (or) even diverge".
3. If θ_1 is at the local optima itself when we started or taken θ_1, then there is no use of "α (or) gradient descent".

1.1.5 Linear Regression for multivariables

Hypothesis,

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n. \tag{1.5}$$

In the total context of Supervised learning, hypothesis is just predicting the output.
For convenience of notation, declare $x_0 = 1$ ($x_0^{(i)} = 1$).

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \epsilon \Re^{n+1} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \epsilon \Re^{n+1}$$

The above matrix is 0 - indexed.

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n . h_\theta(x) = \theta^\top x. \tag{1.6}$$

Cost function,

$$J(\theta) = J(\theta_0, \theta_1, \theta_2, ..., \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} [h_\theta(x^{(i)}) - y^{(i)}]^2 \tag{1.7}$$

Gradient descent,

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1, \theta_2, ..., \theta_n) \tag{1.8}$$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \tag{1.9}$$

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} [h_\theta(x^{(i)}) - y^{(i)}] x_j^{(i)} \tag{1.10}$$

Feature scaling,
Get every feature into approximately -1$\leq x_i \leq$ 1 *range.*
Mean normalization,
Replacex_i *with* $x_i - \mu_i$ *to make features have approximately zero mean.*

1.1.6 Polynomial regression

Hypothesis,

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3. \tag{1.11}$$

Housing price prediction
$x_1 = size$
$x_2 = (size)^2$
$x_3 = (size)^3$

Table 1.2: Features be-like in Polynomial regression.

1.1.7 Normal Equation

Intuition,

$$\frac{d}{d\theta} J(\theta) = 0.$$

Cost function,

$$\theta \, \epsilon \, \Re^{n+1}, \; J(\theta_0, \theta_1, \theta_2, ..., \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} [h_\theta(x^{(i)}) - y(i)]^2 \tag{1.12}$$

$$\frac{\partial}{\partial \theta_j} J(\theta) = 0, \; (solve \; for \; \theta_0 \; , \theta_1 \; , ... \; , \theta_n) \tag{1.13}$$

Example,

	Size (feet^2)	No.of Bed rooms	No.of floors	Age of home (years)	Price($1000)
x_0	x_1	x_2	x_3	x_4	y
1	2104	5	1	45	460
1	1416	3	2	40	232
1	1534	3	2	30	315
1	1852	2	1	36	178

Table 1.3: Sample Training set for Multi-Variate Linear regression.

1. n = number of Features.
2. $x_j^{(i)}$ = value of j in the i^{th} training example.
3. $x_{(i)}$ = the input(features) of the i^{th} training example.

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix} \qquad Y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

X is m×(n+1)-dimensional matrix and Y is a m-dimensional vector.

$$\theta = (X^\top X)^{-1} X^\top Y \tag{1.14}$$

The above θ value is optimal θ value.
For Normal equation method, then no need to use **feature scaling.**
We use 'Gradient descent' and 'Normal equation' methods to minimise cost function.

Gradient descent	Normal equation
1⟩ Need to choose 'α'.	1⟩ No need to choose 'α'.
2⟩ Need many iterations.	2⟩ Don't need to iterate.
3⟩ Works well even, when 'n' is large. (n¿10000)	3⟩ Need to compute n× n matrix inverse $(X\top X)^{-1}$
	4⟩ Works Now if n is very large.

Table 1.4: Why should we use the particular method? Advantages and Disadvantages of two methods.

$$\theta = (X^\top X)^{-1} X^\top Y \tag{1.15}$$

Q⟩ What is $X^\top X$ is non-invertible(singular/degenerate) ?
Reasons,

1. Redundant features (linearly dependent)
   - x_1 = Size in $feet^2$
   - x_2 = Size in m^2
   - $x_2 = (3.28)^2\ x_1$, 1m = 3.28feet.
2. Too many features. (m≤n)
   - m = 10
   - n = 100, $\theta\ \epsilon\ \Re^{101}$, Delete some features (or) Regularization.

1.1.8 Classification

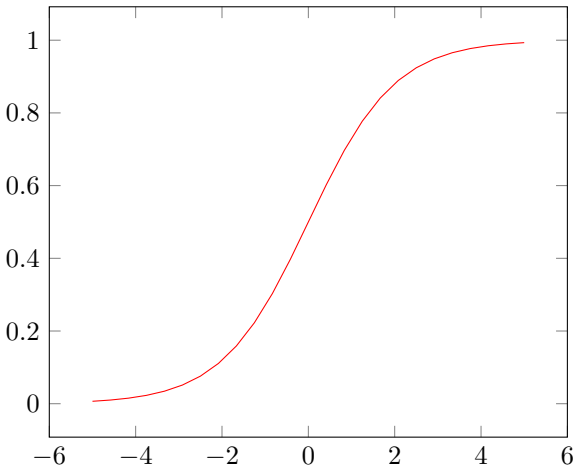
The output value 'y' is **discrete value.**
The algorithm used is **logistic regression.**

1.1.9 Logistic Regression Model

We want $0 \leq h_\theta(x) \leq 1$.

$$h_\theta(x) = g(\theta^\top x) \tag{1.16}$$
$$g(z) = \frac{1}{1+e^{-z}} \tag{1.17}$$
$$h_\theta(x) = g(\theta^\top x) \tag{1.18}$$
$$= \frac{1}{1+e^{-\theta^\top x}} \tag{1.19}$$

The above g(z) is called sigmoid function (or) logistic function.
Graph,



$$g(z) \geq 0.5, \ when \ z \geq 0. \tag{1.20}$$
$$h_\theta(x) = g(\theta^\top x) \geq 0.5, \ when \ \theta^\top x \geq 0. \tag{1.21}$$

1.1.10 Interpretation of hypothesis output for Logistic Regression

$$h_\theta(x) = P(y = 1|x;\theta) \tag{1.22}$$
$$P(y = 0|x;\theta) + P(y = 1|x;\theta) = 1 \tag{1.23}$$
$$P(y = 0|x;\theta) = 1 - P(y = 1|x;\theta) \tag{1.24}$$
$$= 1 - h_\theta(x) \tag{1.25}$$
$$\tag{1.26}$$

y = 0 (or) 1.

Decision boundary,

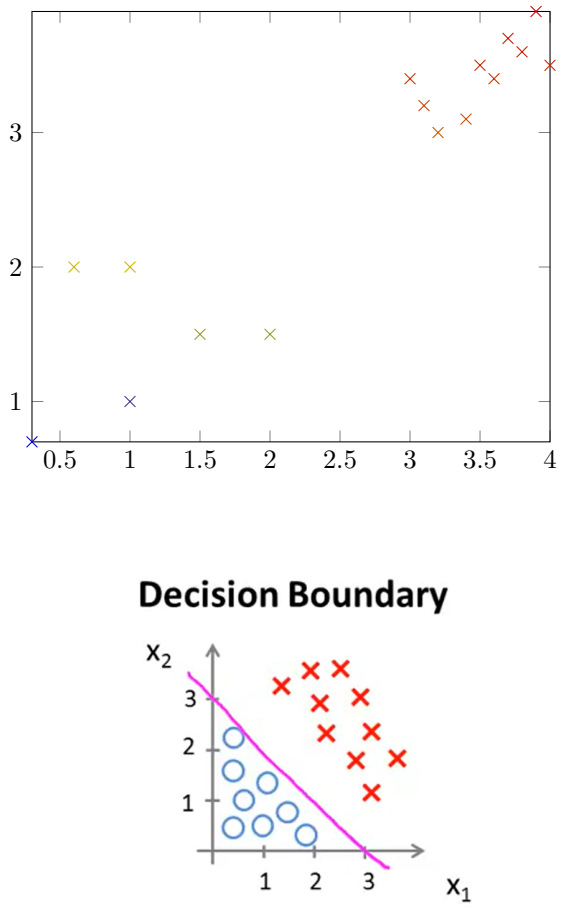


Figure 1.1: A simple decision boundary looks like.

For, the above diagram decision boundary will be a line separating the two output values of y(y=0 (or) y=1).

$$h_\theta(x) \geq 0.5 \rightarrow y = 1. \tag{1.27}$$
$$h_\theta(x) < 0.5 \rightarrow y = 0. \tag{1.28}$$
$$h_\theta(x) = g(\theta^\top x) \tag{1.29}$$
$$= g(\theta_0 + \theta_1 x_1 + \theta_2 x_2) \tag{1.30}$$

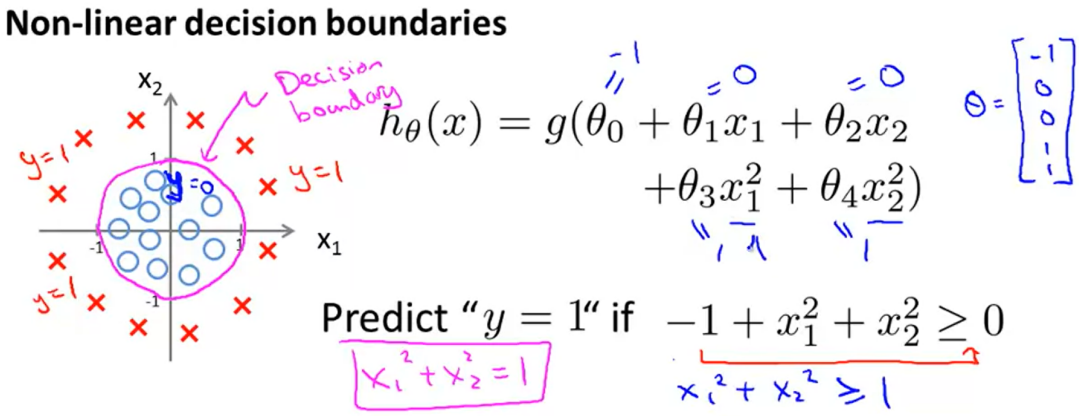
$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} \qquad x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$$

Example,
Let,

$$\theta = \begin{bmatrix} -3 \\ 1 \\ 1 \end{bmatrix}$$

$$y = 1, if\ \theta^\top x \geq 0 \tag{1.31}$$
$$-3 + x_1 + x_2 \geq 0 \tag{1.32}$$
$$x_1 + x_2 \geq 3,\ y = 1 \tag{1.33}$$
$$x_1 + x_2 < 3,\ y = 0. \tag{1.34}$$

Non-Linear Decision Boundary,



So, in the above non-linear classification, the **decision boundary is a circle** of radius of 1unit.

$$Inside\ circle,\ y = 0 \tag{1.35}$$
$$Outside\ circle,\ y = 1. \tag{1.36}$$

1.1.11 Cost function for Logistic Regression

Training set,

$x^{(1)}$	$y^{(1)}$
$x^{(2)}$	$y^{(2)}$
$x^{(3)}$	$y^{(3)}$
$\vdots$	$\vdots$
$x^{(m)}$	$y^{(m)}$

Table 1.5: Training set of m-examples for Logistic Regression

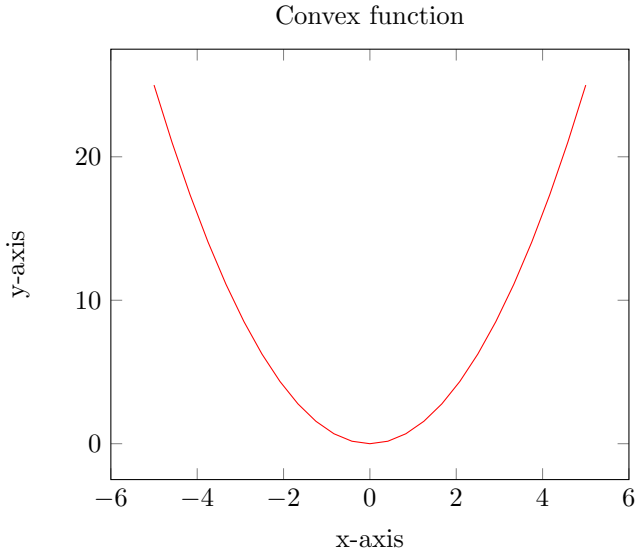
$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \epsilon \Re^{n+1} \; - \; n \; features. \; x_0 = 1, \; y \; \epsilon \; 0, 1.$$

For linear regression,

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} [h_\theta(x^{(i)} - y^{(i)}]^2 \tag{1.37}$$

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost(h_\theta(x^{(i)}, y^{(i)}) \tag{1.38}$$

$$Cost(h_\theta(x^{(i)}), y^{(i)}) = \frac{1}{2}[h_\theta(x^{(i)} - y^{(i)}]^2 \tag{1.39}$$



The above graph is a convex.

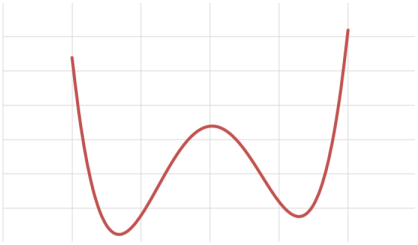
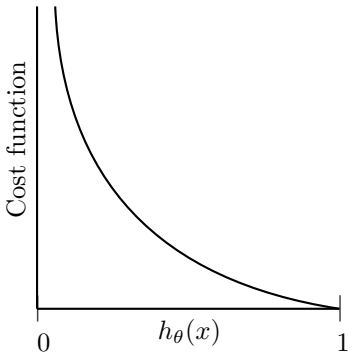


Figure 1.2: Graph of non-convex function.

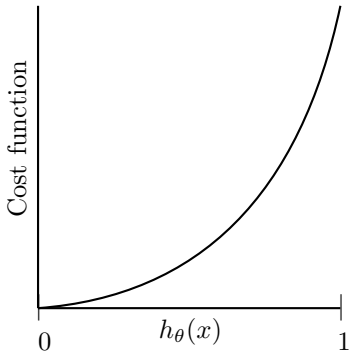
The below graph is a non-convex.
If we use the cost function of linear regression in logistic regression, the the we would get non-convex cost function, because the **hypothesis is** $\frac{1}{1+e^{-\theta^\top x}}$.
For Logistic regression,

$$Cost(h_\theta(x^{(i)}), y^{(i)}) = \begin{cases} -\log(1 - h_\theta(x)), & if\ y = 0 \\ -\log(h_\theta(x)), & if\ y = 1. \end{cases} \tag{1.40}$$

If y=1,



If y=0,



Simplified Cost function,

$$Cost(h_\theta(x^{(i)}), y^{(i)}) = -(1-y)\log(1-h_\theta(x)) - y\log(h_\theta(x)).\ \forall\ y\ \epsilon\ \{0,1\}. \tag{1.41}$$

$$If\ y = 1 : Cost(h_\theta(x), y) = -\log(h_\theta(x)). \tag{1.42}$$

$$If\ y = 0 : Cost(h_\theta(x), y) = -\log(1-h_\theta(x)). \tag{1.43}$$

$$Costfunction = J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}, y^{(i)}) \tag{1.44}$$

$$= \frac{1}{m}[-\sum_{i=1}^{m}(1-y^{(i)})\log(1-h_\theta(x^{(i)})) + y^{(i)}\log(h_\theta(x^{(i)}))] \tag{1.45}$$

1.1.12 Gradient Descent for Logistic Regression

$$\text{Repeat until convergence (minimum)}\begin{cases}\theta_j := \theta_j - \alpha\dfrac{\partial}{\partial\theta_j}J(\theta) \end{cases} \tag{1.46}$$

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[h_\theta(x^{(i)}) - y^{(i)}]x_j^{(i)} \tag{1.47}$$

1.1.13 Optimization algorithm

1. Gradient descent.
2. Conjugate gradient.
3. BFGS.
4. L - BFGS.

These are the 4 algorithms to minimise **cost function.**
Advantages of the **last three advanced optimization algorithm.**

- No need to manually pick α.
- Often faster than Gradient descent.
- They themselves choose α, for faster convergence.

