Stanford University

Andrew Ng

Summary

Machine Learning

---

by Amber Brands

January 22, 2018

[1] https://www.coursera.org/learn/machine-learning/home/welcome

Contents

1 Week 1

1.1 Introduction

1.1.1 Introduction: Machine Learning

There exists no well excepted definition of machine learning. A few examples of a definition are:

- Arthur Samuel (1959): "The field of study that gives computers the ability to learn without being explicitly programmed."
- Tom Mitchell (1998): "Well-posed learning problem: A computer program is said to *learn* from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."
  - Example playing checkers: E = the experience of playing many games of checkers. T = the task of playing checkers. P = the probability that the program will win the next game.

Machine learning algorithms (see Figure 1):

- Supervised learning.
- Unsupervised learning.
- Others: Reinforcement learning, recommender systems and practical advice for applying learning algorithms.

1.1.2 Introduction: Supervised learning

In supervised learning, we are given a data set and already know what our correct output should look like - in other words, assumed is that there is a relationship between the input and the output. Supervised refers to the fact that the algorithm was given **a dataset of right answers**. Supervised learning problems are categorized into "regression" and "classification" problems. In a regression problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function. In a classification problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories. Examples of supervised learning are:

- **Regression problem** Relation between housing prices and size (continues valued output).
- **Classification problem** Relation between breast cancer malignancy and tumour size (discrete valued output of 0 or 1).

1.1.3 Introduction: Unsupervised learning

Unsupervised learning allows us to approach problems with little or no idea what our results should look like. We can derive this structure by clustering the data based on relationships among the variables in the data. With unsupervised learning there is no feedback based on the prediction results. No answers given prior to learning. The algorithm has to find a structure in the given data. Different types of unsupervised learning:

- **Clustered algorithms** Take a collection of 1,000,000 different genes, and find a way to automatically group these genes into groups that are somehow similar or related by different variables, such as lifespan, location, roles, and so on. Other examples are organize computing clusters, social network analysis, market segmentation (grouping of customers), astronomical data analysis.
- **Non-clustered algorithms** The "Cocktail Party Algorithm", allows you to find structure in a chaotic environment (i.e. identifying individual voices and music from a mesh of sounds at a cocktail party). An unsupervised algorithm separates the voices based on the distance/loudness of different sounds.
  - one line of code can separate these sounds: [W,s,v] = svd((repmat(sum(x.*x,1),size(x,1),1).*x)*x')
  - svd (singular value decomposition) function: returns the singular values of matrix A in descending order.

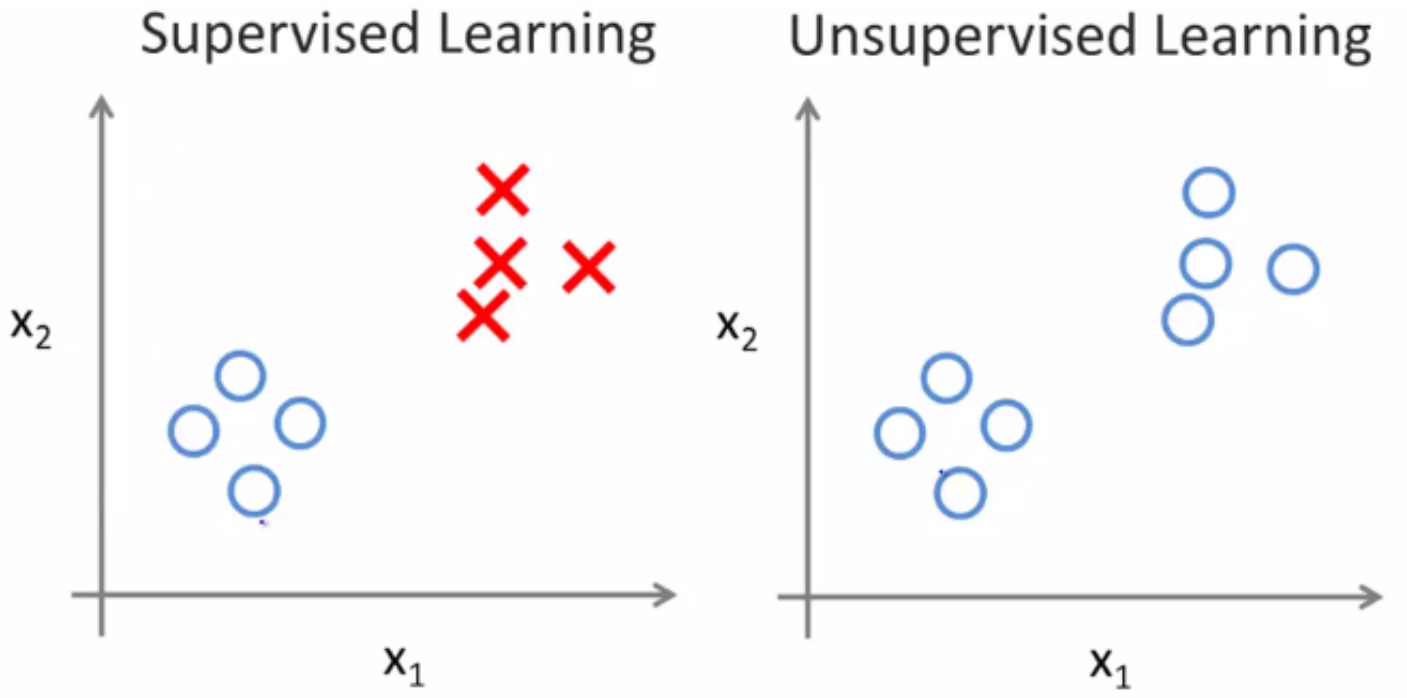


Figure 1: Examples of learning algorithms

1.2 Linear Regression with One Variable

1.2.1 Model and Cost Function: Model representation

Predicting house prices (Portland, OR) based on size is considered a supervised learning logarithm and regression problem. A training set (given dataset) is given to predict the housing prices. Symbols that are being used:

- m = number of training examples
- x^i = "input" variable/features
- y^i = "output" variable/ "target" variable
- (x, y) = denote one training example
- $(x^{(i)}, y^{(i)}) = i^{th}$ training example (index)

Training set $\rightarrow$ Learning algorithm $\rightarrow$ h (hypothesis, gives estimated value y based on x), h maps from x's to y's (see Figure 2). How do we represent h?

$$h_\Theta(x) = \Theta_0 + \Theta_1 \cdot x \tag{1}$$

This example is linear regression with one variable x, also known as **univariate (one variable) linear regression**.

$$(x^{(i)}, y^{(i)}); i = 1, ..., m. \tag{2}$$

1.2.2 Model and Cost Function: Cost Function

A cost function determines how to best fit a linear line to the data and looks as follows with parameters Θ_i:

$$h_\Theta(x) = \Theta_0 + \Theta_1 \cdot x \tag{1}$$

We can measure the accuracy of our hypothesis function by using a cost function. This takes an average difference of all the results of the hypothesis with inputs from x's and the actual output y's. Now how to choose $\Theta_{i's}$? We want the

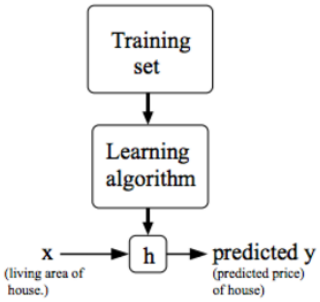


Figure 2: Model representation.

parameters that adapt a line that fits the data correctly. The idea is as follows: Choose Θ_0 and Θ_1 so that $h_\Theta(x)$ is close to y for our training example (x, y). In addition Θ_0 and Θ_1 need to be minimized, as well as:

$$J(\Theta_0, \Theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\Theta(x_i) - y_i)^2 = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \tag{3}$$

For the example of predicting house prices: the prediction of the hypothesis minus the actual housing price needs to be minimized (squared difference). m is the size of the training set (number of training examples). Putting the constant in front makes the math easier. In conclusion: find me the values of Θ_0 and Θ_1 so that the average of $\frac{1}{2m}$ times the sum squared errors between my predictions on the training set minus the actual values on the training set is minimized. $J(\Theta_0, \Theta_1)$ is also known as the "Squared error function", or "Mean squared error". The mean is halved $\frac{1}{2}$ as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the $\frac{1}{2}$ term.

1.2.3 Model and Cost Function: Cost Function - Intuition I

To recap, the hypothesis function is defined as

$$h_\Theta(x) = \Theta_0 + \Theta_1 \cdot x \tag{1}$$

with parameters Θ_0 and Θ_1. The cost function is defined as

$$J(\Theta_0, \Theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \tag{3}$$

with the goal to

$$\min_{\Theta_0, \Theta_1} J(\Theta_0, \Theta_1) \tag{4}$$

In order to explain it better equation (1) will be simplified to the following: $h_\Theta(x) = \Theta_1 x$ where $\Theta_1 = 0$. The new optimalization objective will be:

$$\min_{\Theta_1} J(\Theta_1) \tag{5}$$

This includes only hypothesis functions that cross the origin (0,0). We will want to understand two functions, namely the hypothesis function - $h_\Theta(x)$, for fixed Θ_1, this is a function of x - and the cost function - $J(\Theta_1)$. Each value of Θ_1 corresponds to a different hypothesis function (or to a different straight line fit). And then for each value of Θ_1 we can derive a different of J of Θ_1 (resulting for instance in a parabola).

1.2.4 Model and Cost Function: Cost function - Intuition II

$$h_\Theta(x) = \Theta_0 + \Theta_1 \cdot x \tag{1}$$

with parameters Θ_0 and Θ_1. Cost function:

$$J(\Theta_0, \Theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \tag{3}$$

with the goal to

$$\min_{\Theta_0, \Theta_1} J(\Theta_0, \Theta_1) \tag{4}$$

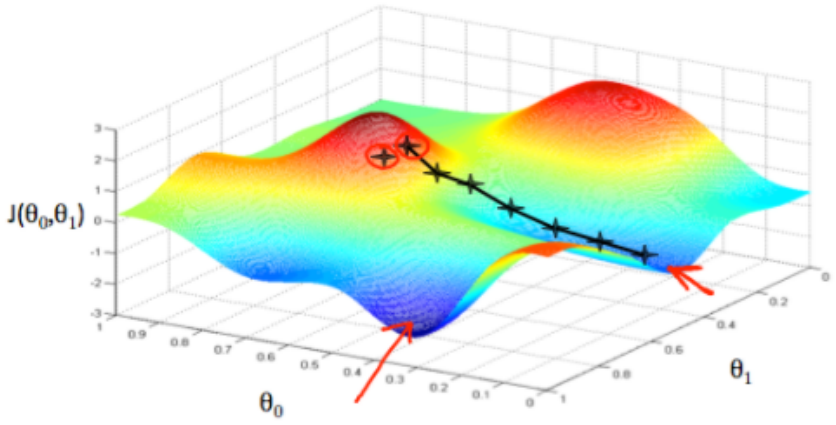


Figure 3: Gradient descent. We put Θ_0 on the x axis and Θ_1 on the y axis, with the cost function on the vertical z axis. The points on our graph will be the result of the cost function using our hypothesis with those specific theta parameters. We will know that we have succeeded when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum. The red arrows show the minimum points in the graph.