1.1.14 Multiclass Classification : One-vs-All

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

To summarize,

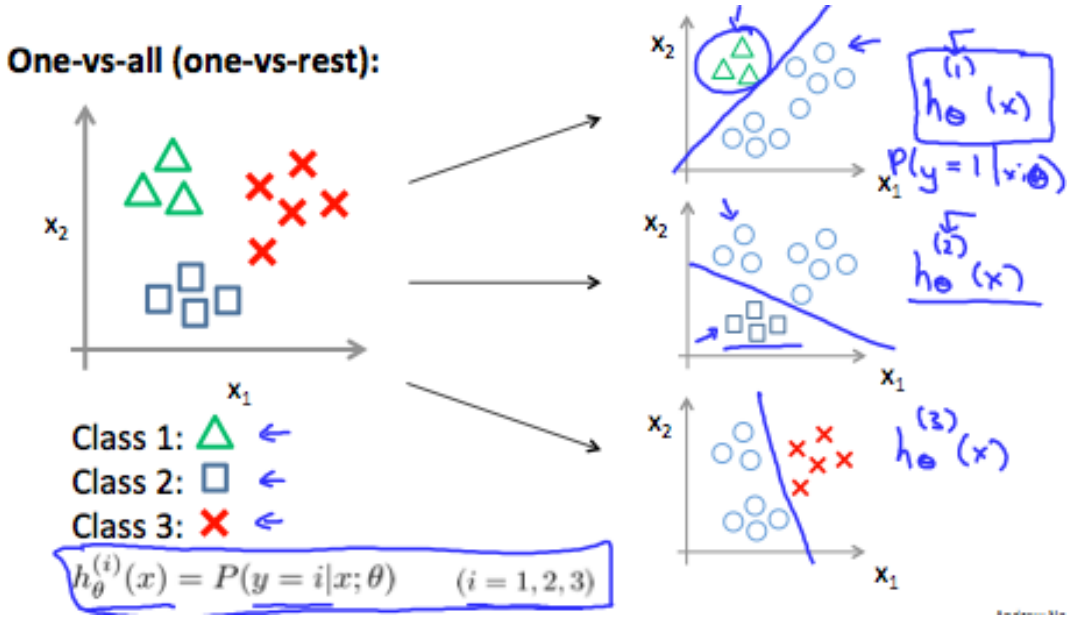


Figure 1.3: Multiclass being classified into n-classifiers for n-types of output.

1. Train a logistic regression classifier $h_\theta(x)$ for each class to predict the probability that y=i.
2. To make a prediction on a new x, pick the class that maximizes $h_\theta(x)$

1.1.15 Problem of Overfitting,

Overfitting: If we have too many features, the learned hypothesis may fit the training set very well ($J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \approx 0$), but fail to generalize to new examples (predict prices on new examples).

Figure 1.4: Example for underfit and overfit hypothesis.

Similar for Logistic Regression.

Addressing Overfitting,

1. Reduce number of features.
   - Manually select which features to keep.
   - Model selection algorithm.
2. Regularization
   - Keep all features, but reduce magnitude/values of parameters $_j$.
   - Works well when we have a lot of features, each of which contributes a bit to predict $y^{(i)}$.

1.1.16 Regularization

Small values for parameters $\theta_0, \theta_1, \theta_2, ..., \theta_n$.

1. Simpler hypothesis.

2. Less prone to overfitting.

Regularized Cost function,

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left[[h_\theta(x^{(i)} - y^{(i)}]^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] \tag{1.48}$$

If 'λ' is extremely large, then the cost function will become underfitting (doesn't fit to our training data).

Repeat {

$$\theta_0 := \theta_0 - \alpha\ \frac{1}{m}\ \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

$$\theta_j := \theta_j - \alpha\ \left[\left(\frac{1}{m}\ \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j\right] \qquad j \in \{1, 2...n\}$$

}

The term $\frac{\lambda}{m}\theta_j$ performs our regularization. With some manipulation our update rule can also be represented as:

$$\theta_j := \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \tag{1.49}$$

The first term in the above equation, $1 - \alpha\frac{\lambda}{m}$ will always be less than 1. Intuitively you can see it as reducing the value of θ_j by some amount on every update. Notice that the second term is now exactly the same as it was before.
Normal equation after regularization,

$$\theta = (X^\top X) + \lambda.L^{-1}X^\top Y \tag{1.50}$$

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

1.1.17 Regularized Logistic Regression

Cost function,

$$J(\theta) = \frac{1}{m}\Big[-\sum_{i=1}^{m}(1 - y^{(i)})\log(1 - h_\theta(x^{(i)})) + y^{(i)}\log(h_\theta(x^{(i)}))\Big] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{1.51}$$

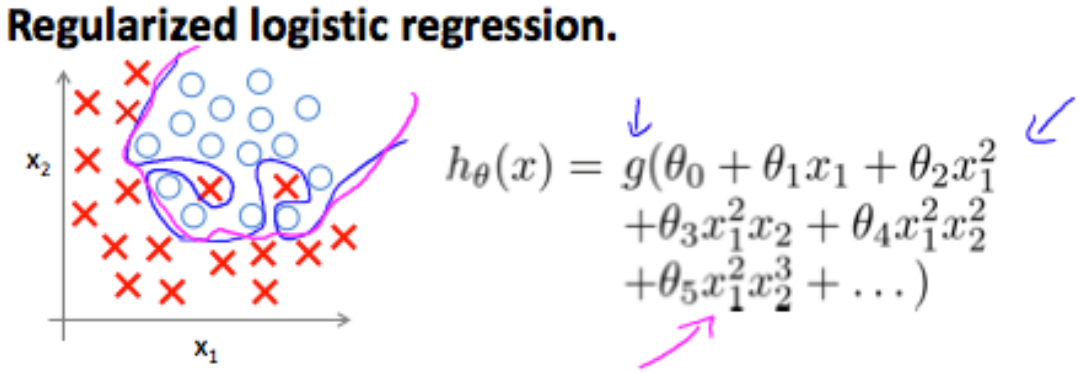


Figure 1.5: Regularization of a overfit data.

The second sum, $\sum_{j=1}^{n} \theta_j^2$ means to explicitly exclude the bias term, θ_0. I.e. the 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. Thus, when computing the equation, we should continuously update the two following equations:

Repeat {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$$

$$\theta_j := \theta_j - \alpha \left[\left(\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right] \qquad j \in \{1, 2 \dots n\}$$

}

1.2 Neural Networks

1.2.1 Model representation I

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 $x_1 \cdots x_n$, 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".**
A simple representation looks like :

$$[\ x_0 x_1 x_2\ ] \rightarrow [\ ] \rightarrow h_\theta(x)$$

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."**
In this example, we label these intermediate or **"hidden"** layer nodes $a_0^2, \cdots, a_n^2 a$ and call them **"activation units."**

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

If we had one hidden layer, 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 :

$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \tag{1.52}$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \tag{1.53}$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \tag{1.54}$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}) \tag{1.55}$$

This is saying that we compute our activation nodes by using a 3×4 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 $\Theta^{(2)}$ containing the weights for our second layer of nodes.
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 +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.

1.2.2 Model representation II

we'll do a vectorized implementation of the above functions. We're going to define a new variable $z_k^{(j)}$ that encompasses the parameters inside our g function.

$$a_1^{(2)} = g(z_1^{(2)}) \tag{1.56}$$
$$a_2^{(2)} = g(z_2^{(2)}) \tag{1.57}$$
$$a_3^{(2)} = g(z_3^{(2)}) \tag{1.58}$$
$$x = a^{(1)} \tag{1.59}$$
$$a^{(2)} = g(\theta^{(1)}x) \tag{1.60}$$
$$= g(\theta^{(1)}a^{(1)}) = g(z^{(2)}) \tag{1.61}$$
$$h_\theta(x) = g(\theta^{(2)}a^{(2)}) \tag{1.62}$$
$$= g(z^{(3)}) \tag{1.63}$$

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

$$z_k^{(2)} = \Theta_{k,0}^{(1)}x_0 + \Theta_{k,1}^{(1)}x_1 + \cdots + \Theta_{k,n}^{(1)}x_n \tag{1.64}$$

The vector representation of x and z^j is :

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ \vdots \\ z_n^{(j)} \end{bmatrix}$$

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

$$z^{(j)} = \theta^{(j-1)}a^{(j-1)} \tag{1.65}$$

We are multiplying our matrix $\Theta^{(j-1)}$ with dimensions $s_j\times$ (n+1)(where s_j is the number of our activation nodes) by our vector $a^{(j-1)}$ with height (n+1). This gives us our vector $z^{(j)}$ with height s_j. Now we can get a vector of our activation nodes for layer j as follows :

$$a^{(j)} = g(z^{(j)})a \tag{1.66}$$

Where our function g can be applied element-wise to our vector $z^{(j)}$
We can then add a bias unit (equal to 1) to layer j after we have computed $a^{(j)}$. This will be element $a_0^{(j)}$ and will be equal to 1. To compute our final hypothesis, let's first compute another z vector :

$$z^{(j+1)} = \Theta^{(j)}a^{(j)} \tag{1.67}$$

We get this final z vector by multiplying the next theta matrix after $\Theta^{(j-1)}$ with the values of all the activation nodes we just got. This last theta matrix $\Theta^{(j)}$

will have only one row which is multiplied by one column $a^{(j)}$ so that our result is a single number. We then get our final result with :

$$h_\Theta(x) = a^{(j+1)} = g(z^{(j+1)}) \tag{1.68}$$

Notice that in this last step, between layer j and layer j+1, we are doing exactly the same thing as we did in logistic regression. Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypothesis.

Examples,

A simple example of applying neural networks is by predicting x_1 AND x_2, which is the logical 'and' operator and is only true if both x_1 and x_2 are 1.

$$\begin{aligned} h_\Theta(x) &= g(-30 + 20x_1 + 20x_2) \\ x_1 = 0 \ \ &and \ \ x_2 = 0 \ \ then \ \ g(-30) \approx 0 \\ x_1 = 0 \ \ &and \ \ x_2 = 1 \ \ then \ \ g(-10) \approx 0 \\ x_1 = 1 \ \ &and \ \ x_2 = 0 \ \ then \ \ g(-10) \approx 0 \\ x_1 = 1 \ \ &and \ \ x_2 = 1 \ \ then \ \ g(10) \approx 1 \end{aligned}$$

So we have constructed one of the fundamental operations in computers by using a small neural network rather than using an actual AND gate. Neural networks can also be used to simulate all the other logical gates. The following is an example of the logical operator 'OR', meaning either x_1 is true or x_2 is true, or both :

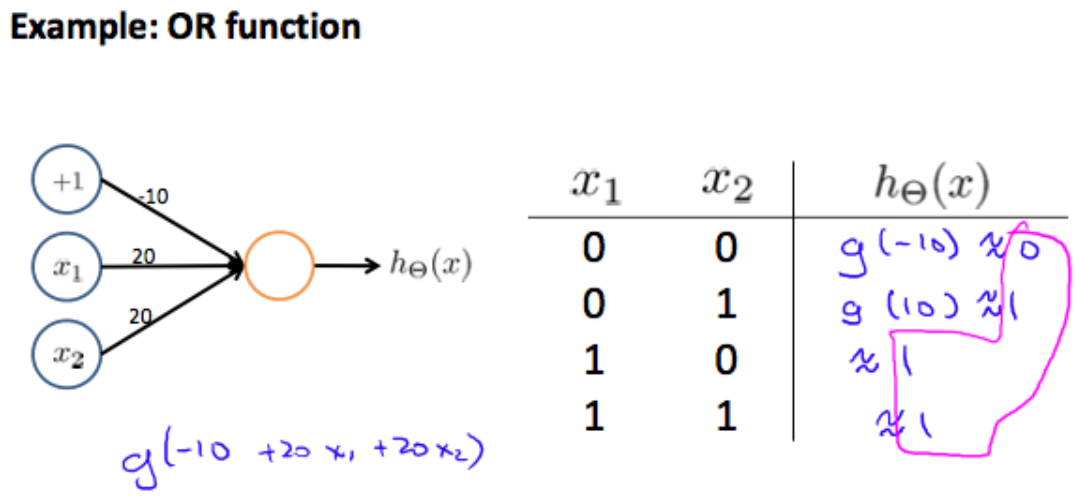


Figure 1.6: Hypothesis of neural networks is represented through logical gates.

The $^{(1)}$ matrices for AND, NOR, and OR are :

$$\begin{aligned} &AND: \\ &\quad \Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \end{bmatrix} \\ &NOR: \\ &\quad \Theta^{(1)} = \begin{bmatrix} 10 & -20 & -20 \end{bmatrix} \\ &\quad OR: \\ &\quad \Theta^{(1)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix} \end{aligned}$$

We can combine these to get the XNOR logical operator (which gives 1 if x_1 and x_2 are both 0 or both 1).

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} a^{(3)} \end{bmatrix} \rightarrow h_\Theta(x)$$

Let's write out the values for all our nodes :

$$\begin{aligned} a^{(2)} &= g(\Theta^{(1)} \cdot x) \\ a^{(3)} &= g(\Theta^{(2)} \cdot a^{(2)}) \\ h_\Theta(x) &= a^{(3)} \end{aligned}$$

1.2.3 Multiclass Classification

To classify data into multiple classes, we let our hypothesis function return a vector of values. Say we wanted to classify our data into one of four categories. We will use the following example to see how this classification is done. This algorithm takes as input an image and classifies it accordingly :

Note: As a general rule, the when the number of classes are greater than 2, then the classes should be one hot encoded to ease the process of classification. It can be seen an defining individual logistic units for determining whether or not it belongs to that class.

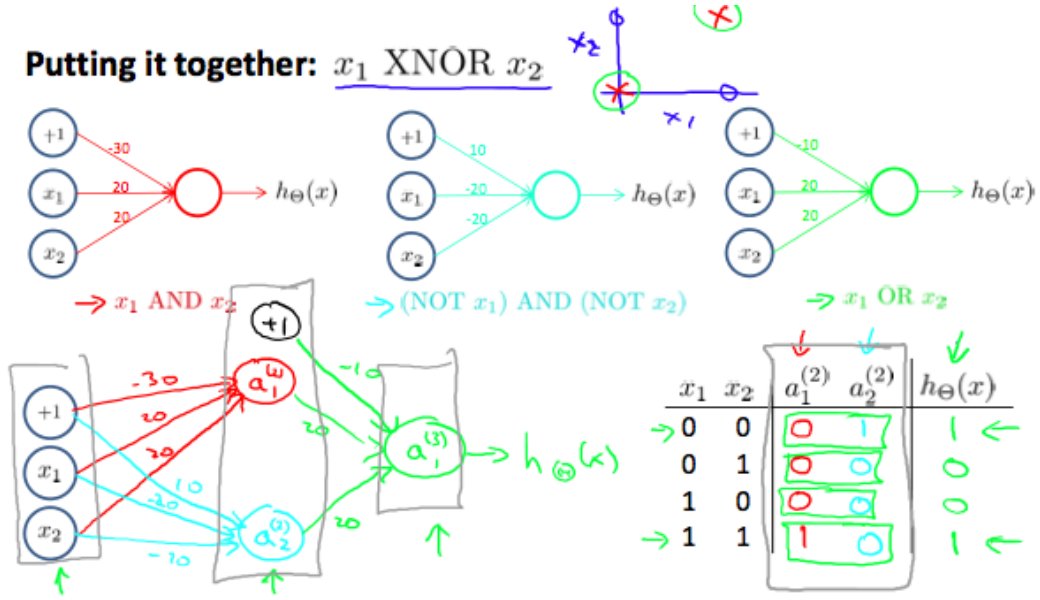


Figure 1.7: Hypothesis in XNOR logical gate.

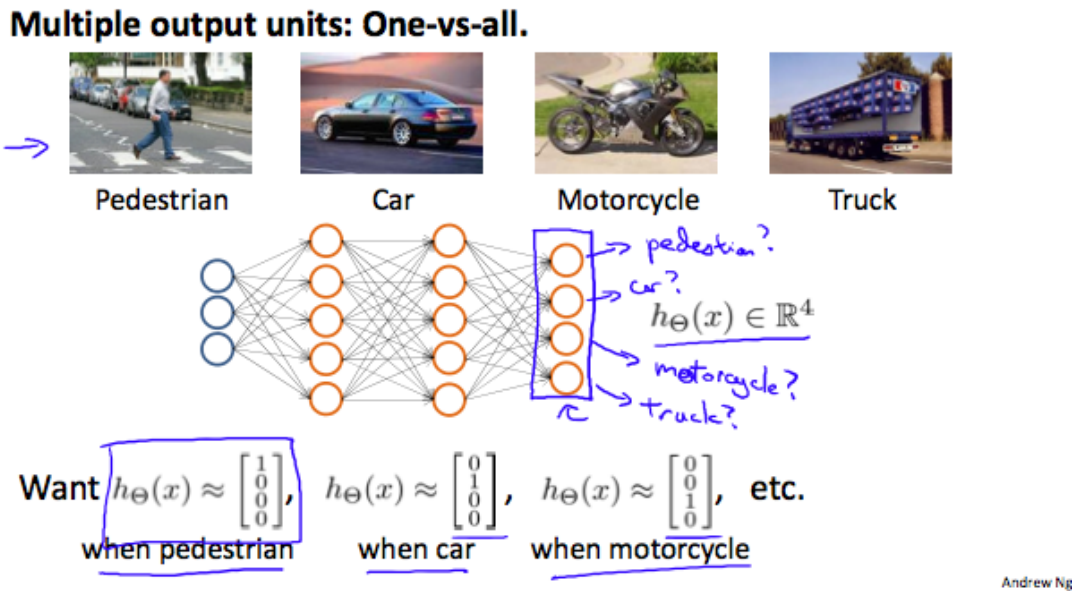


Figure 1.8: Multiclass Classification for different outputs for different things.

We can define our set of resulting classes as y :

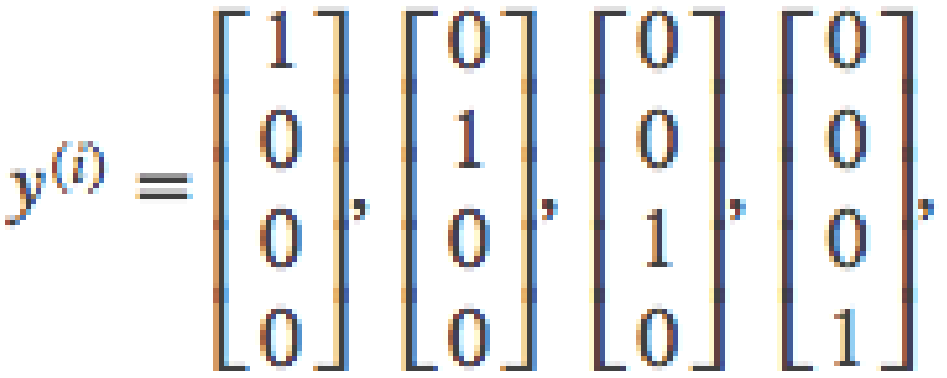


Figure 1.9: The output 'y' of neural networks.

Each $y^{(i)}$ represents a different image corresponding to either a car, pedestrian, truck, or motorcycle. The inner layers, each provide us with some new information which leads to our final hypothesis function. The setup looks like :

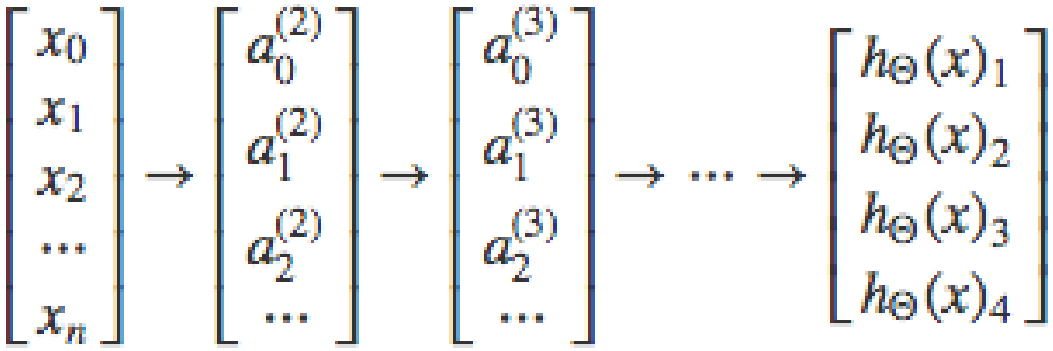


Figure 1.10: Hypothesis calculation of neural networks.

Our resulting hypothesis for one set of inputs may look like :

$$h_\theta(x) = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$$

In which case our resulting class is the third one down, or $h_\Theta(x)_3$, which represents the motorcycle.

1.2.4 Cost function for Neural networks

Let's declare some variables.

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

The cost function for regularized logistic regression,

$$J(\theta) = \frac{1}{m}\Bigg[-\sum_{i=1}^{m}(1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) + y^{(i)} \log(h_\theta(x^{(i)}))\Bigg] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{1.69}$$

The cost function for Neural networks,

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[(1 - y_k^{(i)}) \log(1 - (h_\theta(x^{(i)}))_k) + y_k^{(i)} \log((h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2 \tag{1.70}$$

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.

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

1.2.5 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 :

$$\min_{\Theta} J(\Theta) \tag{1.71}$$

That is, we want to minimize our cost function J using an optimal set of parameters in theta. In this section we'll look at the equations we use to compute the partial derivative of J() :

$$\frac{\partial}{\partial \Theta_{j,i}^{(l)}} J(\Theta) \tag{1.72}$$

Backpropagation algorithm,

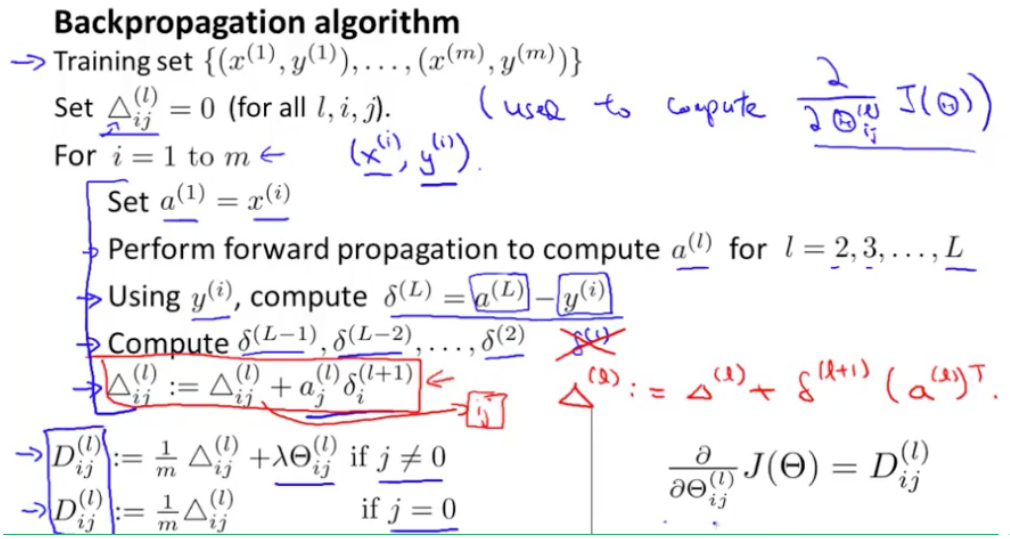


Figure 1.11: Backpropagation algorithm.

1. Given training set $\{(x^{(1)}, y^{(1)}) \cdots (x^{(m)}, y^{(m)})\}$.
   - Set $\Delta_{i,j}^{(l)} := 0$ for all (l,i,j), (hence you end up having a matrix full of zeros)
2. For training example t = 1 to m :
   - Set $a^{(1)} := x^{(t)}$.
   - Perform forward propagation to compute $a^{(l)}$ for l=2,3,...,L.

- 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 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 :

Gradient computation

Given one training example (x, y):

Forward propagation:

$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$

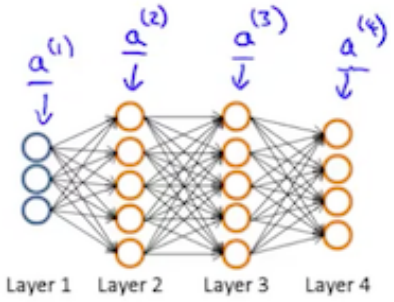


Figure 1.12: Calculation of values of activation and Θ.

- Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ 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 :

$$g'(z^{(l)}) = a^{(l)} .* (1 - a^{(l)}) \tag{1.73}$$

- $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ (or) with vectorization, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$.
  Hence we update our new Δ matrix.
  (a) $D_{i,j}^{(l)} := \frac{1}{m}(\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)})$, if j$\neq$ 0.
  $D_{i,j}^{(l)} := \frac{1}{m}(\Delta_{i,j}^{(l)}$, 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 $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$.

If we consider simple non-multiclass classification (k = 1) and disregard regularization, the cost is computed with :

$$Cost(t) = (1 - y^{(t)}) \log(1 - h_\theta(x^{(t)})) + y^{(t)} \log(h_\theta(x^{(t)})) \tag{1.74}$$

Intuitively, $\delta_j^{(l)}$ is the **"error"** for $a_j^{(l)}$ (unit j in layer l). More formally, the delta values are actually the derivative of the cost function :

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(t) \tag{1.75}$$

Our derivative is the slope of a line tangent to the cost function, so the steeper the slope the more incorrect we are. Let us consider the following neural network below and see how we could calculate some $\delta_j^{(l)}$:

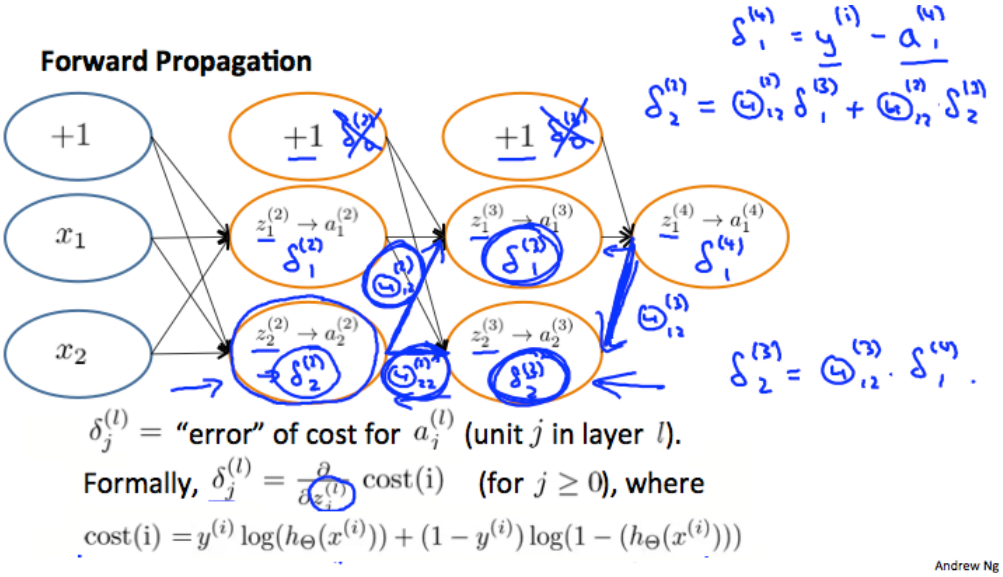


Figure 1.13: The forward propagation in backward propagation for $\delta_j^{(l)}$.

In the image above, to calculate $\delta_2^{(2)}$, we multiply the weights $\Theta_{12}^{(2)}$ and $\Theta_{22}^{(2)}$ by their respective δ values found to the right of each edge. So we get $\delta_2^{(2)} = \Theta_{12}^{(2)} * \delta_1^{(3)} + \Theta_{22}^{(2)} * \delta_2^{(3)}$. To calculate every single possible $\delta_j^{(l)}$, we could start from the right of our diagram. We can think of our edges as our Θ_{ij}. Going from right to left, to calculate the value of $\delta_j^{(l)}$, you can just take the over all sum of each weight times the δ it is coming from. Hence, another example would be $\delta_2^{(3)} = \Theta_{12}^{(3)} * \delta_1^{(4)}$.

1.2.6 Gradient Checking

Gradient checking will assure that our backpropagation works as intended. We can approximate the derivative of our cost function with :

$$\frac{\partial}{\partial\theta}J(\Theta) \approx \frac{J(\Theta+\epsilon) - J(\Theta-\epsilon)}{2\epsilon} \tag{1.76}$$

With multiple theta matrices, we can approximate the derivative with respect to Θ_j as follows :

$$\frac{\partial}{\partial\theta_j}J(\Theta) \approx \frac{J(\Theta_1, ..., \Theta_j+\epsilon, ..., \Theta_n) - J(\Theta_1, ..., \Theta_j-\epsilon, ..., \Theta_n)}{2\epsilon} \tag{1.77}$$

A small value for ϵ (epsilon) such as $\epsilon = 10^{-4}$, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems.
We previously saw how to calculate the deltaVector. So once we compute our gradApprox vector, we can check that gradApprox $\approx$ deltaVector.
Once you have verified once that your backpropagation algorithm is correct, you don't need to compute gradApprox again. The code to compute gradApprox can be very slow.

1.2.7 Random Intialization

Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our Θ matrices using the following method :

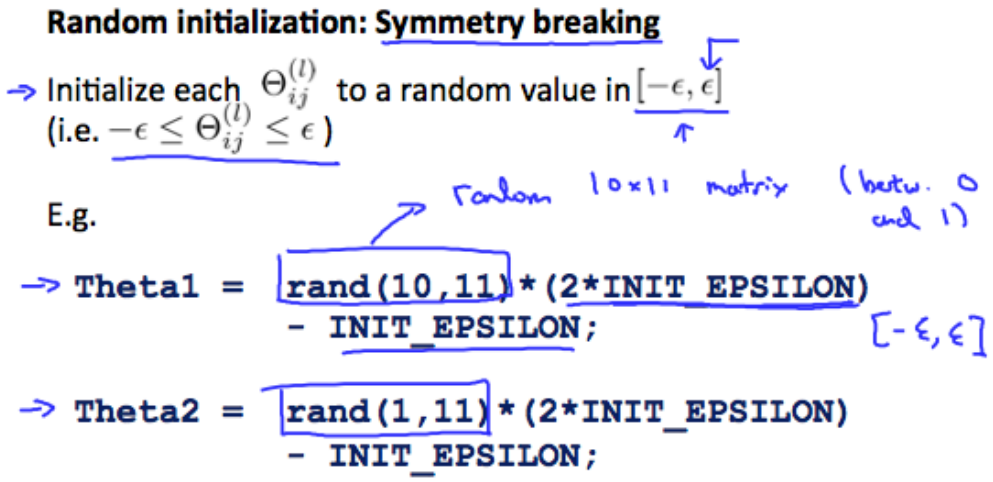


Figure 1.14: Random intialization to begin and get into the problem.