Again, we will want to understand two functions, namely the hypothesis function - $h_\Theta(x)$, for fixed Θ_1,Θ_0, which is a function of x - and the cost function - $J(\Theta_0, \Theta_1)$. With two parameters, namely Θ_0 and Θ_1, the cost function will have the 3D shaped curve. Contour plots/figures show ovals that have a set of point that take the same values for $J(\Theta_0$,$\Theta_1)$. A contour plot is a graph that contains many contour lines. A contour line of a two variable function has a constant value at all points of the same line. The centred ovals have the smallest value of J. Ultimately, we want an algorithm that automatically determines ("learns") the values for Θ_0 and Θ_1.

1.2.5 Parameter Learning - Gradient Descent

There exists an more general algorithm called gradient descent minimizes the value of J (and other functions). As mentioned earlier, we have some function of $J(\Theta_0, \Theta_1)$ where we want to

$$\min_{\Theta_0,\Theta_1} J(\Theta_0, \Theta_1) \tag{4}$$

Gradient descent can also solve this problem for a broader range of Θ_n. The outline is to start with some values for Θ_0 and Θ_1, subsequently changing the values in order to reduce $J(\Theta_0$,$\Theta_1)$ until we hopefully end up at a minimum (see Figure 3). The vertical height of the figure responds to the value of J. If you want to go down from the red hill in baby steps (to deepest descent) a couple of times you'll end up at a local minimum/optimum. Depending on where you start, you can end up at different local optima. This is an important property of gradient descent. The points on our graph will be the result of the cost function using our hypothesis with those specific Θ_j parameters. We will know that we have succeeded when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum. The red arrows show the minimum points in the graph. We will know that we have succeeded when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum. The red arrows show the minimum points in the graph. The way we do this is by taking the derivative (the tangential line to a function) of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent. The size of each step is determined by the parameter α (always positive value), which is called the learning rate. For example, the distance between each 'star' in the graph above represents a step determined by our parameter α. A smaller α would result in a smaller step and a larger α results in a larger step. The direction in which the step is taken is determined by the partial derivative of $J(\Theta_0$,$\Theta_1)$. Depending on where one starts on the graph, one could end up at different points. The gradient descent algorithm is as follows:

repeat until convergence {

$$\Theta_j := \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta_0, \Theta_1) \qquad \textit{(for } j = 0 \textit{ and } j = 1\textit{)} \tag{6}$$

}

One property of gradient descent is that the parameter Θ should be updates simultaneously. Written out more formally

the correct way is as follows:

$$temp0 := \Theta_0 - \alpha \frac{\partial}{\partial \Theta_0} J(\Theta_0, \Theta_1) \tag{7a}$$

$$temp1 := \Theta_1 - \alpha \frac{\partial}{\partial \Theta_1} J(\Theta_0, \Theta_1) \tag{7b}$$

$$\Theta_0 := temp0 \tag{7c}$$

$$\Theta_1 := temp1 \tag{7d}$$

Incorrect:

$$temp0 := \Theta_0 - \alpha \frac{\partial}{\partial \Theta_0} J(\Theta_0, \Theta_1) \tag{8a}$$

$$\Theta_0 := temp0 \tag{8b}$$

$$temp1 := \Theta_1 - \alpha \frac{\partial}{\partial \Theta_1} J(\Theta_0, \Theta_1) \tag{8c}$$

$$\Theta_1 := temp1 \tag{8d}$$

$:=$ is an assignment and overwrites the earlier value of Θ_j. α is the "learning rate" and determines the size of the step we take downhill with gradient descent. $\frac{\partial}{\partial \Theta} J(\Theta_0,\Theta_1)$ is a derivative term, which will be discussed later. During the gradient descent Θ_0 and Θ_1 will be updated simultaneously.

1.2.6 Parameter Learning - Gradient Descent Intuition

In the previous section, we discussed the gradient descent algorithm:

$$\Theta_j := \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta_0, \Theta_1) \qquad \textit{(for } j = 0 \textit{ and } j = 1\textit{)} \tag{6}$$

Now we will look at the derivative term in the gradient descent algorithm namely $\frac{\partial}{\partial \Theta} J(\Theta_0,\Theta_1)$. As an example we'll use a simplification and see how gradient descent affects this function. We'll use the same simplified function that we used earlier:

$$\min_{\Theta_1} J(\Theta_1) \tag{5}$$

where $\Theta_1 \in \mathbb{R}$ (equals real number). Where the shape of the cost function $J(\Theta_1)$ corresponds to a parabola. Imagine that we start far from the minimum. Gradient descent will update Θ_1 with the following equation:

$$\Theta_1 := \Theta_1 - \alpha \frac{\partial}{\partial} J(\Theta_1) \tag{9}$$

The derivative looks at the slope (height divided by horizontal value) of the line that touches the value of Θ_1. Dependent on whether the slope is positive or negative the Θ_1 will become smaller or bigger, respectively (due to the - α)(see Figure 4). If α is too small, the gradient descent can be slow. If α is too large on the other hand, gradient descent can overshoot the minimum. It may fail to converge, or even diverge. Moreover, gradient descent can converge to a local minimum, even with the learning rate α fixed. As we approach a local minimum, gradient descent will automatically take smaller steps. So, no need to decrease α over time. As we approach the minimum, the derivative will be closer to 0.

1.2.7 Parameter Learning - Gradient Descent For Linear Regression

Now we are going to put gradient descent together with the cost function giving us an algorithm for linear regression. To recap, here is the gradient descent algorithm:

$$\Theta_j := \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta_0, \Theta_1) \qquad \textit{(for } j = 0 \textit{ and } j = 1\textit{)} \tag{6}$$

and the linear regression model:

$$h_\Theta(x) = \Theta_0 + \Theta_1 \cdot x \tag{1}$$

$$J(\Theta_0, \Theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \tag{3}$$

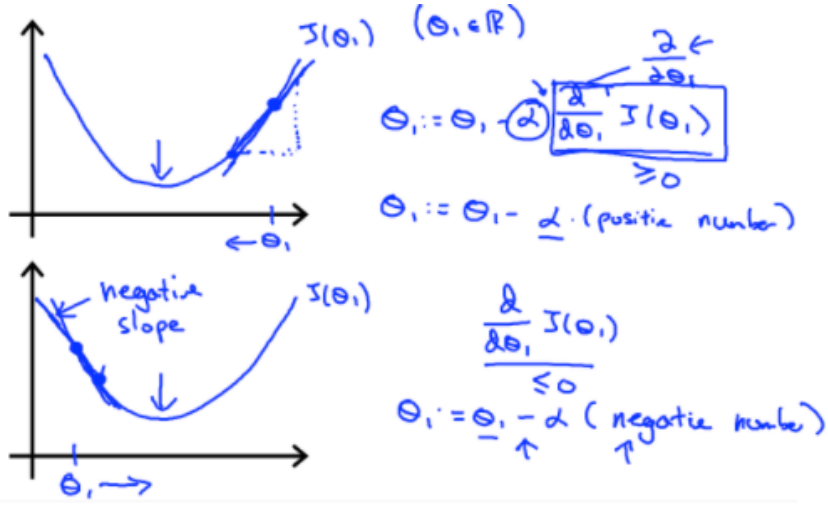


Figure 4: Gradient descent

Now we're going to apply gradient descent to minimize our cost function - $J(\Theta_0,\Theta_1)$. The key term we need is the partial derivative term:

$$\frac{\partial}{\partial\Theta_j}J(\Theta_0,\Theta_1) = \frac{\partial}{\partial\Theta_j}\frac{1}{2m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y_i)^2 \tag{10a}$$

$$\frac{\partial}{\partial\Theta_j}J(\Theta_0,\Theta_1) = \frac{\partial}{\partial\Theta_j}\frac{1}{2m}\sum_{i=1}^{m}(\Theta_0 + \Theta_1 x^i - y_i)^2 \tag{10b}$$

We want to know this for $j = 0$ and $j = 1$:

$$j = 0 : \frac{\partial}{\partial\Theta_0}J(\Theta_0,\Theta_1) = \frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y_i) \tag{11a}$$

$$j = 1 : \frac{\partial}{\partial\Theta_1}J(\Theta_0,\Theta_1) = \frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y_i)\cdot x^i \tag{11b}$$

So if we go back to the gradient descent algorithm you get the following:

repeat until convergence {

$$\begin{aligned}\Theta_0 &:= \Theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y_i)\\ \Theta_1 &:= \Theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y_i)\cdot x^i\end{aligned} \tag{12}$$

}

Quick reminder: update Θ_0 and Θ_1 simultaneously! A cost function for linear regression will always be a bowl shaped function (technical term is a "convex function"). This type of cost function has one global optimum and no local optima. This algorithm is sometimes called a "batch" gradient descent referring to the fact that at each step of gradient descent uses all the training examples. Some other versions look only at a small batch of gradient descent.

1.3 Linear Algebra Review

1.3.1 Linear Algebra Review - Matrices and Vectors

A matrix is a rectangular array of numbers between square brackets. The dimension of the matrix is defined as the number of rows times the number of columns. Matrices are often written as $\mathbb{R}^{numberofrows\times numberofcolumns}$. Denotation

is as follows, A_{ij} is equal to the "i,j entry" in the i^{th} row and j^{th} column. An example: $A_{i,j} = \begin{pmatrix} 1402 & 191 \\ 1371 & 821 \\ 949 & 1437 \\ 147 & 1448 \end{pmatrix}$

Moreover, a vector is a $n \times$ matrix defined as $\mathbb{R}^{numberofrows}$. Denotation is as follows, A_{ij} is equal to the "i entry" in the i^{th} row. 1-indexed (default) function start at row number 1 while 0-indexed starts at row number 0. An example: $y_i = \begin{pmatrix} 1402 \\ 1371 \\ 949 \\ 147 \end{pmatrix}$

1.3.2 Linear Algebra Review - Addition and Scalar Multiplication

You can only add matrices that have the same dimension (same number of rows and columns). If matrices are not of the same dimensions, the program will return an error. Scalar (real number) multiplication multiplies each element of the matrix by that scalar. Scalar division also divides each element of the matrix by the given scalar. When there is a combination of operands, first scalar multiplication and division are executed where after the addition and subtractions are performed.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} w & x \\ y & z \end{pmatrix} = \begin{pmatrix} a+w & b+x \\ c+y & d+z \end{pmatrix} \qquad \begin{pmatrix} a & b \\ c & d \end{pmatrix} - \begin{pmatrix} w & x \\ y & z \end{pmatrix} = \begin{pmatrix} a-w & b-x \\ c-y & d-z \end{pmatrix}$$

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times x = \begin{pmatrix} a \times x & b \times x \\ c \times x & d \times x \end{pmatrix} \qquad \begin{pmatrix} a & b \\ c & d \end{pmatrix} / x = \begin{pmatrix} a/x & b/x \\ c/x & d/x \end{pmatrix}$$

1.3.3 Linear Algebra Review - Matrix Vector Multiplication

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix} \times \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a \times x & b \times y \\ c \times x & d \times y \\ e \times x & f \times y \end{pmatrix}$$

For matrices and vector to be multiplied, the number of columns of the matrix should be equal to the number of rows in the vector. As an example, we have a several values of housing size (e.g. 2104, 1416, 1534, 852) with the following hypothesis function: $\mathbf{h}_\Theta$ = -40 + 0.25x. This can be expressed in the following matrix and vector resulting in the prediction of the the house sizes:

$$\begin{pmatrix} 1 & 2104 \\ 1 & 1416 \\ 1 & 1534 \\ 1 & 8520 \end{pmatrix} \times \begin{pmatrix} -40 \\ 0.25 \end{pmatrix} = \begin{pmatrix} -40 \times 1 & 0.25 \times 2104 \\ -40 \times 1 & 0.25 \times 1416 \\ -40 \times 1 & 0.25 \times 1534 \\ -40 \times 1 & 0.25 \times 8520 \end{pmatrix}$$

1.3.4 Linear Algebra Review - Matrix Matrix Multiplication

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix} \times \begin{pmatrix} w & x \\ y & z \end{pmatrix} = \begin{pmatrix} a \times w + b \times y & a \times x + b \times z \\ c \times w + d \times y & c \times x + d \times z \\ e \times w + f \times y & e \times x + f \times z \end{pmatrix}$$

As an example we again take the housing sizes (e.g. 2104, 1416, 1534, 852). However, instead of one hypothesis function, we have three:

1. $\mathbf{h}_\Theta$ = -40 + 0.25x
2. $\mathbf{h}_\Theta$ = 200 + 0.1x
3. $\mathbf{h}_\Theta$ = -150 + 0.4x

This yields the following result which represent the prediction of the house sizes:

$$\begin{pmatrix} 1 & 2104 \\ 1 & 1416 \\ 1 & 1534 \\ 1 & 852 \end{pmatrix} \times \begin{pmatrix} -40 & 200 & -150 \\ 0.25 & 0.1 & 0.4 \end{pmatrix} = \begin{pmatrix} 486 & 410 & 692 \\ 314 & 342 & 416 \\ 344 & 353 & 464 \\ 173 & 285 & 191 \end{pmatrix}$$

1.3.5 Linear Algebra Review - Matrix Multiplication Properties

Matrix matrix multiplication is really useful since you can pack a lot of computation in one matrix operation. For real numbers the following applies: $3 \times 5 = 5 \times 3$. This is called "commutative". This property is however not true for matrix multiplication.

$$A \times B \neq B \times A \qquad \textit{(not commutative)} \tag{13}$$

So in general if you have *matrix* $A_{m,n}$ and *matrix* $B_{n,m}$:
$A \times B$ is $m \times m$
$B \times A$ is $n \times n$

Matrix multiplication is "associative". This mean when you have $A \times B \times C$:

$$A \times (B \times C) = (A \times B) \times C \tag{14}$$

Another type of matrix is an "Identity matrix".

$$1 \times z = z \times 1 = z \qquad \textit{(1 is identity)} \tag{15}$$

You also have identity matrices denoted as I (or $I_{n \times n}$). Examples of identity matrices:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

An identity has the property that it has 1's along its diagonals and 0's everywhere else. It turns out that an identity matrix has the following property:

$$A \cdot I = I \cdot A = A \tag{16}$$

- if $A \times I \rightarrow I_{n \times n}$
- if $I \times A \rightarrow I_{m \times m}$

In other words, when multiplying the identity matrix after some matrix - $A \times I$ - the square identity matrix's dimension should match the other matrix's columns. When multiplying the identity matrix before some other matrix - $I \times A$ - the square identity matrix's dimension should match the other matrix's rows. Moreover, when one of the two matrices is an identity matrix, the following applies:

$$I \times A = A \times I \tag{17}$$

1.3.6 Linear Algebra Review - Inverse and transpose

A number multiplies by the inverse number yields 1 ($3 \cdot 3^{-1} = 1$). However, not all numbers have an inverse (i.e. 0, 0^{-1} is undefined). For matrices, the following applies: if A is an m $\times$ m matrix and if it has an inverse:

$$AA^{-1} = A^{-1}A = I_{m,m} \tag{18}$$

These type of matrices are also called a "square matrix", because the number of rows equals number of columns. The inverse of a matrix is mostly done by numerical software (not manually). Moreover, if matrix A only includes 0's it also has no inverse. Matrices that don't have an inverse are "singular" or "degenerate". A matrix transpose if defined as follows:

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 5 & 9 \end{pmatrix} \qquad A^T = \begin{pmatrix} 1 & 3 \\ 2 & 5 \\ 0 & 9 \end{pmatrix}$$

So, let A be an m $\times$ n matrix and let $B = A^T$. Then B is a n $\times$ m matrix and

$$B_{i,j} = A_{j,i} \tag{19}$$

2 Week 2

2.1 Linear Regression with Multiple Variables

2.1.1 Multivariate Linear Regression - Multiple Features

To following notations are used:

- n = number of features
- $x^{(i)}$ = input (features) of i^{th} training example.
- $x_j^{(i)}$ = value of feature j in i^{th} training example.
- y as output and m as number of training examples.

The previously formed hypothesis was:

$$h_\Theta(x) = \Theta_0 + \Theta_1 \cdot x \tag{1}$$

The newly formed hypothesis including more that one feature will be as follows:

$$h_\Theta(x) = \Theta_0 + \Theta_1 x_1 + \Theta_2 x_2 + \Theta_3 x_3 + \cdots + \Theta_n x_n \tag{20}$$

For convenience of notation define $x_0 = 1$ ($x_0^{(i)} = 1$). So for every feature i_0 is equal to 1. You can see this as defining an additional 0 feature. Previously, we had n features. This allows us to do matrix operations with Θ and x. So this means that:

$$x = \begin{pmatrix} i_0 \\ i_1 \\ i_2 \\ \vdots \\ i_n \end{pmatrix} \in \mathbb{R}^{n+1} \qquad \Theta = \begin{pmatrix} \Theta_0 \\ \Theta_1 \\ \Theta_2 \\ \vdots \\ \Theta_n \end{pmatrix} \in \mathbb{R}^{n+1} \qquad \text{(0-indexed vectors)}$$

When we take the newly formed hypothesis (20), this can also be written as $h_\Theta(x) = \Theta^T\ x$. This results in the product of the following to matrices:

$$h_\Theta(x) = \begin{pmatrix} \Theta_0 & \Theta_1 \dots \Theta_n \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} = \Theta_0 x_0 + \Theta_1 x_1 + \Theta_2 x_2 + \Theta_3 x_3 + \cdots + \Theta_n x_n$$

This also called **multivariate linear regression**, meaning that we have multiple features.

2.1.2 Multivariate Linear Regression - Gradient Descent for Multiple Variables

The hypothesis stated earlier (20) will yield the following cost function:

$$J(\Theta_0, \Theta_1, \dots, \Theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \tag{21}$$

Instead of thinking of J having multiple parameters, we notate the cost function with $J(\Theta)$, where Θ is a vector of values $(\Theta_0, \Theta_1, \dots, \Theta_n)$. The cost function can also be written as follows:

$$J(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} (\Theta^T x^{(i)} - y^{(i)})^2 \tag{22a}$$

$$J(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} ((\sum_{j=0}^{n} \Theta_j x_j^{(i)}) - y^{(i)})^2 \qquad \textit{(Inner sum starts at 0)} \tag{22b}$$

Gradient descent looks as follows:

Repeat {

$$\Theta_j := \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta) \qquad \textit{(simultanously update for every j = 0,...,n)} \tag{23}$$

}

The gradient descent for one feature ($n = 1$). The gradient descent function was as follows:

repeat until convergence {

$$\begin{aligned}\Theta_0 &:= \Theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y_i)\\ \Theta_1 &:= \Theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y_i)\cdot x^i\end{aligned} \tag{12}$$

}

The gradient descent for more than one feature ($n \geq 1$):

repeat until convergence {

$$\begin{aligned}\Theta_0 &:= \Theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y_i)\cdot x_0^{(i)}\\ \Theta_1 &:= \Theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y_i)\cdot x_1^{(i)}\\ \Theta_2 &:= \Theta_2 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y_i)\cdot x_2^{(i)}\end{aligned} \tag{24}$$

}

2.1.3 Multivariate Linear Regression - Gradient Descent in Practice I - Feature Scaling

Feature scaling means making sure that features are on similar scale (same range of values). In this way, gradient descent can converge more quickly. Scaling can be done by dividing a value for a feature by the maximum value, this yields a value between 0 and 1. More generally, we want to get every feature into approximately a -1 $\leq x_i \leq$ 1 range. You can also perform mean normalization, which means replacing x_i with x_i - μ_i followed be division with the maximum value to make features have approximately zero mean (-0.5 $\leq x_i \leq$ 0.5 range). In summary:

$$x_i \leftarrow \frac{x_i - \mu_i}{s_i} \tag{25}$$

where μ_i is de average value of x_i and s_i the range (*max* - *min*, or the standard deviation). This way, gradient descent goes much quicker (less iteration steps are needed). So in recap, we can speed up gradient descent by having each of our input values in roughly the same range. This is because Θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven. Two techniques to help with this are feature scaling and mean normalization. Feature scaling involves dividing the input values by the range (i.e. the maximum value minus the minimum value) of the input variable, resulting in a new range of just 1. Mean normalization involves subtracting the average value for an input variable from the values for that input variable resulting in a new average value for the input variable of just zero. To implement both of these techniques, adjust your input values as shown in equation (25).