Hence, we initialize each $\Theta_{ij}^{(l)}$ to a random value between $[-\epsilon, \epsilon]$. Using the above formula guarantees that we get the desired bound.

The epsilon used above is unrelated to the epsilon from Gradient Checking.

1.2.8 Choosing Neural network

(h) Pick a network architecture.

2. Choose the layout of your neural network.
3. Including how many hidden units in each layer and how many layers in total you want to have.
   - Number of input units = dimension of features $x^{(i)}$.
   - Number of output units = number of classes.
   - Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units).
   - **Defaults** : 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

1.2.9 Training a Neural Network

1. Randomly initialize the weights.

2. Implement forward propagation to get $h_\Theta(x^{(i)})$.

3. Implement the cost function.

4. Implement backpropagation to compute partial derivatives. $(\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta))$

5. Use gradient checking to compare $(\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta))$ given by backpropagation vs the numerical estimate of gradient. S0, to confirm that your backpropagation works. Then disable gradient checking.

6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

Note: Ideally, you want $h_\Theta(x^{(i)}) \approx y^{(i)}$. This will minimize our cost function. However, keep in mind that $J(\Theta)$ is not convex and thus we can end up in a local minimum instead.

1.2.10 Neural network Architecture

This usually means to pick the connectivity pattern between the neurons.

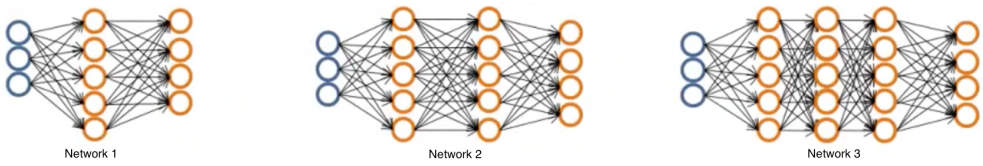


Figure 1.15: Different architectures of neural networks.

Above figure shows a few examples of network architectures. It can be seen that the three networks have the same number of input and output units. This is so because the input units equals input features in a given training example while the output units equals the number of target classes.

Note: For neural networks the cost function, $J(\Theta)$ is non-convex and hence susceptible to local minima. But in practice it does not present a serious problem in the implementation.

Chapter 2

Deciding whether the algorithm is perfect or not

2.1 Evaluating the algorithm

2.1.1 Evaluating a Hypothesis

Fails to generalize to new examples not in training set.
Once we have done some trouble shooting for errors in our predictions by :

1. Getting more training examples.
2. Trying smaller sets of features.
3. Trying additional features.
4. Trying polynomial features.
5. 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 :

Dataset,
Model selection,

Size	Price		
2104	400		$(\mathrm{x}^{(1)}, y^{(1)})$
1600	330		$(\mathrm{x}^{(2)}, y^{(1)})$
2400	369		$(\mathrm{x}^{(3)}, y^{(1)})$
1416	232	$\rightarrow$	$\vdots$
3000	540		$\vdots$
1985	300		$\vdots$
1534	315		$(\mathrm{x}^{(m)}, y^{(m)})$
1427	199		$(\mathrm{x}^{(1)}_{test}, y^{(1)}_{test})$
1380	212	$\longmapsto$	$(\mathrm{x}^{(2)}_{test}, y^{(2)}_{test})$
1494	243		$(\mathrm{x}^{(m_{test})}_{test}, y^{(m_{test})}_{test})$

Table 2.1: Evaluating your hypothesis

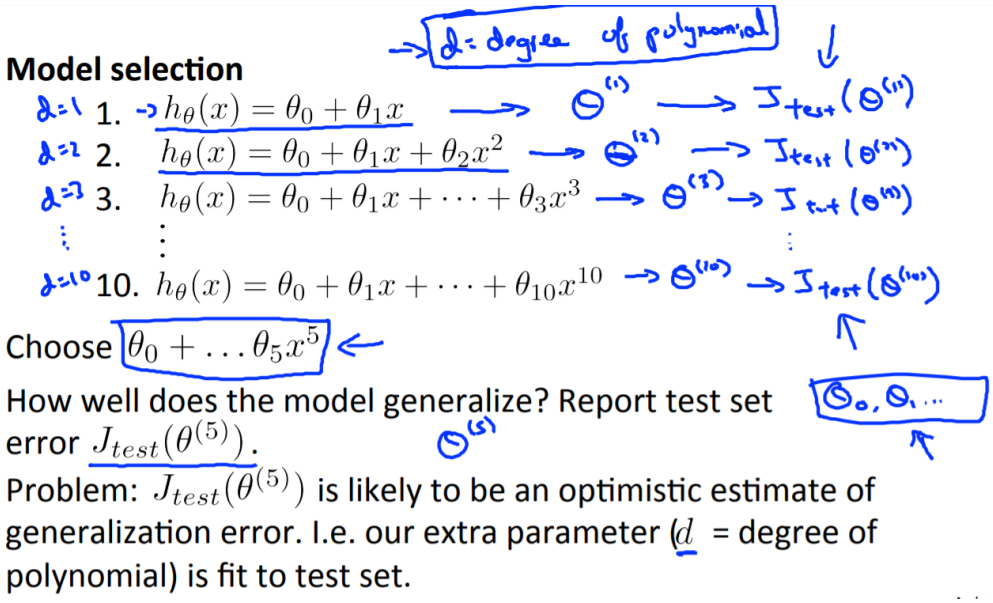


Figure 2.1: Model selectio process.

1. m_{test} = number of test example.
2. Learn Θ and minimize $J_{train}(\Theta)$ using the training set.
3. Compute the test set error $J_{test}(\Theta)$.

2.1.2 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 Misclassification error (aka 0/1 misclassification error):

$$err(h_\Theta(x), y) = \begin{cases} 1 & if\ h_\Theta(x) \geq 0.5\ and\ y = 0\ or\ h_\Theta(x) < 0.5\ and\ y = 1 \\ 0 & otherwise \end{cases} \tag{2.1}$$

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

$$\text{Test Error} = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err(h_\Theta(x_{test}^{(i)}), y_{test}^{(i)}) \tag{2.2}$$

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

Train/validation/test error

Training error:

$$\rightarrow J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Cross Validation error:

$$\rightarrow J_{cv}(\theta) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

Test error:

$$\rightarrow J_{test}(\theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_\theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$$

Figure 2.2: Cost function for training, cross-validation and test error.

2.1.3 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 :

1. Training set: 60%.
2. Cross validation set: 20%.
3. Test set: 20%.

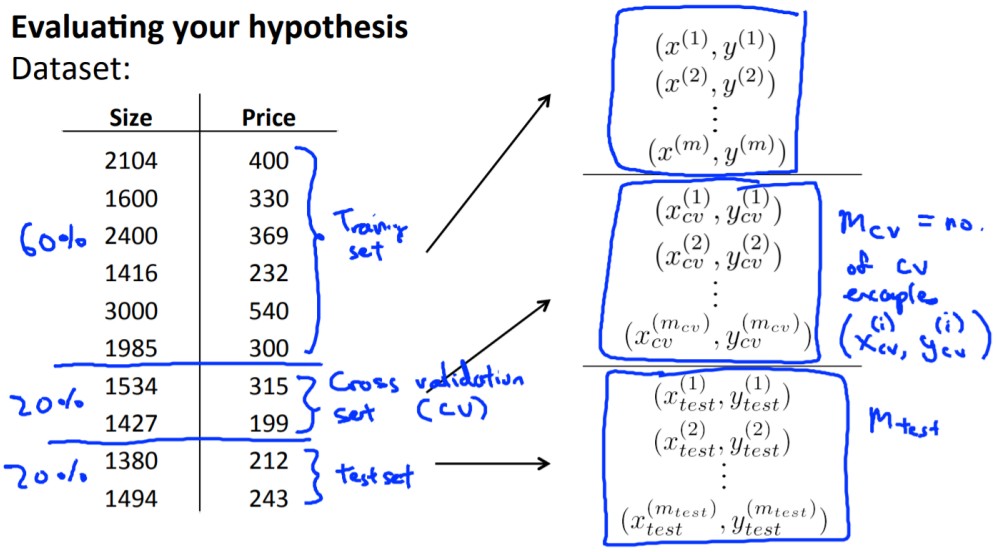


Figure 2.3: Separation of dataset into trsin, cross-validation and test sets.

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

1. Optimize the parameters in 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.

2.1.4 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.
- High bias is underfitting and high variance is overfitting. Ideally, we need to find a golden mean between these two.

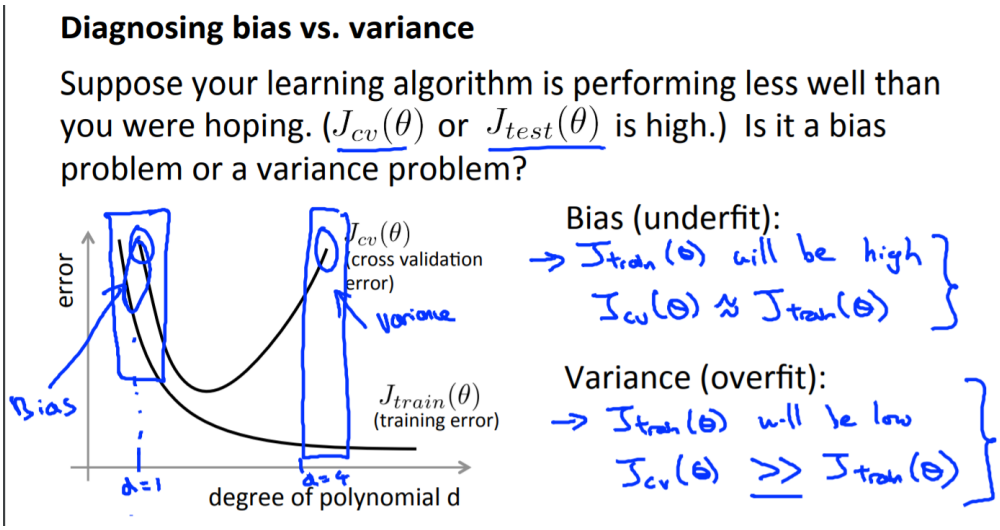


Figure 2.4: Diagnosing bias and variance.

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 $J_{train}(\Theta)$.

2.1.5 Regularization

In the figure below, we see that as λ increases, our fit becomes more rigid. On the other hand, as λ approaches 0, we tend to over 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:

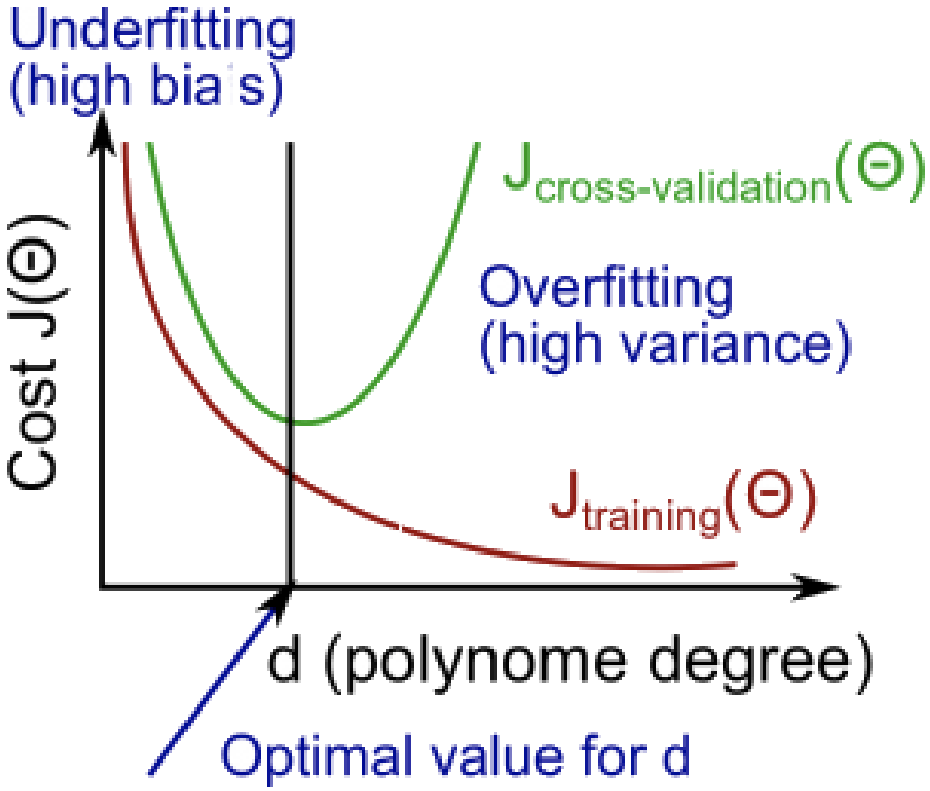


Figure 2.5: graph for optimal value of 'd'.

1. Create a list of lambdas (i.e. λ 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 λs 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 and , apply it on $J_{test}(\Theta)$ to see if it has a good generalization of the problem.

Machine Learning Diagnostics are tests that help gain insight about what would or would not work with a learning algorithm, and hence give guidance about how to improve the performance. These can take time to implement, but are still worth venturing into during the time of uncertainties.

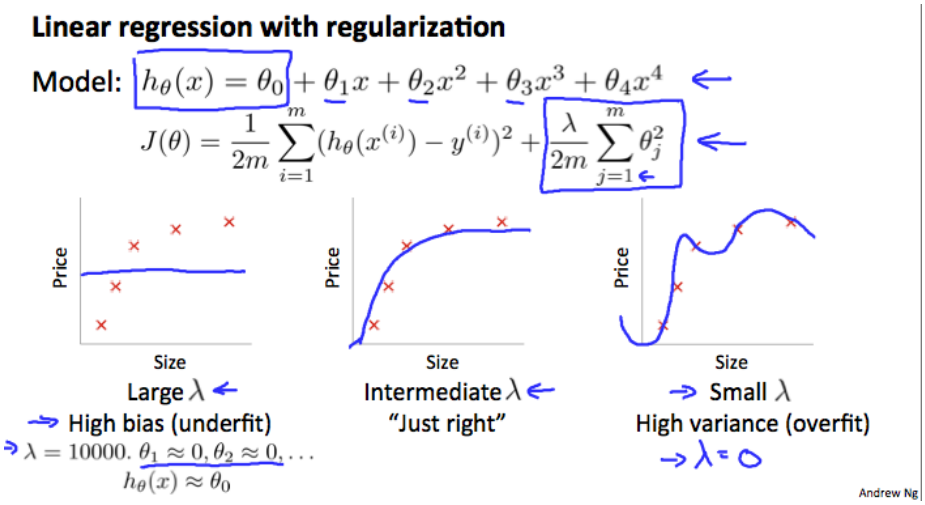


Figure 2.6: Regularization for Linear regression.

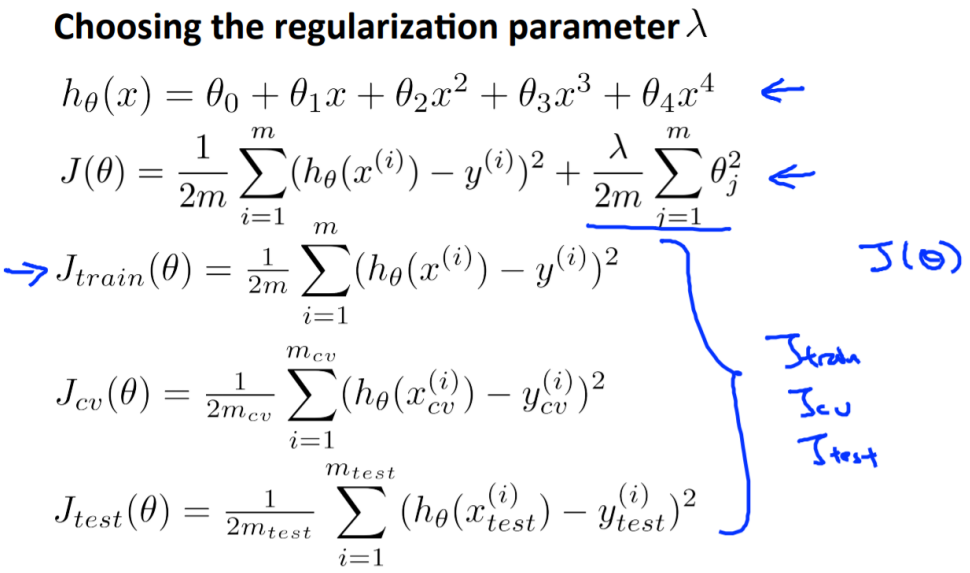


Figure 2.7: Regularization parameter.