2.1.4 Multivariate Linear Regression - Gradient Descent in Practice II - Learning Rate

$$\Theta_j := \Theta_j - \alpha\frac{\partial}{\partial\Theta_j}J(\Theta) \qquad \textit{(simultanously update for every j = 0,...,n)} \tag{23}$$

The following section describes how to choose learning rate α. "Debugging" is a method that improves the efficiency of the gradient descent. This is done by making a plot with number of iterations on the x-axis. Now plot the cost function, J_Θ over the number of iterations of gradient descent. If J_Θ ever increases, then you probably need to decrease α. An example of an automatic convergence test is a test that declares if convergence if J_Θ decreases by less than 10^{-3} in one iteration. If J_Θ becomes larger at more iterations the learning rate (α) is probably too large. For sufficiently small α, J_Θ should decrease on every iteration. But if α is too small, gradient descent can be slow to converge (to choose α try 0.001, 0.01, 0.1, 1, etc.).

Examples: $m = 4$.

	Size (feet²)	Number of bedrooms	Number 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	852	2	1	36	178

Figure 5: Normal equation

2.1.5 Multivariate Linear Regression - Features and Polynomial Regression

We can improve our features and the form of our hypothesis function in a couple of different ways. We can combine multiple features into one. For example, we can combine x_1 and x_2 into a new feature x_3 by taking $x_1 \cdot x_2$. Moreover, our hypothesis function need not be linear (a straight line) if that does not fit the data well. We can change the behaviour or curve of our hypothesis function by making it a quadratic, cubic or square root function (or any other form). One important thing to keep in mind is, if you choose your features this way then feature scaling becomes very important.

2.1.6 Computing Parameters Analytically - Normal Equation

Gradient descent gives one way of minimizing J. Let's discuss a second way of doing so, this time performing the minimization explicitly and without resorting to an iterative algorithm. In the "Normal Equation" method, we will minimize J by explicitly taking its derivatives with respect to the Θ_j's, and setting them to zero. This allows us to find the optimum Θ without iteration. Let's look at an example: If 1D ($\Theta \in \mathbb{R}$) with the following function. To minimize the function you set the derivative to 0. This allows you to solve the value of Θ that minimizes $J(\Theta)$.

$$J(\Theta) = a\Theta^2 + b\Theta + c \qquad \textit{(quadratic function)} \tag{26}$$

$$\frac{d}{d\Theta} J(\Theta) = 0 \tag{27}$$

However, in this case Θ is a real number while actually Θ is a $n + 1$ dimensional parameter vector ($\mathbb{R}^{n+1}$) with a corresponding cost function $J(\Theta_0, \Theta_1, \ldots, \Theta_m)$

$$J(\Theta_0, \Theta_1, \ldots, \Theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \tag{21}$$

and the following minimization function:

$$\frac{\partial}{\partial \Theta_j} J(\Theta) = \ldots = 0 \qquad \textit{(for every j)} \tag{28}$$

Solve for $\Theta_0, \Theta_1, \ldots, \Theta_n$. Let's look at an example for housing prices (Figure 5). The dataset can be represented in a matrix:

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

So X is going to be an $m \times (n + 1)$ dimensional matrix and Y an m-dimensional vector. In the general case you have m examples with n features which gives the following design matrix (X):

$$x^{(i)} = \begin{pmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{pmatrix} \in \mathbb{R}^{n+1} \qquad X = \begin{pmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ (x^{(3)})^T \\ \vdots \\ (x^{(m)})^T \end{pmatrix} \qquad \text{m} \times (\text{n} + 1) \text{ dimensional matrix, with } Y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ \vdots \\ y^{(m)} \end{pmatrix}$$

Subsequently, we get the following value of Θ that minimizes the cost function:

$$\Theta = (X^T X)^{-1} X^T y \tag{29}$$

$(X^T X)^{-1}$ is the inverse of matrix $X^T X$. If you use this formula, feature scaling is not necessary (unless you use gradient descent. Using gradient descent versus normal equation has the following (dis)advantages (for m training examples and n features). For **gradient descent**:

- Need to choose α
- Needs many iterations
- Works well even when n is large.

For **normal equation**:

- No need to choose α
- Don't need to iterate.
- Need to compute $(X^T X)^{-1}$. With the normal equation, computing the inversion has complexity $O(n^3)$. So if we have a very large number of features, the normal equation will be slow. In practice, when n exceeds 10,000 it might be a good time to go from a normal solution to an iterative process.
- Slow if n is very large.

2.1.7 Computing Parameters Analytically - Normal Equation Noninvertibility

What if $(X^T X)^{-1}$ is non-invertible (singular/degenerate)?

- Redundant features (linearly dependent)
- Too many features (e.g. $m \leq n$) $\rightarrow$ delete some features, or use regularization.

2.2 Octave/MATLAB Tutorial

2.2.1 Octave/MATLAB Tutorial - Basic Operations

- *PS1('>>')* $\rightarrow$ defines prompt (Octave)
- *sprintf(' ')* $\rightarrow$ printing out string
- *ones(), zeros()* $\rightarrow$ creates a matrix of ones or zeros
- *rand()* $\rightarrow$ creates matrix with random continuous numbers (6 decimals)
- *randn()* $\rightarrow$ random numbers with mean of 0
- *hist(dataset, number of bins)* $\rightarrow$ creates histogram
- *eye(n)* $\rightarrow$ creates a $(n \times n)$ identity matrix

2.2.2 Octave/MATLAB Tutorial - Moving Data Around

- *size()* $\rightarrow$ returns size of matrix
- *length()* $\rightarrow$ returns longest dimension of a matrix/vector
- *pwd* $\rightarrow$ shows current directory
- *cd* $\rightarrow$ changes current directory
- *ls* $\rightarrow$ shows files in current directory
- *load(' ')* $\rightarrow$ loads files name
- *who* $\rightarrow$ shows variables in work space

- *whos* → list variables with details in work space
- *clear* → deletes variable
- *save('filename', dataset)* → saves variable in cd (compressed file)
- *save('filename.txt', dataset)-ascii* → saves file as text (ASCII)
- *A(:)* → puts all elements of A into a single vector

2.2.3 Octave/MATLAB Tutorial - Computing on Data

- A *.** B → element-wise multiplication of two matrices (also applies to vectors)
- *A'* → transpose of matrix A
- *[val,ind]* → gives certain value and index number
- *magic()* → returns matrices in which all rows and columns have an equal sum
- *[r,c]* → indexes rows and columns
- *help ...* → finds function
- *floor()/ceil()* → rounds downs or up, respectively
- within functions *1* refers to rows and *2* refers to columns
- $sum(sum(A. * eye(9)))$ → gives the sum of the diagonal values
- *flipud()* → flips matrix vertically
- *pinv()* → returns inverse of matrix

2.2.4 Octave/MATLAB Tutorial - Plotting Data

- *plot()* → plots data
- *xlabel(),ylabel()* → label axes
- *legend()* → adds legend to plot
- *title()* → adds title to plot
- *close* → closes figure
- *subplot(x,y,z* → creates a x by y grid and plots element z
- *clf* → clears figure
- *imagesc()* → plot grid of colours corresponding to values in the matrix
- *colorbar* → adds colourbar
- a comma separating commands can do different operations in one prompt line ("comma chaining")

2.2.5 Octave/MATLAB Tutorial - Control Statements: for, while, if statement

- *break* → terminates execution of for or while loop
- *addpath()* → add directory to search path

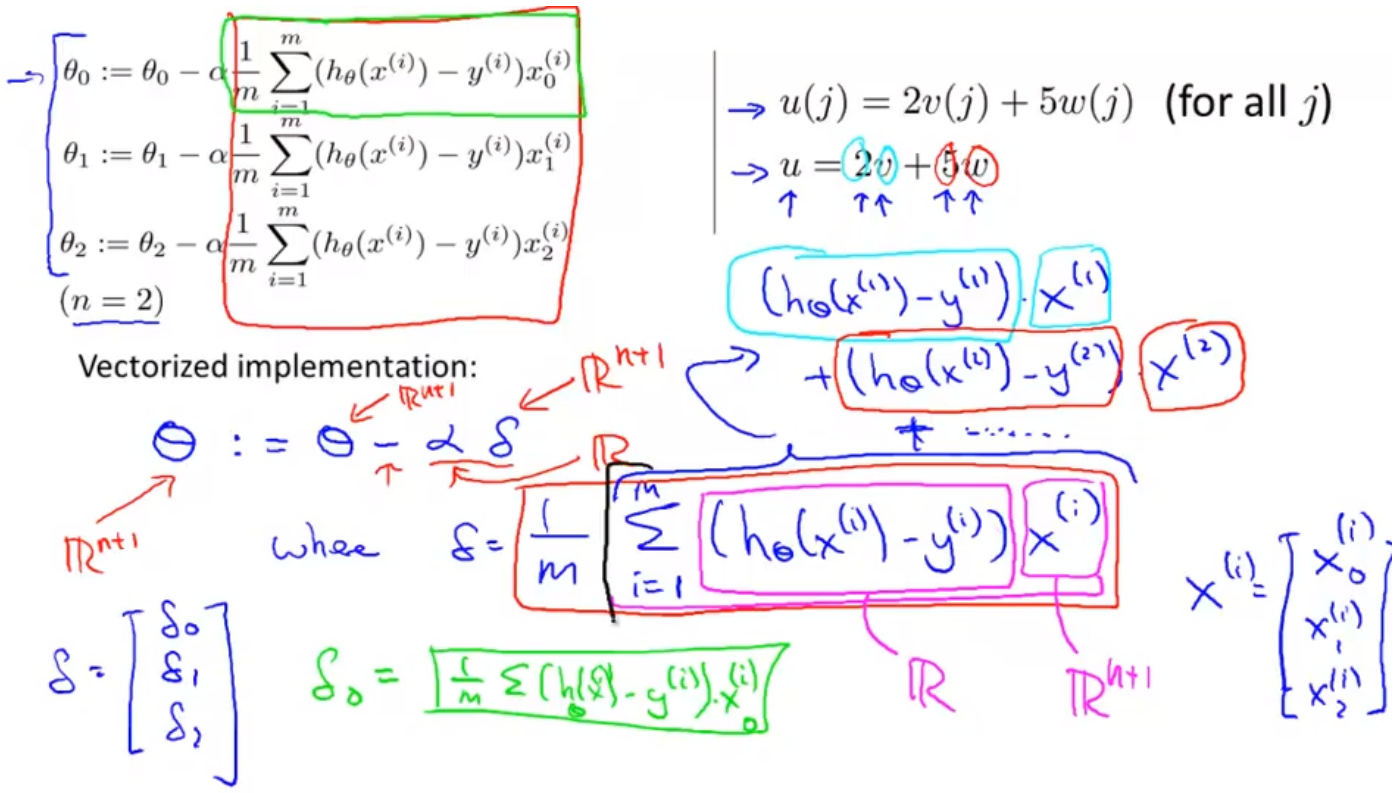


Figure 6: Normal equation

2.2.6 Octave/MATLAB Tutorial - Vectorization

Vectors in MATLAB are 1-indexed. Vectorizing Θ by transposing the values doesn't require a loop statement (runs faster and more efficiently). For example, if we take the hypothesis function:

$$h_\Theta(x) = \sum_{j=0}^{n} \Theta_j x_j \tag{30}$$

leading to an unvectorized implementation:

```
prediction = 0.0
for j = 1: n + 1,
    prediction = prediction + Θ_j · x_j
end;
```

However, when we vectorize Θ_j:

$$h_\Theta(x) = \Theta^T x \tag{31}$$

we are able to perform a vectorized implementation:

```
prediction = Θ' · x;
```

Now let's look at a more complex example.

repeat until convergence {

$$\begin{aligned}
\Theta_j &:= \Theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y_i) \cdot x_j^{(i)} \\
&\textit{with simultenous update} \\
\Theta_0 &:= \Theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y_i) \cdot x_0^{(i)} \\
\Theta_1 &:= \Theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y_i) \cdot x_1^{(i)} \\
\Theta_2 &:= \Theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y_i) \cdot x_2^{(i)}
\end{aligned} \tag{24}$$

}

A vectorized implementation will go as follows:

$$\begin{aligned}
\Theta := \Theta - \alpha \cdot \delta \qquad \textit{where } \delta &= \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y_i) \cdot x^{(i)} \\
\Theta &= \mathbb{R}^{n+1} \\
\alpha &= \mathbb{R} \\
\delta &= \textit{vector}
\end{aligned} \tag{32}$$

The vector δ is a $n + 1$-dimensional vector. So we get $\delta = \begin{pmatrix} \delta_0 \\ \delta_1 \\ \delta_2 \end{pmatrix}$

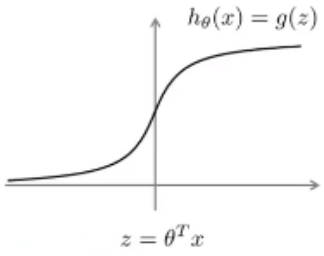


Figure 7: Sigmoid Function/Logistic Function. The function *g(z)* is shown here, which maps any real number to the *(0,1)* interval. z is defined as $\Theta^T x$.

3 Week 3

3.1 Logistic Regression

3.1.1 Classification and Representation - Classification

Classification problems are problems where the variable you want to predict is a discrete value (output is 0 or 1). The denotation which is 0 or 1 is arbitrary.

$$y \in \{0, 1\} \qquad \textit{0: "Negative Class" and 1: "Positive Class"} \tag{33}$$

You have binary (2 output variables) and multiclass (more than 2 output variables) classification problems. With classification problems, you also want to compute a hypothesis function as a function of linear regression:

$$h_\Theta(x) = \Theta^T x \tag{31}$$

with a threshold classifier output, e.g:

- if $h_\Theta(x) \geq 0.5$, predict $y = 1$
- if $h_\Theta(x) < 0.5$, predict $y = 0$

However, applying linear regression to a classification problem does often not work well, because classification is not actually a linear function. Subsequently, for classification problems **logistic regression** is often used, which has the following hypothesis function output: $0 \leq h_\Theta(x) \leq 1$ (where y has the value 0 or 1). Logistic regression is used as a classification algorithm. In other words, the classification problem is just like the regression problem, except that the values we now want to predict take on only a small number of discrete values. For now, we will focus on **the binary classification problem** in which y can take on only two values, 0 and 1. For instance, if we are trying to build a spam classifier for email, then $x^{(i)}$ may be some features of a piece of email, and y may be 1 if it is a piece of spam mail, and 0 otherwise. Hence, $y \in \{0, 1\}$. 0 is also called the negative class, and 1 the positive class, and they are sometimes also denoted by the symbols "-" and "+". Given $x^{(i)}$, the corresponding $y^{(i)}$ is also called the label for the training example.

3.1.2 Classification and Representation - Hypothesis Representation

The hypothesis representation is the function that represent our hypothesis when we have a classification problem. The Logistic Regression Model is as follows:

$$h_\Theta(x) = g(\Theta^T x) \tag{34}$$

where g is a Sigmoid/Logistic function:

$$g(z) = \frac{1}{1 + e^{(-z)}} \tag{35}$$

When equation (??) and (??) are put together, we get the following hypothesis function:

$$h_\Theta(x) = \frac{1}{1 + e^{-\Theta^T x}} \tag{36}$$

The Sigmoid function is shown in Figure 7. For this function we have to fit the parameter Θ. But let's first talk about the interpretation of this model. $h_\Theta(x)$ is an estimated probability that $y = 1$ on input x:

$$h_\Theta(x) = P(y = 1|x; \Theta) \qquad \textit{"probability that y = 1, given x, paramterized by } \Theta\textit{"} \tag{37}$$

From this, the following can be concluded:

- $P(y = 0|x; \Theta) + P(y = 1|x; \Theta) = 1$
- $P(y = 0|x; \Theta) = 1 - P(y = 1|x; \Theta)$

3.1.3 Classification and Representation - Decision Boundary

To recap, the hypothesis function is as follows:

$$h_\Theta(x) = g(\Theta^T x) = P(y = 1|x; \Theta) \tag{38}$$

Suppose we have the following hypothesis function:

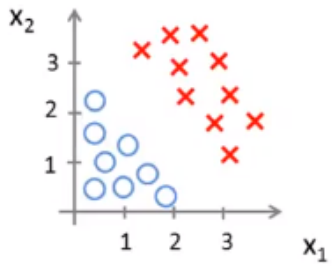


Figure 8: Decision Boundary with 2 parameters

$$h_\Theta(x) = g(\Theta_0 + \Theta_1 x_1 + \Theta_2 x_2) \tag{39}$$

This will give the following prediction if $\Theta = \begin{pmatrix} -3 \\ 1 \\ 1 \end{pmatrix}$. The line that separates these two predictions in Figure 8 is called the **decision boundary** where:

- "$y = 1$" if $-3 + x_1 + x_2 \geq 0$ which leads to $x_1 + x_2 \geq 3$
- "$y = 0$" if $-3 + x_1 + x_2 \leq 0$ which leads to $x_1 + x_2 \leq 3$

Now lets look at **non-linear decision boundaries** (see Figure 9). Suppose we have the following hypothesis function:

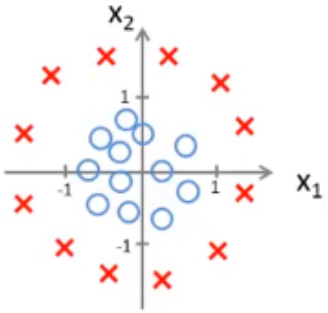


Figure 9: Non-Linear Decision Boundary with 2 parameters

$$h_\Theta(x) = g(\Theta_0 + \Theta_1 x_1 + \Theta_2 x_2) + \Theta_3 x_1^2 + \Theta_4 x_2^2 \tag{40}$$

This will give the following prediction if $\Theta = \begin{pmatrix} -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$. The decision boundary will have the form of a circle where:

- "$y = 1$" if $-1 + x_1^2 + x_2^2 \geq 0$ which leads to $x_1^2 + x_2^2 \geq 1$
- "$y = 0$" if $-1 + x_1^2 + x_2^2 \geq 0$ which leads to $x_1^2 + x_2^2 \leq 1$

If you have even higher polynomial terms than it is possible to show that you get more complex decision boundaries. In summary, the decision boundary is a property of the hypothesis and parameters. The values of the parameters Θ defines the decision boundary.

3.1.4 Logistic Regression Model - Cost Function

Training set: $\{(x^1, y^1), (x^2, y^2), \cdots, (x^m, y^m)\}$

with m examples where $x \in \begin{pmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{pmatrix}$ $x_0 = 1$, $y \in \{0, 1\}$ and

$$h_\Theta(x) = \frac{1}{1 + e^{-\Theta^T x}} \tag{36}$$

Now how to choose parameters Θ? The cost function of a logistic regression is called **"non-convex"** because the function of $J(\Theta)$ has many local minima. Consequently, it is not guaranteed that the convergence to the global minimum will occur. In contrast, a **"convex"** has a bowl-like shape with one global minimum and no local minima. So we want to have a cost function for logistic regression that is convex. The **cost function** that we will use for logistic regression is as follows:

$$J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} Cost(h_\Theta(x^{(i)}), y^{(i)}) \tag{41}$$

where:

$$Cost(h_\Theta(x), y) = \begin{cases} -log(h_\Theta(x)) & \text{if } y = 1 \\ -log(1 - h_\Theta(x)) & \text{if } y = 0 \end{cases} \tag{42}$$

Captures intuition that if $h_\Theta(x) = 0$ (predict $P(y = 1|x; \Theta) = 0$), but $y = 1$, we'll penalize learning algorithm by a very large cost (see Figure 10). Writing the cost function in this way guarantees that $J(\Theta)$ is convex for logistic regression