2.1.6 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 :

1. As the training set gets larger, the error for a quadratic function increases.
2. 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.

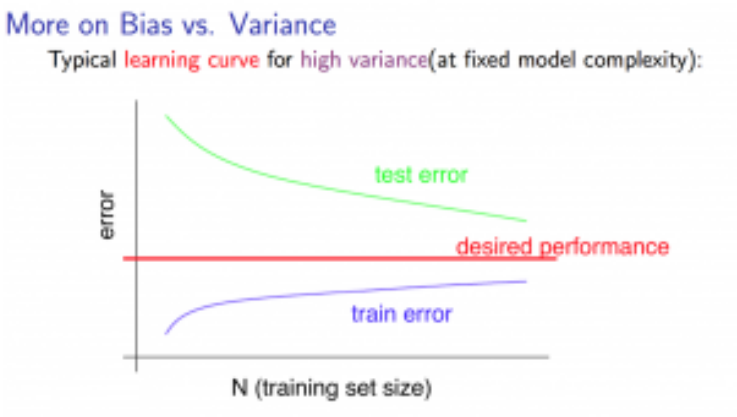


Figure 2.8: High variance.

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

2.1.7 Deciding What to Do Next Revisited

Our decision process can be broken down as follows:

1. **Getting more training examples:** Fixes high variance.
2. **Trying smaller sets of features:** Fixes high variance.
3. **Adding features:** Fixes high bias.
4. **Adding polynomial features:** Fixes high bias.
5. **Decreasing λ:** Fixes high bias.
6. **Increasing λ:** Fixes high variance.

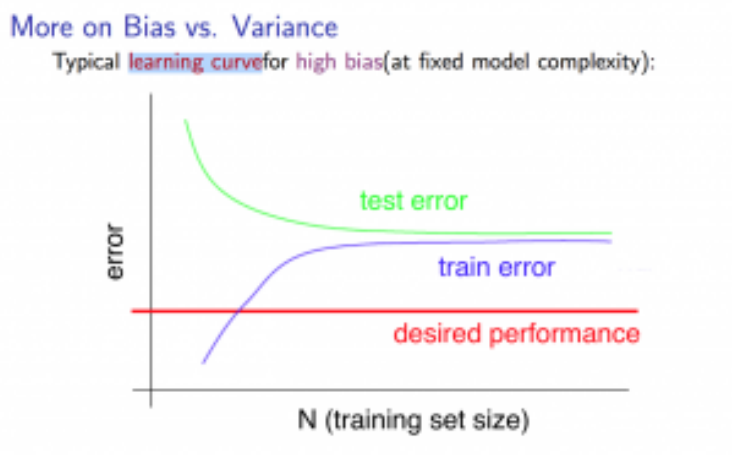


Figure 2.9: High bais.

Diagnosing Neural Networks

1. A neural network with fewer parameters is **prone to underfitting**. It is also **computationally cheaper.**

2. 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.

Machine learning diagnostic

Diagnostic: A test that you can run to gain insight what is/ isn't working with a learning algorithm, and gain guidance as to how best to improve its performance.
Diagnostic can take time to implement, but doing so can be a very good use of your time.

Model Complexity Effects:

1. Lower-order polynomials (low model complexity) have high bias and low variance. In this case, the model fits poorly consistently.

2. 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.

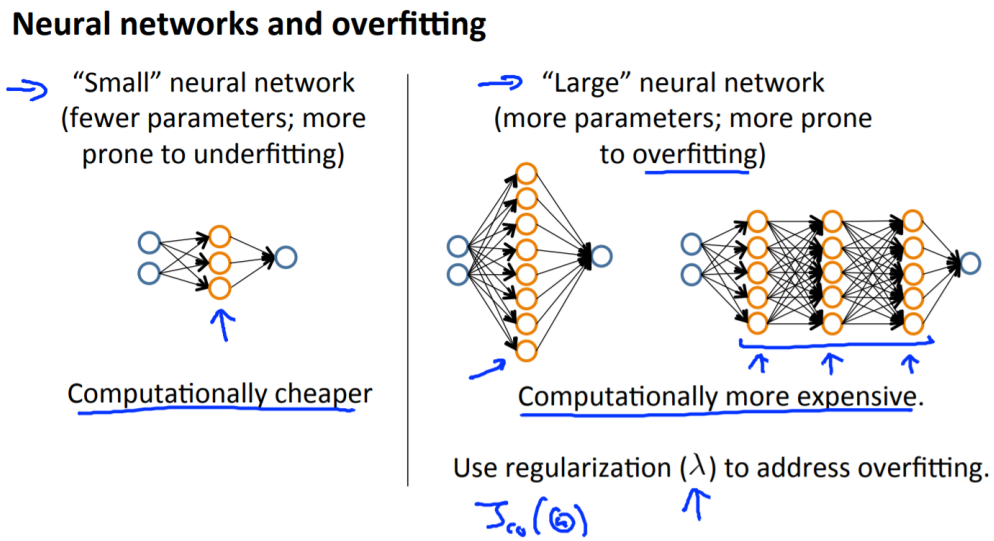


Figure 2.10: Regularization for neural networks with overfitting hypothesis.

3. In reality, we would want to choose a model somewhere in between, that can generalize well but also fits the data reasonably well.

2.1.8 Example on Building a spam classifier

Spam classifier is a problem of classification under Supervised learning.

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.

As we will train our algorithm by giving a large set of data, and it will classify on what terms the emails are being classified into spam and non-spam, by seeing and learning from that our algorithm classifies the given set of emails given by us into spam and non-spam.

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

1. Collect lots of data (for example **"honeypot"** project but doesn't always work).

2. Develop sophisticated features (for example: using email header data in

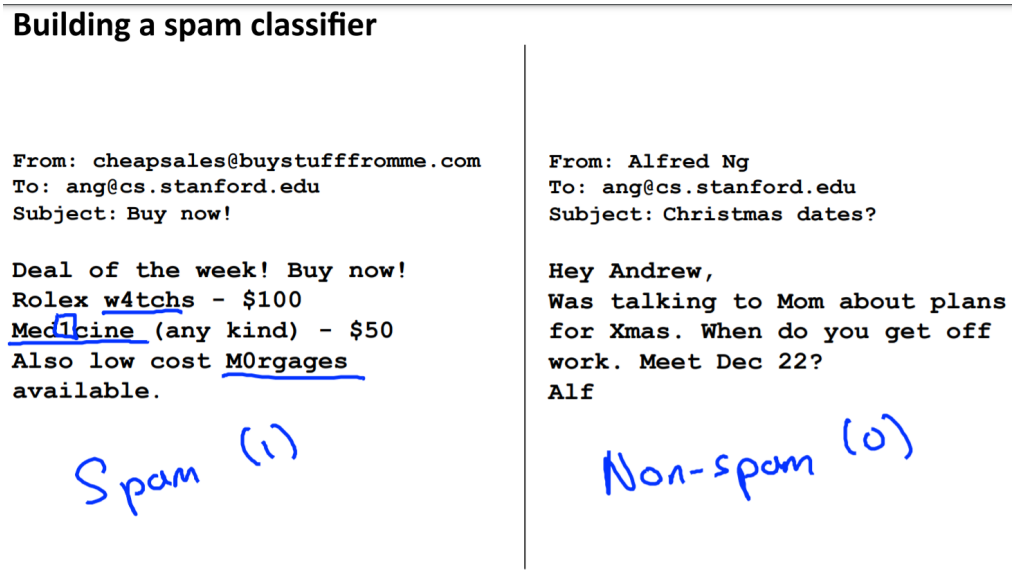


Figure 2.11: Classifier of mails based spam/non-spam..

spam emails).

3. 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.

2.1.9 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.

For example, assume that we have 500 emails and our algorithm misclassifies a 100 of them. We could manually analyze the 100 emails and categorize them based on what type of emails they are. We could then try to come up with new cues and features that would help us classify these 100 emails correctly. Hence, if most of our misclassified emails are those which try to steal passwords, then we could find some features that are particular to those emails and add them to our model. We could also see how classifying each word according to its root

changes our error rate:
We do **error analysis** to know that whether our algorithm is working fine (or) not. As we take certain cases(even though we are taking large set of data, there could be a corner case that our algorithm might not encountered in the given data set) and it is must that we take 20% each of cross-validation set and test set.

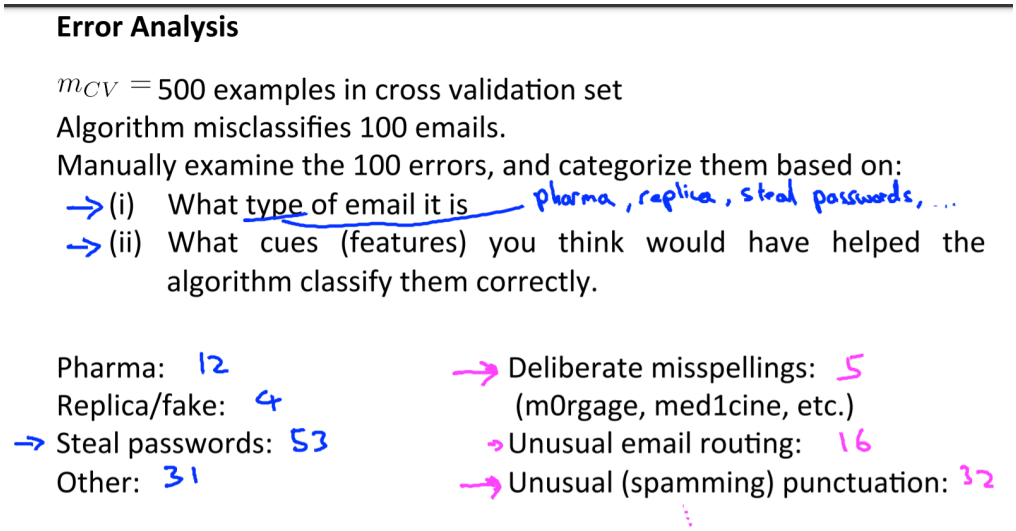


Figure 2.12: Error analysis.

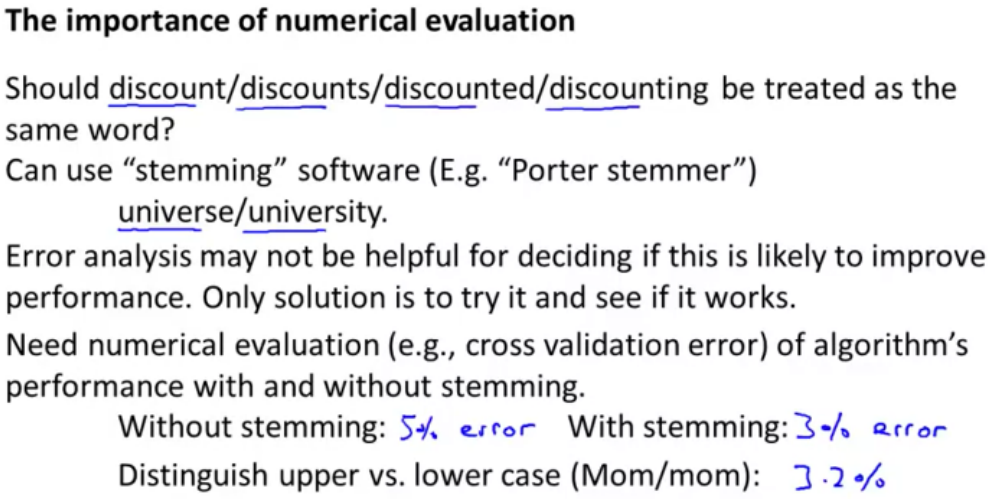


Figure 2.13: Examples of Numerical evaluation.

It is very important to get error results as a single, numerical value. Otherwise it is difficult to assess your algorithm's performance. For example if we use stemming, which is the process of treating the same word with different forms (fail/failing/failed) as one word (fail), and get a 3% error rate instead of 5%, then we should definitely add it to our model. However, if we try to distinguish between upper case and lower case letters and end up getting a 3.2% error rate instead of 3%, then we should avoid using this new feature. Hence, we should try new things, get a numerical value for our error rate, and based on our result decide whether we want to keep the new feature or not.

2.1.10 Error metrics for skewed classes

Skewed classes

Skewed classes : Skewed classes basically refer to a dataset, wherein the number of training example belonging to one class out-numbers heavily the number of training examples beloning to the other.
Consider a binary classification, where a cancerous patient is to be detected based on some features. And say only 1 of the data provided has cancer positive. In a setting where having cancer is labelled 1 and not cancer labelled 0, if a system naively gives the prediction as all 0's, still the prediction accuracy will be 99%.
Therefore, it can be said with conviction that the accuracy metrics or mean-squared error for skewed classes, is not a proper indicator of model performance. Hence, there is a need for a different error metric for skewed classes. Skewed class distributions present a challenge in many different domains. Specifically,

most supervised machine learning algorithms exhibit poor performance when faced with skewed class distributions. This is referred to as the class imbalance problem.
Class imbalance often occurs in real-life datasets involving rare events such as detecting certain medical conditions, fraudulent transactions.
When the majority of data items in your dataset represents items belonging to one class, we say the dataset is skewed or imbalanced. For better understanding, lets consider a binary classification problem, cancer detection. Say we've five thousand instances in our dataset but only five hundred positive instances, i.e., instances were cancer was actually present. Then we've an imbalanced dataset. This happens more often with datasets in real life as the chances of finding cancer among all the checks that happen or a fraudulent transaction among all the transactions that occur daily is comparatively low.

```
In [64]: df.sample(5)
```

Out[64]:

	mean_radius	mean_texture	mean_perimeter	mean_area	mean_smoothness	diagnosis
233	10.94	18.59	70.39	370.0	0.10040	1
35	16.16	21.54	106.20	809.8	0.10080	0
132	16.07	19.65	104.10	817.7	0.09168	0
175	14.78	23.94	97.40	668.3	0.11720	0
190	15.85	23.95	103.70	782.7	0.08401	0

```
In [65]: df.diagnosis.value_counts()
```

```
Out[65]: 0    212
         1     57
         Name: diagnosis, dtype: int64
```

Figure 2.14: Sample output of a Classification problem.

Why does it matter if the dataset is skewed?
When your dataset do not represent all classes of data equally, the model might overfit to the class that's represented more in your dataset and become oblivious to the existence of the minority class. It might even give you a good accuracy but fail miserably in real life. In our example, a model that keeps predicting that there's no cancer every single time will also have a good accuracy as the occurrence of cancer itself will be rare among the inputs. But it will fail when an actual case of cancer is subjected to classification, failing its original purpose.