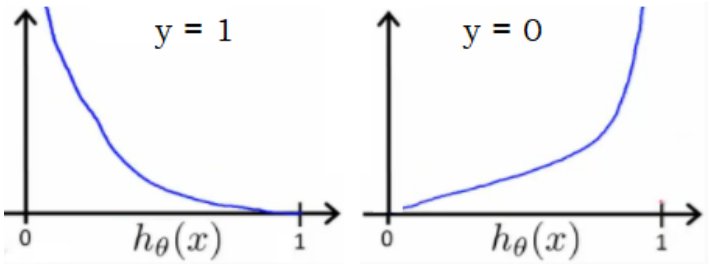


Figure 10: Cost function classification problem

3.1.5 Logistic Regression Model - Simplified Cost Function and Gradient Descent

Recap from the earlier sections:

$$J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} Cost(h_\Theta(x^{(i)}), y^{(i)}) \tag{41}$$

$$Cost(h_\Theta(x), y) = \begin{cases} -log(h_\Theta(x)) & \text{if } y = 1 \\ -log(1 - h_\Theta(x)) & \text{if } y = 0 \end{cases} \tag{42}$$

Note: *y = 0 or 1* always! Now we are going to combine the two functions in order to perform gradient descent. The combined cost function is as follows:

$$Cost(h_\Theta(x), y) = -y \cdot log(h_\Theta(x)) - (1 - y) \cdot log(1 - h_\Theta(x)) \tag{43}$$

So this definition is a more compact way of taking both expression and writing them in a more convenient form with just one line. So we get the following logistic regression cost function:

$$J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} Cost(h_\Theta(x^{(i)}), y^{(i)}) \tag{41}$$

$$J(\Theta) = -\frac{1}{m} [\sum_{i=1}^{m} y^{(i)} \cdot log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) \cdot log(1 - h_\Theta(x^{(i)}))] \tag{44}$$

This function can be derived from statistics using the principle of maximum likelihood estimation. This is an idea in statistics for how to efficiently find parameters' data for different models. And it also has a nice property that it is convex. So this is the cost function that essentially everyone used when fitting logistic regression models. To fit parameters we want to:

$$\min_{\Theta_1} J(\Theta_1) \tag{5}$$

We are going to minimize Θ as a function of J by gradient descent. Our usually template is as follows:

Repeat {

$$\Theta_j := \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta_0, \Theta_1) \qquad \textit{(simultanously update all } \Theta_j\textit{)} \tag{6}$$

}

If you take the derivative by using calculus you get the following gradient descent algorithm:

Repeat {

$$\Theta_j := \Theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y_i) x_j^{(i)} \qquad \textit{(simultanously update all } \Theta_j\textit{)} \tag{12}$$

}

You will notice that this algorithm looks identical to the linear regression gradient descent algorithm. So are linear and logistic regression different algorithm or not? What has changed is the definition of the hypothesis function:

- for linear regression $h_\Theta(x) = \Theta^T x$
- for logistic regression $h_\Theta(x) = \frac{1}{1+e^{-\Theta^T x}}$

So, although the update rule look identical, they are not! The method for convergence that was discussed earlier also applies for logistic regression. A vectorized implementation (for an efficient way of working) is as follows:

$$\Theta := \Theta - \frac{\alpha}{m} X^T (g(X\Theta) - \overrightarrow{y}) \tag{45}$$

3.1.6 Logistic Regression Model - Advanced Optimization

Given Θ, we have a code that can compute

- $J(\Theta)$
- $\frac{\partial}{\partial \Theta_j} J(\Theta)$ for j = *0,1,...,n*

So we have some cost function and we want to minimize it. So what we need to do is write a code that can take as input the parameters Θ and as output 1) $J(\Theta)$ and 2) the partial derivative terms for j equals *0, 1* up to *n*. This can be done by using optimization algorithms. These are algorithms that find the best solution from all feasible solutions for specific problems (e.g. logistic and linear regression). Several optimization algorithms include:

- Gradient Descent
- Conjugate Gradient
- BFGS
- L-BFGS

The advantages of the last three algorithms is that there is no need to manually pick α (is done automatically) and often have a faster gradient descent. A disadvantage is that they are more complex. Let's look at en example, you have the following Θ parameter:

$\Theta = \begin{pmatrix} \Theta_1 \\ \Theta_2 \end{pmatrix}$

$J(\Theta) = (\Theta_1 - 5)^2 + (\Theta_2 - 5)^2$

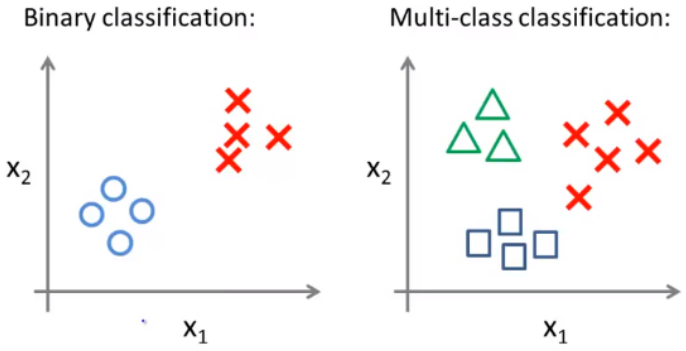


Figure 11: Classification problems

$\frac{\partial}{\partial \Theta_1} J(\Theta) = 2(\Theta_1 - 5)$

$\frac{\partial}{\partial \Theta_2} J(\Theta) = 2(\Theta_2 - 5)$

The code in MATLAB is as follows:

```
function [jVal, gradient] = costFunction(theta)
    jVal = (theta(1) - 5)^2 + (theta(2) - 5)^2;
    gradient = zeros(2,1);
    gradient(1) = 2 * (theta(1) - 5);
    gradient(2) = 2 * (theta(2) - 5);
```

Than you can call the optimization function "*fminunc*" which stands for "function minimization unconstrained":

```
options = optimset('GradObj' , 'on' , 'MaxIter' , '100');
initialTheta = zeros(2,1);
[optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta, options);
```

You set a few options. "GradObj" and "on" sets the gradient objective parameter to on. It means that you are going to provide a gradient to this algorithm. With "MaxIter" to the number of iterations, in this case 100. Subsequently, we are going to provide it with an initial guess of Θ. Then the "fminunc" function is called upon. The "@costFunction" calls upon the function that we just defined earlier. Than it performs more advanced optimization algorithms where you don't have to manually choose your α rate. Eventually, the optimal value of Θ is returned (in this case $\Theta_1 = 5$ and $\Theta_2 = 5$). How to implement this with the logistic regression algorithm?

You have theta $= \begin{pmatrix} \Theta_0 \\ \Theta_1 \\ \vdots \\ \Theta_n \end{pmatrix}$

```
function [jVal, gradient] = costFunction(theta)
    jVal = [code to compute J(Θ)]
    gradient(1) = [code to compute ∂/∂Θ0 J(Θ)];
    gradient(2) = [code to compute ∂/∂Θ1 J(Θ)];
    ⋮
    gradient(n + 1) = [code to compute ∂/∂Θn J(Θ)] ;
```

3.1.7 Multiclass Classification - Multiclass Classification: One-vs-all

We already know how to do binary classification. With an algorithm called one-vs-all classification we can solve multi-class classification as well (see Figure 12). This include defining multi-class classification problems as separate binary classification problems:

$y \in \{0, 1, \cdots, n\}$

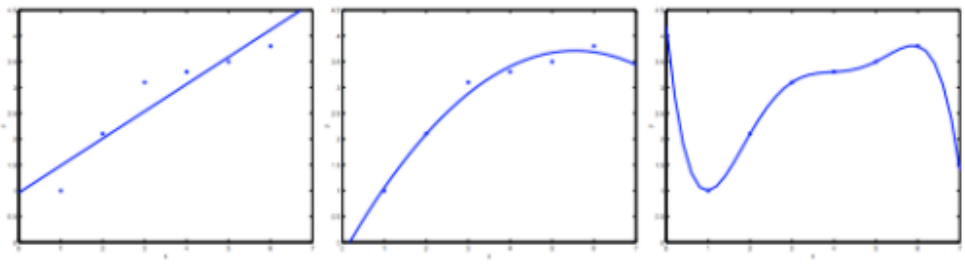


Figure 12: Underfitting, or "high bias", is when the form of our hypothesis function h maps poorly to the trend of the data (left). Overfitting, or "high variance", is caused by a hypothesis function h that fits the available data but does not generalize well to predict new data (right).

$h_\Theta^{(i)}(x) = P(y = 0|x; \Theta)$
$h_\Theta^{(i)}(x) = P(y = 1|x; \Theta)$
$\dots$
$h_\Theta^{(i)}(x) = P(y = n|x; \Theta)$
prediction $= \max_i h_\Theta^{(i)}(x)$

We are basically choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that returned the highest value as our prediction. So to summarize, we want to train a logistic regression classifier $h_\Theta^{(i)}(x)$ for each class i to predict the probability that $y = 1$. On a new input x, to make a prediction, pick the class i that maximizes

$$\max_i h_\Theta^{(i)}(x).$$

3.2 Regularization

3.2.1 Solving the Problem of Overfitting - The Problem of Overfitting

Fitting a straight line to the data could result in an underfit ("high bias"). Choosing a higher order polynomial line can result in an overfit ("high variance"). The under- and overfit are two extremes (see Figure 12). In other words, the problem of overfitting comes when we have too many features, resulting in an hypothesis that may fit the training set very well, but fails to generalize to new examples. So in summary, **underfitting**, or "high bias", is when the form of our hypothesis function h maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features. At the other extreme, **overfitting**, or "high variance", is caused by a hypothesis function h that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data. This terminology is applied to both linear and logistic regression. There are two main options to address the issue of overfitting:

1. Reduce number of features
   - Manually select which features to keep.
   - Model selection algorithm.
2. Regularization
   - Keep all the features, but reduce magnitude/values of paramters Θ_j.
   - Works well when we have a lot of features, each of which contributes a bit to predicting y.

3.2.2 Solving the Problem of Overfitting - Cost Function

Suppose you want to penalize and make Θ_3 Θ_4 really small in the following hypothesis function:
$\Theta_0 + \Theta_1 x + \Theta_2 x^2 + \Theta_3 x^3 + \Theta_4 x^4$
You can take the cost function and these parameters:

$$J(\Theta_0, \Theta_1, \dots, \Theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 + 1000\Theta_3^2 + 1000\Theta_4^2 \tag{21}$$

Now, if we want to minimize the cost function we want to reduce Θ_3 and Θ_4 resulting in removing these parameters from the equation. Then the final hypothesis function will be as follows:
$\Theta_0 + \Theta_1 x + \Theta_2 x^2 + \Theta_3 x^3 + \Theta_4 x^4 \quad \rightarrow \quad \Theta_0 + \Theta_1 x + \Theta_2 x^2$
The idea of regularization is that when we have small values for parameters $\Theta_0, \Theta_1, \cdots, \Theta_n$ this leads to a "simpler" hypothesis which is less prone to overfitting. However, because we don't know which Θ_j we want to reduce what we'll do is modifying the cost function by adding a regularization term which shrinks every single parameters. So with this added term we tend to shrink all of my parameters Θ_{1-n}:

$$J(\Theta) = \frac{1}{2m}[\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 + \lambda \sum_{i=1}^{m} \Theta_j] \tag{46}$$

λ is called the regularization parameter and controls a trade of between two different goals: the first goal is to fit the training data well and the second goal is keeping the Θ parameter small (resulting in a relatively simple hypothesis function). If λ is set to an extremely large value then we end up with all the parameters Θ close to zero.

3.2.3 Solving the Problem of Overfitting - Regularized Linear Regression

For linear regression, we have previously worked out two learning algorithms. One based on gradient descent and one based on the normal equation. Now we'll take those two algorithms and generalize them to the case of regularized linear regression. Here's the optimization objective that we came up with last time for regularized linear regression:

$$J(\Theta) = \frac{1}{2m}[\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 + \lambda \sum_{i=1}^{m} \Theta_j] \tag{46}$$

Gradient descent

Gradient descent goes as follows when including the regularization term:

repeat until convergence {

$$\begin{aligned} \Theta_0 &:= \Theta_0 - \alpha \frac{1}{m}[\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y_i) \cdot x_0^{(i)}] \quad \rightarrow \quad \frac{\partial}{\partial \Theta_0} J(\Theta) \\ \Theta_j &:= \Theta_j - \alpha[\frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y_i) \cdot x_j^{(i)} + \frac{\lambda}{m}\Theta_j] \quad \rightarrow \quad \frac{\partial}{\partial \Theta_j} J(\Theta) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (j = 1,2,3,\ldots,n) \end{aligned} \tag{47}$$

}

We distinguish between Θ_0 and Θ_j because we do not tend to penalize Θ_0. Now, if you group all the terms together that depend on Θ_j you can show that this update can be written equivalently as follows:

$$\Theta_j := \Theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i) \cdot x_j^{(i)} \tag{48}$$

The term $(1 - \alpha\frac{\lambda}{m})$ is a term with an interesting effect. $(1 - \alpha\frac{\lambda}{m})$ will always have a value smaller than 1 ($(1 - \alpha\frac{\lambda}{m}) < 1$). So with every iteration we always multiply Θ_j with a number smaller that 1.

Normal equation

Now let's look at normal equation:

$$X = \begin{pmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ (x^{(3)})^T \\ \vdots \\ (x^{(m)})^T \end{pmatrix} \in \mathbb{R}^{n+1} \qquad Y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ \vdots \\ y^{(m)} \end{pmatrix}$$

Subsequently, we get the following value of Θ that minimizes the cost function. We perform regularization by adding λ and an identity matrix (with an *(n + 1) × (n + 1)* dimension).

$$\Theta = (X^T X + \lambda \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix})^{-1} X^T y \tag{49}$$

Now we'll describe the issue of non-invertibility. Suppose $m \leq n$. If you have fewer examples than features ($X^T X$) will be non-invertible/singular/degenerate. If you implement this anyway it will partly to the right thing but it will not give you a proper hypothesis. However if $\lambda > 0$, than $(X^T X + \lambda \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix})$ will be invertible.

3.2.4 Solving the Problem of Overfitting - Regularized Logistic Regression

We saw earlier that the cost function of logistic regression can be written as followed:

$$J(\Theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \cdot log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) \cdot log(1 - h_\Theta(x^{(i)}))] \tag{44}$$

If we want to modify this with regularization we only have to add the following term:

$$J(\Theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \cdot log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) \cdot log(1 - h_\Theta(x^{(i)}))] + \frac{\lambda}{2m} \sum_{i=1}^{m} \Theta_j^2 \tag{50}$$

For gradient descent, we do the same thing as we did for linear regression:

repeat until convergence {

$$\begin{aligned} \Theta_0 &:= \Theta_0 - \alpha \frac{1}{m}[\sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y_i) \cdot x_0^{(i)}] \quad \rightarrow \quad \frac{\partial}{\partial \Theta_0} J(\Theta) \\ \Theta_j &:= \Theta_j - \alpha[\frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y_i) \cdot x_j^{(i)} + \frac{\lambda}{m} \Theta_j] \quad \rightarrow \quad \frac{\partial}{\partial \Theta_j} J(\Theta) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (j = 1,2,3,\ldots,n) \end{aligned} \tag{47}$$

}

Now let's look how regularization works when using advanced optimization:

function [jVal, gradient] = costFunction(theta)
  jVal = [code to compute $J(\Theta)$]
  gradient(1) = [code to compute $\frac{\partial}{\partial \Theta_0} J(\Theta)$];
  gradient(2) = [code to compute $\frac{\partial}{\partial \Theta_1} J(\Theta)$];
  ⋮
  gradient(n + 1) = [code to compute $\frac{\partial}{\partial \Theta_n} J(\Theta)$] ;

The $J(\Theta)$ of "jVal" now has been modified with the added term $\frac{\lambda}{2m} \Theta_j^2$. The gradient(1) has not been modified since this computes $\frac{\partial}{\partial \Theta_0} J(\Theta)$, while gradient(2, n + 1) has the additional regularization term $\frac{\lambda}{2m} \Theta_j^2$ since it computes $\frac{\partial}{\partial \Theta_{(2,n+1)}} J(\Theta)$.

4 Week 4

4.1 Neural Networks: Representation

4.1.1 Motivations: Non-linear Hypotheses

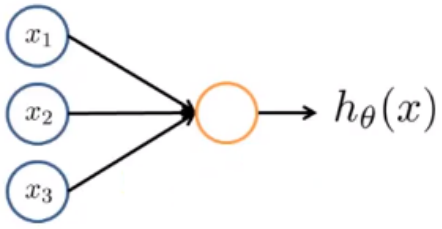
Computer vision includes predicting classes of objects (e.g. cars). An image consists of pixels. With these pixels computers predict and classify certain objects. However, this demands polynomial hypothesis functions. However, computational cost increases tremendously with an increasing pixel amount (exponential). Therefore, this is not an efficient way for classifying objects. A solution for this problem is the use of Neural Networks.

4.1.2 Motivations: Neurons and the Brain

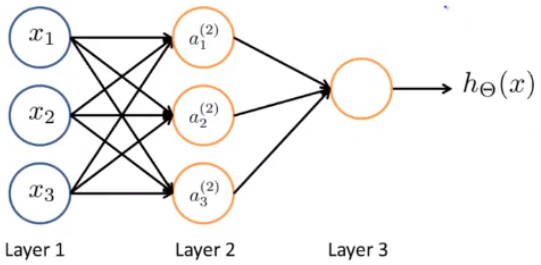
The origins of Neural Networks are algorithms that try to mimic the brain. It was very widely used in the 80s and 90s, however, popularity diminished in late 90s. However, recent resurgence is state-of-the art technique for many applications. The "one learning algorithm" hypothesis refers to the fact that the human brain uses essentially the same algorithm to understand many different input modalities.

4.1.3 Neural Networks - 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 channelled to outputs (axons). An artificial neural network includes models of neurons (logistic unit):



In a neural network, the dendrites are like the input features x_n, and the output is the result of our hypothesis function ($h_\Theta(x)$). 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 ($g(z) = \frac{1}{1+e^{-\Theta x}}$) yet we sometimes call it a sigmoid (logistic) activation function. In this situation, our Θ parameters are sometimes called "weights". 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".



The following parameters are used:

- $a_i^{(j)}$ = "activation" of unit i in layer j
- $\Theta_{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$
- In general, if network has s_j units in layer j, 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 Θ_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. So the calculations are as follows:

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

4.1.4 Neural Networks - Model Representation II

Now we are going to look at a vectorized implementation. We first are going to define some extra terms:

$a_1^{(2)} = g(z_1^{(2)})$
$a_2^{(2)} = g(z_2^{(2)})$
$a_3^{(2)} = g(z_3^{(2)})$

$$x = \begin{pmatrix} x_0 \\ x_1 \\ x_3 \\ x_4 \end{pmatrix} \qquad z^{(2)} = \begin{pmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{pmatrix}$$

$z^{(2)} = \Theta^{(1)} a^{(1)}$
$a^{(2)} = g(z^{(2)})$

Now add $a_0^{(2)} = 1$:
$z^{(3)} = \Theta^{(2)} a^{(2)}$
$h_\Theta = a^{(3)} = g(z^{(3)})$

This process of computing h_Θ of x is also called **forward propagation** and is called this way because we start of with the activations of the input units and then we sort of forward propagate to the hidden layer and compute activations of the hidden layer and then we sort of forward propagate and compute the activations of the output layer. You can have other network architectures with more hidden layers.

4.1.5 Applications - Examples and Intuitions I

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.

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \rightarrow [g(z^{(2)})] \rightarrow h_\Theta(x) \qquad \textit{with} \qquad \Theta^{(1)} = [-30 \quad 20 \quad 20]$$

This will cause the output of our hypothesis to only be positive if both x_1 and x_2 are 1. In other words:

$h_\Theta(x) = g(-30 + 20x_1 + 20x_2)$
$x_1 = 0$ 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 $g(-30) \approx 1$

The OR function, which is the logical 'or' operator and is true if both x_1 or x_2 are 1 could also be computed using a neural network. This could be performed by a network using $\Theta^{(1)} = [-10 \quad 20 \quad 20]$.

4.1.6 Applications - Examples and Intuitions II

See Figure 13.

4.1.7 Applications - Multiclass Classification

We will talk about using neural networks for multiclass classification. This goes as follows (see Figure 14).