Different ways to deal with an imbalanced dataset
A widely adopted technique for dealing with highly unbalanced datasets is called resampling. Resampling is done after the data is split into training, test and validation sets. Resampling is done only on the training set or the performance measures could get skewed. Resampling can be of two types:

1. Over-sampling.
2. Under-sampling.

Under sampling involves removing samples from the majority class and over-sampling involves adding more examples from the minority class . The simplest implementation of over-sampling is to duplicate random records from the minority class, which can cause overfitting. In under-sampling, the simplest technique involves removing random records from the majority class, which can cause loss of information.
Another technique similar to upsampling is to create synthetic samples. Adding synthetic samples is also only done after the train-test split, into the training data.

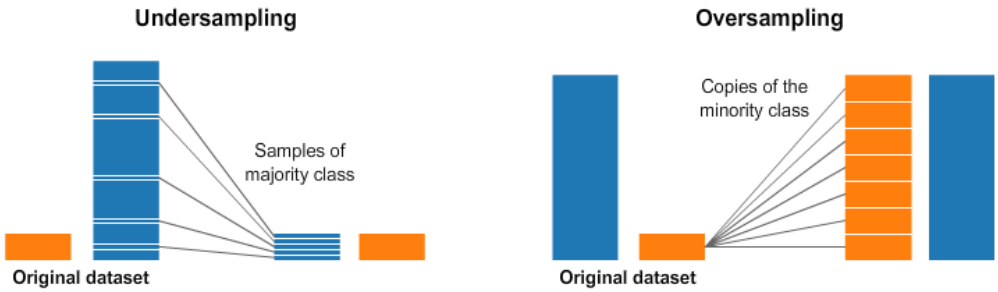


Figure 2.15: Undersampling and Oversampling.

Precision/Recall

Note: y = 1 is the rarer class among the two.
In a binary classification, one of the following four scenarios may occur,

- **True Positive (TP):** the model predicts 1 and the actual class is 1.

- **True Negative (TN):** the model predicts 0 and the actual class is 0.
- **False Positive (FP):** the model predicts 1 but the actual class is 0.
- **False Negative (FN):** the model predicts 0 but the actual class is 1.

Confusion matrix

Confusion matrix is a table that tells us how well our model has performed after it has been trained.
A confusion matrix is a table with two rows and two columns that reports the number of false positives, false negatives, true positives, and true negatives.

		Predicted	
		Negative	Positive
Actual	Negative	a	b
	Positive	c	d

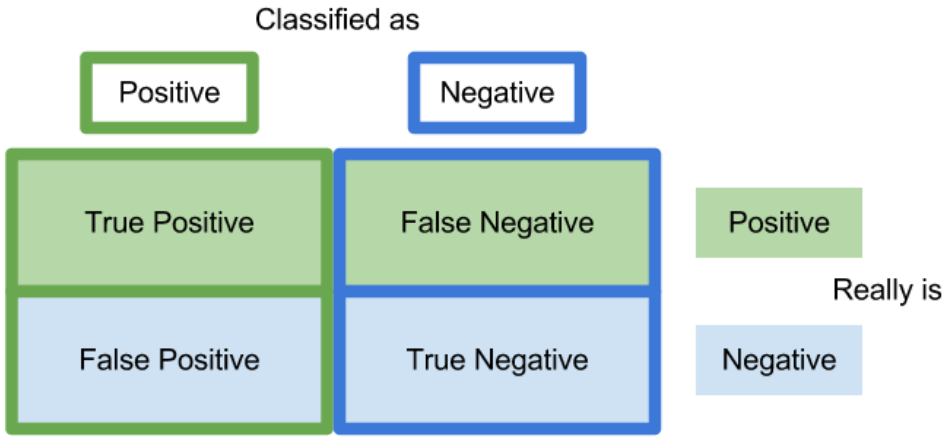


Figure 2.16: Basic binary confusion matrix.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.3}$$

$$\text{Specificity} = \frac{TN}{FP + TN} \tag{2.4}$$

$$\textbf{Sensitivity} = \frac{TN}{TN + FP} \tag{2.5}$$

Then, precision and recall can be defined as :
Precision : Of all patients where we predicted y = 1, what fraction actually has cancer?

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.6}$$

Recall : Of all patients that actually have cancer, what fraction did we correctly detect as having cancer?

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.7}$$

Recall defines of all the actual y = 1, which ones did the model predict correctly.
Now, if we evaluate a scenario where the classifier predicts all 0's then the recall of the model will be 0, which then points out the inability of the system.

2.1.11 Trading of precision and recall

By changing the threshold value for the classifier confidence, one can adjust the precision and recall for the model.
For example, in a logistic regression the threshold is generally at 0.5. If one increases it, we can be sure that of all the predictions made more will be correct, hence, high precision. But there are also higher chances of missing the positive cases, hence, the lower recall.
Similarly, if one decreases the threshold, then the chances of false positives increases, hence low precision. Also, there is lesser probability of missing the actual cases, hence high recall.
A precision-recall trade - off curve may look like one among the following,

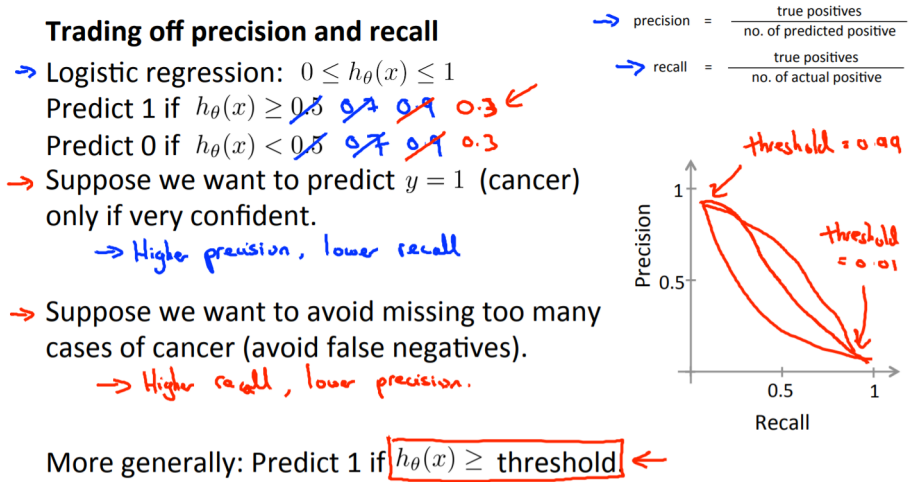


Figure 2.17: Trading of precision and recall.

2.1.12 F Score

Given two pairs of precision and recall, how to choose the better pair. One of the options would be to choose the one which higher average. That is not the ideal solution as the pair with (precision=0.02 and recall=1) has a better mean than the pair (precision= 0.5 and recall=0.4). Enter F Score or F_1 Score, which is the harmonic mean of precision and recall, defined as,

$$F_1 = 2\frac{P \times R}{P + R} \tag{2.8}$$

The above formula has advantage over the average method because, if either precision or recall is small, the the numerator product PR will weigh the F-Score low and consequently lead to choosing the better pair of precision and recall. So,

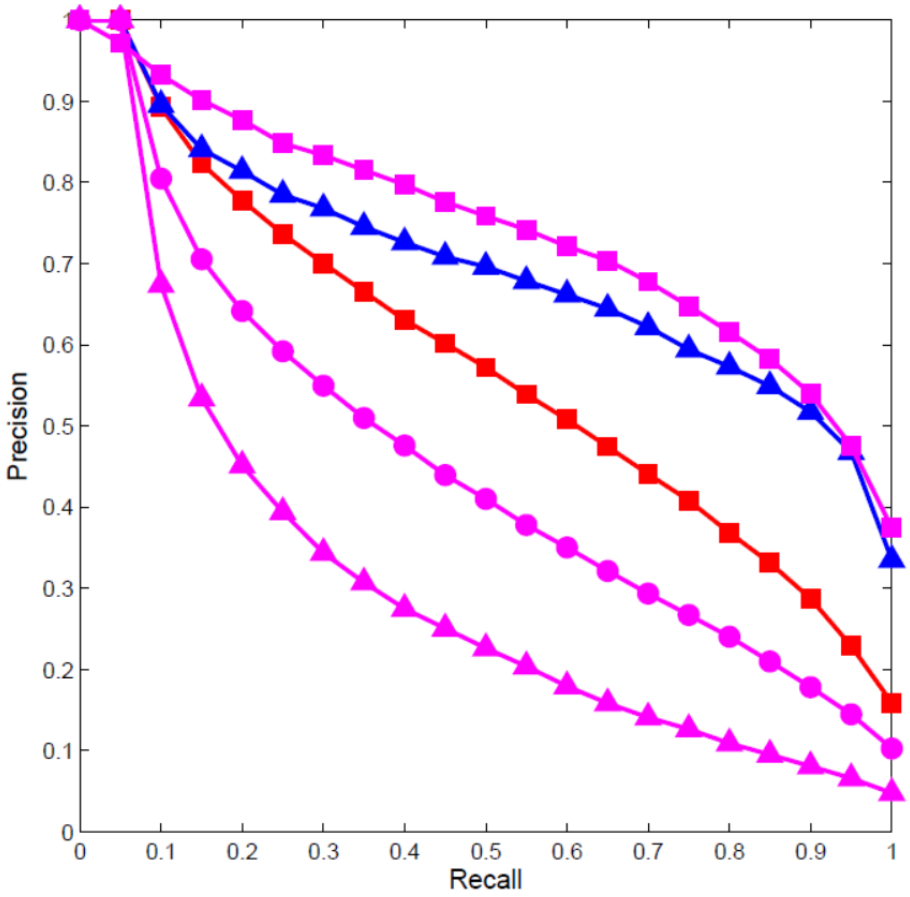


Figure 2.18: Precision/Recall for different values of threshold hypothesis.

1. If P=0 (or) R=0, then F_1=0.
2. If P=1 and R=1, then F_1=1.

Note: One reasonable way of automatically choosing threshold for classifier is to try a range of them on the cross-validation set and pick the one that gives the highest F-Score.

Sensitivity/Specifivity

Apart from precision and recall, sensitivity and specifivity are among the most used error metrics in classfication.

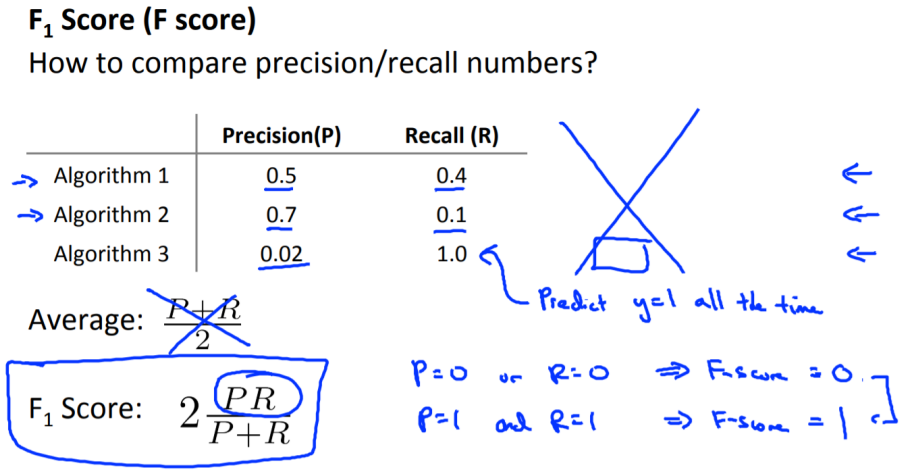


Figure 2.19: Calculation of F score through precision/recall.

1. **Sensitivity or True Positive Rate (TPR)** is another name for recall and is also called **hit rate.**
2. **Specificity (SPC) or True Negative Rate.**

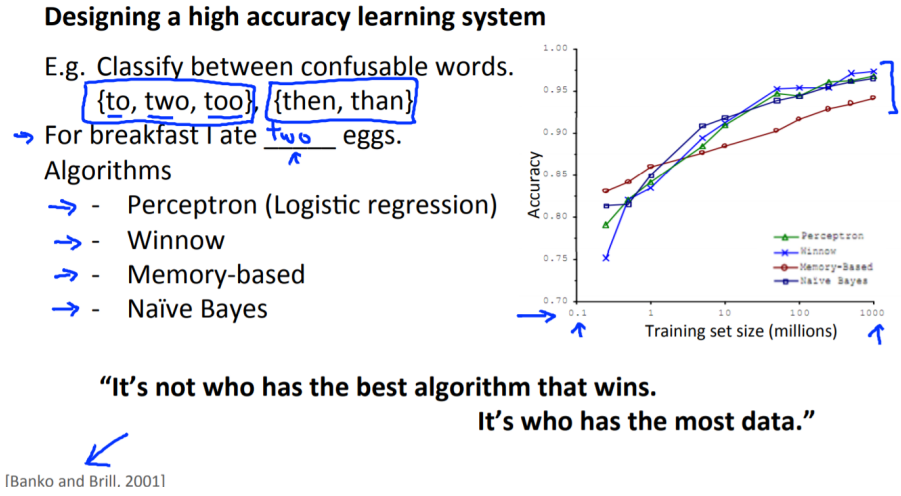


Figure 2.20: Designing a high accuracy learning system.

Using Large Datasets

For a high bias problem in the model, gathering more and more data will not help the model improve.
But under certain conditions, getting a lot of data and training on a certain type of training algorithm can be an effective way to improve the learning algorithm's performance.
The following are the conditions that should be met for the above statement to hold true,

1. The features, x, must have sufficient information to predict y accurately. One way to test this would be to check if human expert can make a confident predition using the features.

2. Using a learning algorithm with a large number of parameters to learn (e.g. logistic regression, linear regression, neural network with many hidden units etc.). What this truly accomplishes is that these algorithms are **low bias algorithm** due to the large number of learnable parameters.

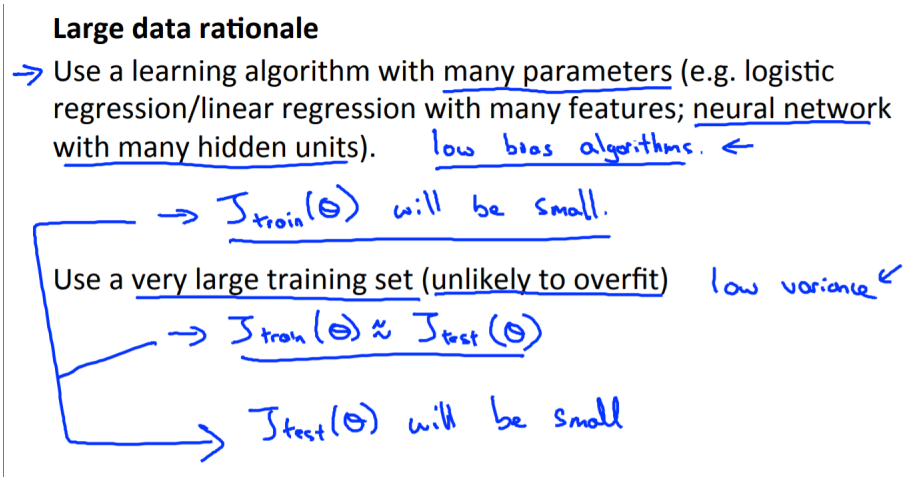


Figure 2.21: Data Rationale

In such settings, where the problem of high bias is removed by the virtue of highly parametrized learning algorithms, a large dataset ensures the **low variance.** Hence, under the listed settings a large number of dataset is almost always going to help improve the model performance.

Chapter 3

Support Vector Machine (SVM)

A support vector machine (SVM) is a supervised machine learning model that uses classification algorithms for two-group classification problems. After giving an SVM model sets of labeled training data for each category, they're able to categorize new text.

Compared to newer algorithms like neural networks, they have too main advantages: higher speed and better performance with a limited number of samples (in the thousands). This makes the algorithm very suitable for text classification problems,where it's common to have access to a dataset of at most a couple of thousands of tagged samples. A SVM is a discriminative classifier formally defined by a separating hyperplane. Given labeled training data, the algorithm outputs an optimal hyperplane which categorizes new examples.

3.0.1 How does SVM work?

The linear SVM classifier works by drawing a straight line between two classes. All the data points that fall on one side of the line will be labeled as one class and all the points that fall on the other side will be labeled as the second. Sounds simple enough, but there's an infinite amount of lines to choose from. How do we know which line will do the best job of classifying the data? This is where the LSVM algorithm comes in to play. The LSVM algorithm will select a line that not only separates the two classes but stays as far away from the closest samples as possible. In fact, the "support vector" in "support vector machine" refers to two position vectors drawn from the origin to the points which dictate the decision boundary.

The basics of support vector machine and how it works are best understood with a simple example. Let's imagine we have two tags: red and blue, and our data has two features: x and y. We want a classifier that, given a pair of (x,y) coordinates, outputs if it's either red or blue. We plot our already labeled training data on a plane:

A support vector machine takes these data points and outputs the hyperplane

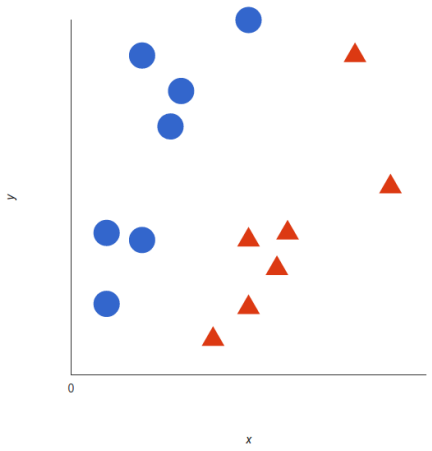


Figure 3.1: Our labeled data

(which in two dimensions it's simply a line) that best separates the tags. This line is the **decision boundary**: anything that falls to one side of it we will classify as blue, and anything that falls to the other as red.

But, what exactly is the best hyperplane? For SVM, it's the one that maximizes the margins from both tags. In other words: the hyperplane (remember it's a line in this case) whose distance to the nearest element of each tag is the largest. **According to the SVM algorithm we find the points closest to the line from both the classes.These points are called support vectors. Now, we compute the distance between the line and the support vectors. This distance is called the margin. Our goal is to maximize the margin. The hyperplane for which the margin is maximum is the optimal hyperplane.**

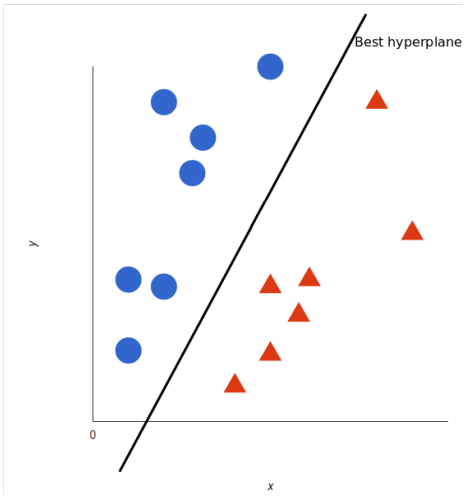


Figure 3.2: The best hyperplane is simply a line

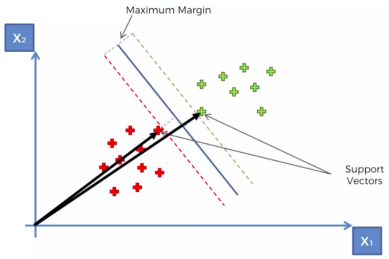


Figure 3.3: Not all hyperplanes are created equal

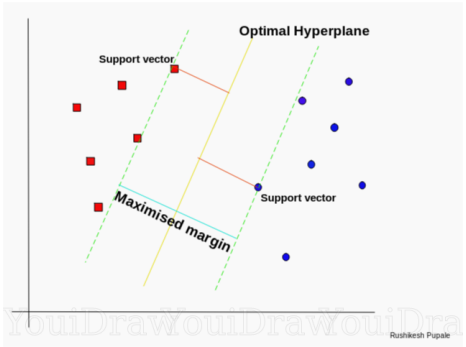
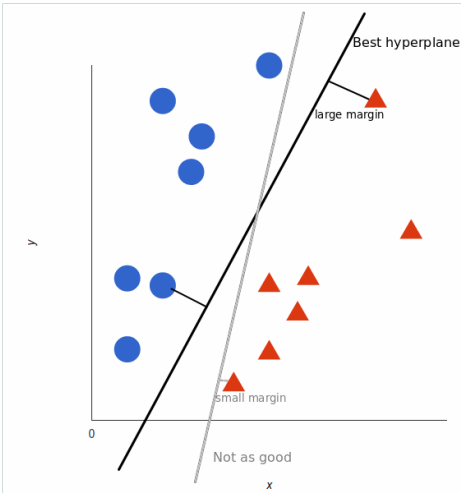


Figure 3.4: More information regarding SVM and hyperplane

It is a very powerful classification algorithm to maximize the margin among class variables. This margin (support vector) represents



the distance between the separating hyperplanes (decision boundary). The reason to have decision boundaries with large margin is to separate positive and negative hyperplanes with adjustable bias-variance proportion. The goal is to separate so that negative samples would fall under negative hyperplane and positive samples would fall under positive hyperplane. SVM is not as prone to outliers as it only cares about the points closest to the decision boundary. It changes its decision boundary depending on the placement of the new positive or negative events.
The decision boundary is much more important for Linear SVM's – the whole goal is to place a linear boundary in a smart way. There isn't a probabilistic interpretation of individual classifications, at least not in the original formulation.
Hence, key points are:

1. SVM try to maximize the margin between the closest support vectors whereas logistic regression maximize the posterior class probability.

2. SVM is deterministic (but we can use Platts model for probability score) while LR is probabilistic.

3. For the kernel space, SVM is faster.

Logistic Regression	Support Vector Machine
1. It is an algorithm used for solving classification problems.	1. It is a model used for both classification and linear regression.
2. It is not used to find the best margin, instead, it can have different decision boundaries with different weights that are near the optimal point.	2. It tries to find the "best" margin (distance between the line and the support vectors) that separates the classes and thus reduces the risk of error on the data.
3. It works with already identified identified independent variable.	3. It works well with unstructured and semi-structured data like text and images.
4. It is based on statistical approach.	4. It is based on geometrical properties of the data.
5. It is vulnerable to overfitting.	5. The risk of overfitting is less in SVM.

3.0.2 SVM for Non-linear data

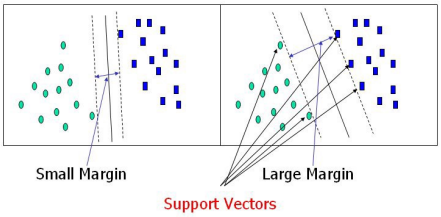


Figure 3.5: Difference between small and large margin.

It's pretty clear that there's not a linear decision boundary (a single straight line that separates both tags). However, the vectors are very clearly segregated and it looks as though it should be easy to separate them.
This data is clearly not linearly separable. We cannot draw a straight line that can classify this data. But, this data can be converted to linearly separable data in higher dimension. Lets add one more dimension and call it z-axis. Let the co-ordinates on z-axis be governed by the constraint,
So here's what we'll do: we will add a third dimension. Up until now we had two dimensions: x and y. We create a new z dimension, and we rule that it be calculated a certain way that is convenient for us: z = $x^2 + y^2$ (you'll notice that's the equation for a circle).
This will give us a three-dimensional space. Taking a slice of that space, it looks like this:
Now, the SVM produces best hyperplane to the linearly separated data,

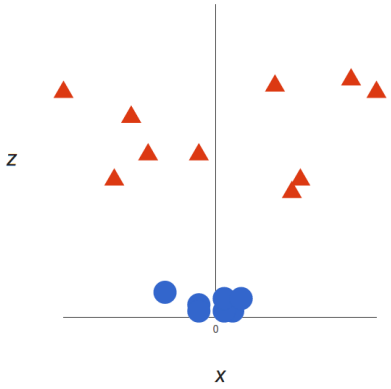


Figure 3.6: From a different perspective, the data is now in two linearly separated groups.

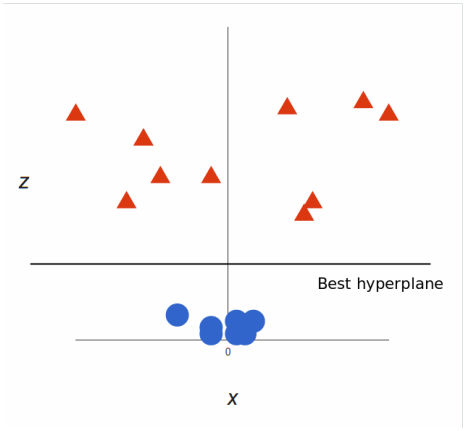


Figure 3.7: Best hyperplane is produced.

Now the data is clearly linearly separable. Let the purple line separating the data in higher dimension be z=k, where k is a constant. Since, $z = x^2 + y^2$ we get $x^2 + y^2 = k$; which is an equation of a circle. So, we can project this linear separator in higher dimension back in original dimensions using this transformation.

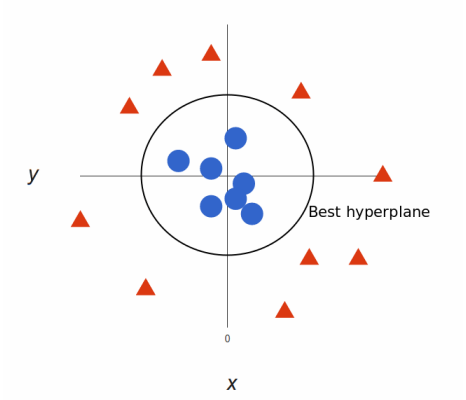


Figure 3.8: Decision boundary in original dimensions.

And there we go! Our decision boundary is a circumference of radius 1, which separates both tags using SVM.
Thus we can classify data by adding an extra dimension to it so that it becomes linearly separable and then projecting the decision boundary back to original dimensions using mathematical transformation. But finding the correct transformation for any given dataset isn't that easy. Thankfully, we can use kernels

in sklearn's SVM implementation to do this job.

3.0.3 Hyperplane

Now that we understand the SVM logic lets formally define the hyperplane.

Note: A hyperplane in an n-dimensional Euclidean space is a flat, n-1 dimensional subset of that space that divides the space into two disconnected parts.

For example let's assume a line to be our one dimensional Euclidean space(i.e. let's say our datasets lie on a line). Now pick a point on the line, this point divides the line into two parts. The line has 1 dimension, while the point has 0 dimensions. So a point is a hyperplane of the line.
For two dimensions we saw that the separating line was the hyperplane. Similarly, for three dimensions a plane with two dimensions divides the 3d space into two parts and thus act as a hyperplane. Thus for a space of n dimensions we have a hyperplane of n-1 dimensions separating it into two parts.

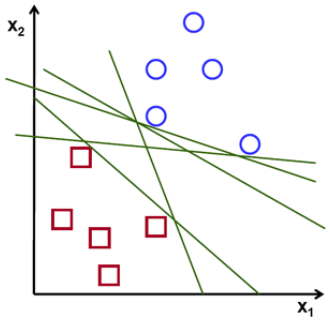


Figure 3.9: Possible hyperplanes.

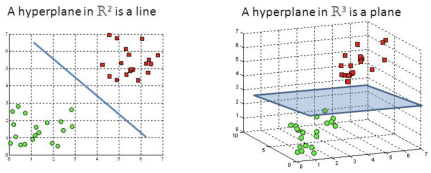


Figure 3.10: Hyperplanes in 2D and 3D feature space.

3.0.4 Tuning parameters

Parameters are arguments that you pass when you create your classifier. Following are the important parameters for SVM-

1. **C($\frac{1}{\lambda}$):**
   It controls the trade off between smooth decision boundary and classifying training points correctly. A large value of c means you will get more training points correctly.

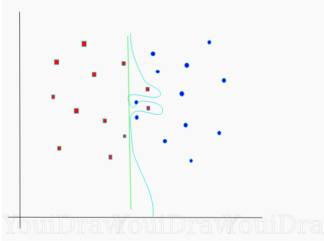


Figure 3.11: Smooth decision boundary vs classifying all points correctly.

   Consider an example as shown in the figure 3.17. There are a number of decision boundaries that we can draw for this dataset. Consider a straight (green colored) decision boundary which is quite simple but it comes at the cost of a few points being misclassified. These misclassified points are called outliers. We can also make something that is considerably more wiggly(sky blue colored decision boundary) but where we get potentially all of the training points correct. Of course the trade off having something that is very intricate, very complicated like this is that chances are it is not going to generalize quite as well to our test set. So something that is simple, more straight maybe actually the better choice if you look at the accuracy. Large value of c means you will get more intricate decision curves trying to fit in all the points. Figuring out how much you want to have a smooth decision boundary vs one that gets things correct is part of artistry of machine learning. So try different values of c for your dataset to get the perfectly balanced curve and avoid over fitting.

2. **Gamma:**
   It defines how far the influence of a single training example reaches. If it has a low value it means that every point has a far reach and conversely high value of gamma means that every point has close reach.
   If gamma has a very high value, then the decision boundary is just going to be dependent upon the points that are very close to the line which effectively results in ignoring some of the points that are very far from the decision boundary. This is because the closer points get more weight and it results in a wiggly curve as shown in previous graph.On the other

hand, if the gamma value is low even the far away points get considerable weight and we get a more linear curve.

3.0.5 Algorithm

Suppose, we had a vector w which is always normal to the hyperplane (perpendicular to the line in 2 dimensions). We can determine how far away a sample is from our decision boundary by projecting the position vector of the sample on to the vector w. As a quick refresher, the dot product of two vectors is proportional to the projection of the first vector on to the second.

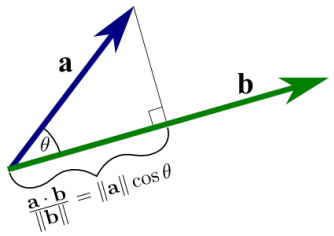


Figure 3.12: Projection vector.

If it's a positive sample, we're going to insist that the proceeding decision function (the dot product of w and the position vector of a given sample plus some constant) returns a value greater than or equal to 1.

$$\vec{w}.\vec{x}_+ + b \geq 1. \tag{3.1}$$

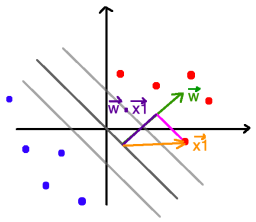


Figure 3.13: Dot product representation for '+' samples.

Similarly, if it's a negative sample, we're going to insist that the proceeding decision function returns a value smaller than or equal to -1.

$$\vec{w}.\vec{x}_- + b \leq -1. \tag{3.2}$$

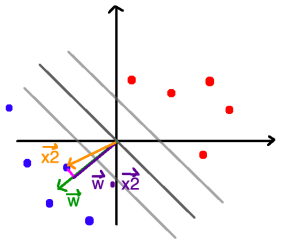
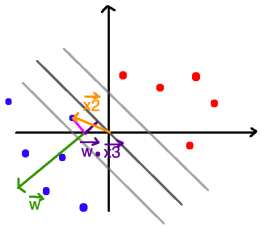


Figure 3.14: Dot product representation for '-' samples.

In other words, we won't consider any samples located between the decision boundary and support vectors.

we introduce an additional variable stickily for convenience. The variable y



will be equal to positive one for all positive samples and negative one for all negative samples.

$$y_i \begin{cases} +1 \; for \; + \; samples. \\ -1 \; for \; - \; samples. \end{cases}$$

After multiplying by y, the equations for the positive and negative samples are equal to one another.

$$y_i(\vec{w}.\vec{x}_+ + b) \geq 1.$$
$$y_i(\vec{w}.\vec{x}_- + b) \leq 1.$$

Meaning, we can simplify the constraints down to a single equation.

$$y_i(\vec{w}.\vec{x}_i + b) - 1 = 0.$$

Next, we need to address the process by which we go about maximizing the margin. To get an equation for the width of the margin, we subtract the first support vector from the one below it and the multiply the result by the unit vector of w which is always perpendicular to the decision boundary.

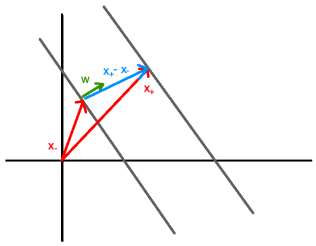


Figure 3.15: Width of the lines produced by samples.

$$width = (\vec{x_+} - \vec{x_-}).\frac{\vec{w}}{\|w\|}$$

Using the constraints from above and a bit of algebra, we get the following.

$$\begin{aligned}
y_i(\vec{w}.\vec{x_i} + b) - 1 = 0.& \\
(1)(\vec{w}.\vec{x}_+ + b) - 1 = 0.& \\
\vec{w}.\vec{x}_+ = 1 - b & \quad (1) \\
(-1)(\vec{w}.\vec{x}_- + b) - 1 = 0.& \\
\vec{w}.\vec{x}_- = -1 - b & \quad (2) \\
width = ((\vec{x_+}.\vec{w}) - (\vec{x_-}.\vec{w}))\frac{1}{\|w\|}& \\
width = ((1 - b) - (-1 - b))\frac{1}{\|w\|}& \\
width = \frac{2}{\|w\|}&
\end{aligned}$$

Therefore, in order to select the optimal decision boundary, we must maximize the equation we just computed. We apply a few more tricks before proceeding. (refer to the *MITLecture*).

$$\max \frac{2}{\|w\|}$$

which is proportional to:

$$\max \frac{1}{\|w\|}$$

which is equivalent to:

$$\min \|w\|$$

we're going to cheat a little and use:

$$\min \frac{1}{2}\|w\|^2 \qquad \textbf{since} \qquad \frac{d1}{dx}\frac{1}{2}x^2 = x$$

Now, in most machine learning algorithms, we'd use something like gradient descent to minimize said function, however, for support vector machines, we use the Lagrangian.In essence, using Lagrangian, we can solve for the global minimum like we'd do in high school level calculus (i.e. take the derivative of the function and make it equal to zero). The Lagrange tells us to subtract the cost function by the summation over all the constraints where each of those constraints will be multiplied by some constant alpha (normally written as lambda for the Lagrangian).

$$L = \frac{1}{2}\|w\|^2 - \sum_i^n \alpha_i [y_i(\vec{w}.\vec{x} + b) - 1]$$

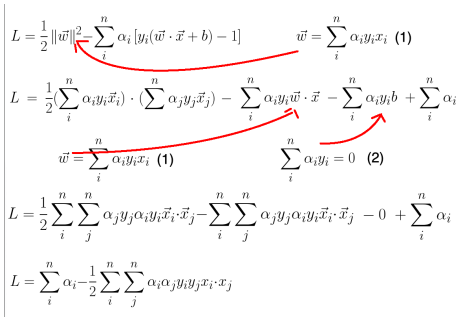
$$\frac{\partial L}{\partial w} = \vec{w} - \sum_i^n \alpha_i y_i x_i = 0$$

$$\boxed{\vec{w} = \sum_i^n \alpha_i y_i x_i}$$

$$\frac{\partial L}{\partial b} = - \sum_i^n \alpha_i y_i = 0$$

$$\boxed{\sum_i^n \alpha_i y_i = 0}$$

Then, we perform some more algebra, plugging the equations we found in the previous step back into the original equation.
Before we can proceed any further, we need to express the equation in terms of matrices instead of summations. The reason being, the $\boxed{qp}$ function from the CVXOPT library, which we'll use to solve the Lagrangian, accepts very specific arguments. Thus, we need to go from:
As this is still beyond your capability. Let us stop it here. But, I will



provide link for futher understandings.

Machine Learning with python.

3.0.6 The Kernel

In our example we found a way to classify nonlinear data by cleverly mapping our space to a higher dimension. However, it turns out that calculating this transformation can get pretty computationally expensive: there can be a lot of new dimensions, each one of them possibly involving a complicated calculation. Doing this for every vector in the dataset can be a lot of work, so it'd be great if we could find a cheaper solution.
And we're in luck! Here's a trick: SVM doesn't need the actual vectors to work its magic, it actually can get by only with the dot products between them. This means that we can sidestep the expensive calculations of the new dimensions. This is what we do instead:

1. Imagine the new space we want:
   $z = x^2 + y^2$.

2. Figure out what the dot product in that space looks like:
   a·b = xa·xb + ya·yb + za·zb
   a·b = xa·xb + ya·yb + (xa^2 + ya^2)·(xb^2 + yb^2)

3. Tell SVM to do its thing, but using the new dot product — we call this a **kernel function.**

That's it! That's the kernel trick, which allows us to sidestep a lot of expensive calculations. Normally, the kernel is linear, and we get a linear classifier. However, by using a nonlinear kernel (like above) we can get a nonlinear classifier without transforming the data at all: we only change the dot product to that of the space that we want and SVM will happily chug along.
Note that the kernel trick isn't actually part of SVM. It can be used with other linear classifiers such as logistic regression. A support vector machine only takes care of finding the decision boundary.

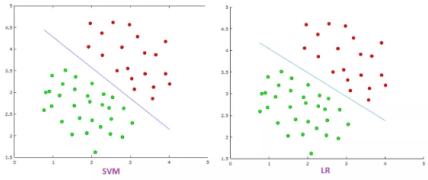
3.0.7 Differences between SVM and Logistic Regression

Logistic Regression: Logistic regression is an algorithm that is used in solving classification problems. It is a predictive analysis that describes data and explains the relationship between variables. Logistic regression is applied to an input variable (X) where the output variable (y) is a discrete value which ranges between 1 (yes) and 0 (no).
Support Vector Machine: The support vector machine is a model used for both classification and regression problems though it is mostly used to solve classification problems. The algorithm creates a hyperplane or line(decision boundary) which separates data into classes. It uses the kernel trick to find the best line separator (decision boundary that has same distance from the boundary point of both classes). It is a clear and more powerful way of learning complex non linear functions.

Problems that can solved using Logistic Regression	Problems that can solved using SVM
1.Cancer detection — can be used to detect if a patient have cancer (1) or not(0).	1.Image classification.
2.Test score — predict if a student passed(1) or failed(0) a test.	2. Recognizing handwriting.
3.Marketing — predict if a customer will purchase a product(1) or not(0).	3. Cancer Detection

Differences

1. SVM tries to finds the "best" margin (distance between the line and the support vectors) that separates the classes and this reduces the risk of error on the data, while logistic regression does not, instead it can have different decision boundaries with different weights that are near the optimal point.



2. SVM works well with unstructured and semi-structured data like text and images while logistic regression works with already identified independent variables.

3. SVM is based on geometrical properties of the data while logistic regression is based on statistical approaches.

4. The risk of overfitting is less in SVM, while Logistic regression is vulnerable to overfitting.

When To Use Logistic Regression vs Support Vector Machine

Depending on the number of training sets (data)/features that you have, you can choose to use either logistic regression or support vector machine.
Lets take these as an example where :
n = number of features,
m = number of training examples.

1. If n is large (1–10,000) and m is small (10–1000) : use logistic regression or SVM with a linear kernel.
2. If n is small (1–10 00) and m is intermediate (10–10,000) : use SVM with (Gaussian, polynomial etc) kernel.
3. If n is small (1–10 00), m is large (50,000–1,000,000+): first, manually add more features and then use logistic regression or SVM with a linear kernel.

Generally, it is usually advisable to first try to use logistic regression to see how the model does, if it fails then you can try using SVM without a kernel (is otherwise known as SVM with a linear kernel). Logistic regression and SVM with a linear kernel have similar performance but depending on your features, one may be more efficient than the other.

Note: Logistic regression and SVM are great tools for training classification and regression problems. It is good to know when to use either of them so as to save computational cost and time.

Note: The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space(N — the number of features) that distinctly classifies the data points.To separate the two classes of data points, there are many possible hyperplanes that could be chosen. Our objective is to find a plane that has the maximum margin, i.e the maximum distance between data points of both classes. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more confidence.Hyperplanes are decision boundaries that help classify the data points.

Data points falling on either side of the hyperplane can be attributed to different classes. Also, the dimension of the hyperplane depends upon the number of features. If the number of input features is 2, then the hyperplane is just a line. If the number of input features is 3, then the hyperplane becomes a two-dimensional plane. It becomes difficult to imagine when the number of features exceeds 3.Support vectors are data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. Using these support vectors, we maximize the margin of the classifier. Deleting the support vectors will change the position of the hyperplane. These are the points that help us build our SVM.

3.0.8 Optimization Objective

The support vector machine objective can seen as a modification to the cost of logistic regression. Consider the sigmoid function, given as,

$$h_\theta(x) = \frac{1}{1+e^{-z}} \tag{3.3}$$

where $z = \theta^T x$.
The cost function of logistic regression as in the post Logistic Regression Model, is given by,

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\Big(y^{(i)}\, log(h_\theta(x^{(i)}) + (1-y^{(i)})\, log(1-h_\theta(x^{(i)}))\Big) \tag{3.4}$$

$$= -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\, log(\frac{1}{1+e^{-\theta^T x}}) + (1-y^{(i)})\, log(1-\frac{1}{1+e^{-\theta^T x}})\right) \tag{3.5}$$

Each training instance contributes to the cost function the following term,

$$-y\, log(\frac{1}{1+e^{-z}}) - (1-y)\, log(1-\frac{1}{1+e^{-z}})$$

So when y=1, the contributed term is $-log(\frac{1}{1+e^{-z}})$, which can be seen in the plot below. The cost function of SVM, denoted as $cost_1(z)$, is a modification the former and a close approximation.

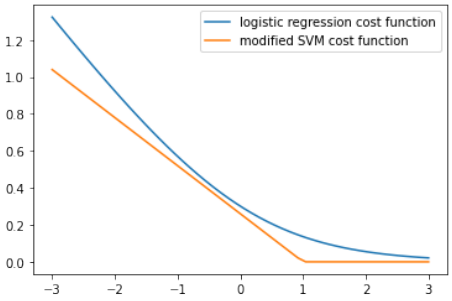


Figure 3.16: SVM Cost function at y = 1.

Similarly, when y=0, the contributed term is $-log(1 - \frac{1}{1+e^{-z}})$, which can be seen in the plot below. The cost function of SVM, denoted as $cost_0(z)$, is a modification the former and a close approximation.

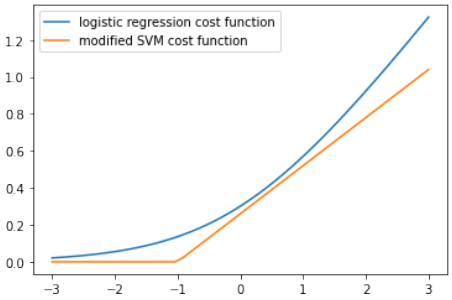


Figure 3.17: SVM Cost function at y = 0.

Note: While the slope the straight line is not of as much importance, it is the linear approximation that gives SVMs computational advantages that helps in formulating an easier optimization problem.

Regularized version of eq:3.5 can from the post Regularized Logistic Regression can rewritten as,

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\left(-log(h_\theta(x^{(i)}))\right) + (1-y^{(i)})\left(-log(1-h_\theta(x^{(i)}))\right)\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{3.6}$$

In order to come up with the cost function for the SVM, eq:3.6 is modified by replacing the corresponding cost terms, which gives,

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\Big(y^{(i)}\, cost_1(z) + (1-y^{(i)})\, cost_0(z)\Big) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{3.7}$$

Following the conventions of SVM the following modifications are made to the cost in eq:3.7, which effectively is a change in notation but not the underlying logic,

1. Removing $\frac{1}{m}$ does not affect the minimization logic at all as the minima of a function is not changed by the linear scaling.
2. Change the form of parameterization from $A+$ to CA+B where it can be intuitively thought that $C = \frac{1}{\lambda}$.

After applying the above changes, eq:3.7 gives,

$$J(\theta) = C\sum_{i=1}^{m}\Big[y^{(i)}\, cost_1(\theta^T x^{(i)}) + (1-y^{(i)})\, cost_0(\theta^T x^{(i)})\Big] + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2. \tag{3.8}$$

The SVM hypothesis does not predict probability, instead gives hard class labels,

$$h_\theta(x) = \begin{cases} 1, \text{ if } \theta^T x \geq 0 \\ 0, \text{ otherwise} \end{cases}$$

3.0.9 Large Margin 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.

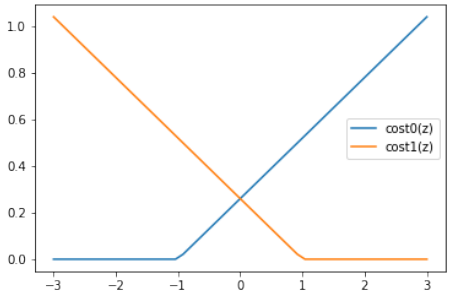


Figure 3.18: SVM Cost function plots.

According to 3.8 and the plots of the cost function as shown in the image above, the following are two desirable states for SVM,

1. If y=1, then $\Theta^\top x \geq 1$ (not just ≥ 0).
2. If y=0, then $\Theta^\top x \geq -1$ (not just ¡0).

Let C in eq:3.8 be a large value. Consequently, in order to minimize the cost, the corresponding term $\sum_{i=1}^{m} \left[y^{(i)}\, cost_1(\theta^T x^{(i)}) + (1 - y^{(i)})\, cost_0(\theta^T x^{(i)})\right]$ must be close to 0.
Hence, in order to minimize the cost function, when y=1, $cost_1(\theta^T x)$ should be 0, and similarly, when y=0, $cost_0(\theta^T x)$ should be 0. And thus, from the plots in fig:3.19, it is clear that it can only fulfilled by the two states listed above.
Following the above intuition, the cost function can we written as,

$$min_\theta J(\theta) = min_\theta \frac{1}{2} \sum_{j=1}^{n} \theta_j^2 \tag{3.9}$$

subject to contraints,

$$\theta^\top x^{(i)} \geq 1, \text{ if } y^{(i)} = 1$$
$$\theta^\top x^{(i)} \leq -1, \text{ if } y^{(i)} = 0$$

What this basically leads to is the selection of a decision boundary that tries to maximize the margin from the support vectors as shown in the plot below. This maximization of the margin as seen for decision boundary A increases the robustness over decision boundaries with lesser margins like B. And it is this property of the SVMs that attributes the name **large margin classifier** to it.

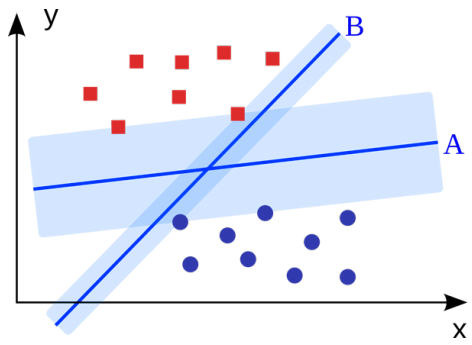


Figure 3.19: Large Margin Decision Boundary.