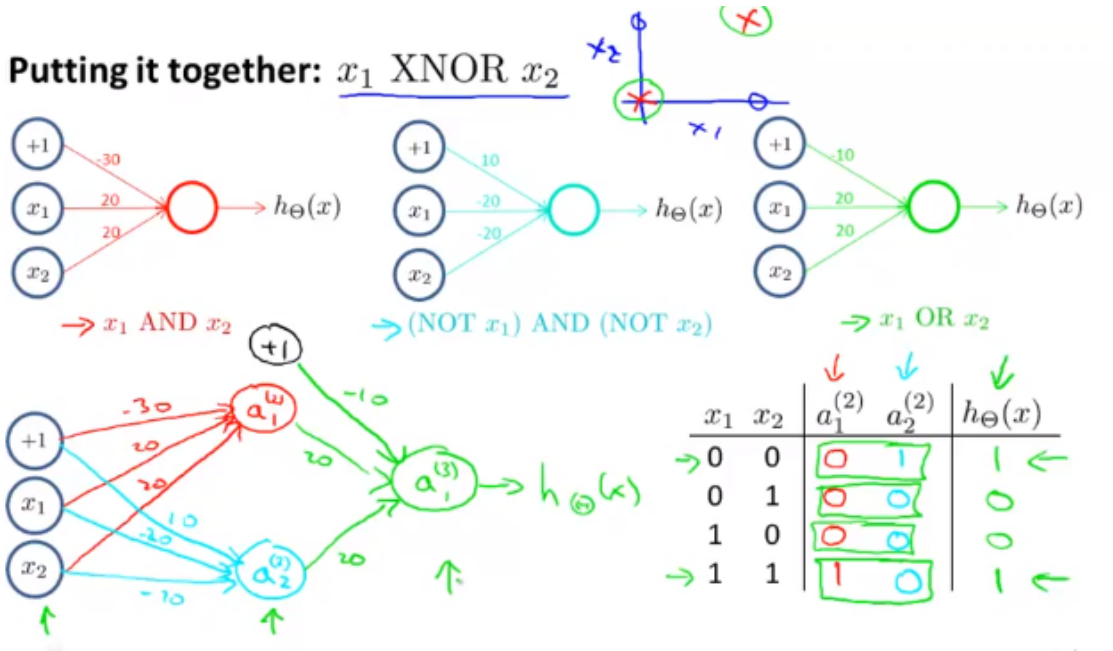


Figure 13: Neural network XNOR

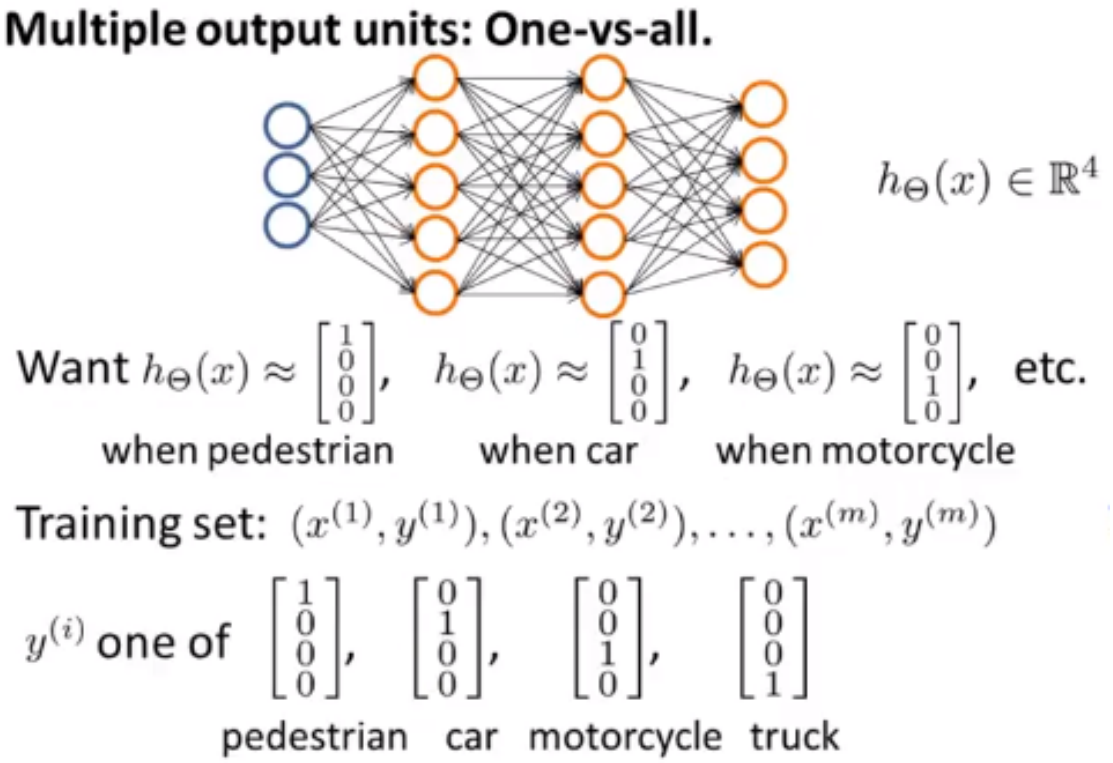


Figure 14: Neural networks for Multiclass Classification

5 Week 5

5.1 Neural Networks: Learning

5.1.1 Cost Function and Backpropagation - Cost Function

Suppose we have a four-layered network (one input layer, two hidden layers and one output layer), where we have the following parameters:

- training set $\{(x^{(1)}, y^{(1)}), x^{(2)}, y^{(2)}), \cdots, x^{(m)}, y^{(m)})\}$
- L = total number of layers in network
- s_l = number of units (not counting bias unit) in layer l
- K = number of output units

We're going to discuss two types of classification problems:

- **Binary classification** where *y = 0 or 1*, where $h_\Theta(x) = \mathbb{R}$
- **Multi-class classification** (K classes) where $y \in \mathbb{R}^K$ and $h_\Theta(x) = \mathbb{R}^K$

The cost function we are going to use is a generalization of the cost function we use for logistic regression:

$$J(\Theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \cdot log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) \cdot log(1 - h_\Theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{i=1}^{m}\Theta_j^2 \tag{50}$$

For a neural network we have more output values for j which leads to the following cost function:

$$h_\Theta(x) \in \mathbb{R}^K \qquad (h_\Theta(x))_i = i^{th}\ output$$
$$J(\Theta) = -\frac{1}{m}[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) log(1 - (h_\Theta(x^{(i)}))_k)] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\Theta_{ji}^{(l)})^2 \tag{51}$$

$\sum_{k=1}^{K}$ sums from 1 through K. This summation is basically a sum over my K output. So if we have four output units, then this is a sum from K equals one through four which is basically the logistic regression algorithm's cost function. However, now we sum the cost function over each of our four output units in turn. So, we have added a few nested summations to account for our multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes. In the regularization part, after the square brackets, we must account for multiple Θ 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.

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

5.1.2 Cost Function and Backpropagation - 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. Here is the cost function that was discussed earlier:

$$h_\Theta(x) \in \mathbb{R}^K \qquad (h_\Theta(x))_i = i^{th}\ output$$
$$J(\Theta) = -\frac{1}{m}[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) log(1 - (h_\Theta(x^{(i)}))_k)] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\Theta_{ji}^{(l)})^2 \tag{51}$$

where we want to:

$$\min_\Theta J(\Theta) \tag{5}$$

and need to compute:

- $J(\Theta)$

- $\frac{\partial}{\partial \Theta_{ij}^{(l)}}$ where $\Theta_{ij}^{(l)} \in \mathbb{R}$

To do so, we use the following algorithm:

Back Propagation Algorithm

Given training set $\{(x^{(1)}, y^{(1)}), x^{(2)}, y^{(2)}), \cdots, x^{(m)}, y^{(m)})\}$

- Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j), used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

For training example $t = 1$ to m:

1. Set $a^{(1)} = x$
2. Perform forward propagation to compute $a^{(1)}$ for $l = 1, 2, 3, \cdots, L$

   $a^{(1)} = x$
   $z^{(2)} = \Theta^{(1)} a^{(1)}$
   $a^{(2)} = g(z^{(2)})$ $(add \quad a_0^{(2)})$
   $z^{(3)} = \Theta^{(2)} a^{(2)}$
   $a^{(3)} = g(z^{(3)})$ $(add \quad a_0^{(3)})$
   $z^{(4)} = \Theta^{(3)} a^{(3)}$
   $a^{(4)} = h_\Theta(x) = g(z(4))$
3. Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$

   Where L is our total number of layers and $a^{(L)}$ is the vector of outputs of the activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y. To get the δ values of the layers before the last layer, we can use an equation that steps us back from right to left:
4. Compute $\delta^{(L-1)}, \delta^{(L-2)}, \cdots, \delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) .* a^{(l)} .* (1 - a^{(l)})$

   The δ values of layer l are calculated by multiplying the δ values in the next layer with the Θ matrix of layer l. We then element-wise multiply that with a function called *g'*, or *g-prime*, which is the derivative of the activation function g evaluated with the input values given by $z^{(l)}$. The *g-prime* derivative terms can also be written out as:

   $g'(z^{(l)}) = a^{(l)} .* (1 - a^{(l)})$
5. $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ Vectorized implementation: $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

   Hence we update our new Δ matrix:

   - $D_{ij}^{(l)} := \frac{1}{m}(\Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)})$ if $j \neq 0$
   - $D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$ if $j = 0$

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

5.1.3 Cost Function and Backpropagation - Backpropagation Intuition

Now what is backpropagation doing? The cost function was as followed:

$$J(\Theta) = -\frac{1}{m}[\sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) log(1 - (h_\Theta(x^{(i)}))_k)] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l+1} (\Theta_{ji}^{(l)})^2 \tag{51}$$

Focusing on a simple example ($x^{(i)}$,$y^{(i)}$), the case of 1 output unit, and ignoring regularization ($\lambda = 0$):
$cost(i) = y^{(i)} log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) log(h_\Theta(x^{(i)}))$ (think of $cost(i) \approx (h_\Theta(x^{(i)}) - y^{(i)})^2$)
I.e. how well is the network doing on example i? See Figure 15, where the following applies: $\delta_j^{(l)}$ = "error" of cost $a_j^{(l)}$ (unit j in layer l). Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(i)$ (for $j \geq 0$), where $cost(i) = y^{(i)} log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) log(h_\Theta(x^{(i)}))$. So backpropagation is calculated as follows: $\delta_1^{(2)} = \Theta_{12}^{(1)} \delta_1^{(3)} + \Theta_{22}^{(1)} \delta_2^{(3)}$. Another example is $\delta_2^{(3)} = \Theta_{12}^{(3)} \delta_1^{(4)}$.

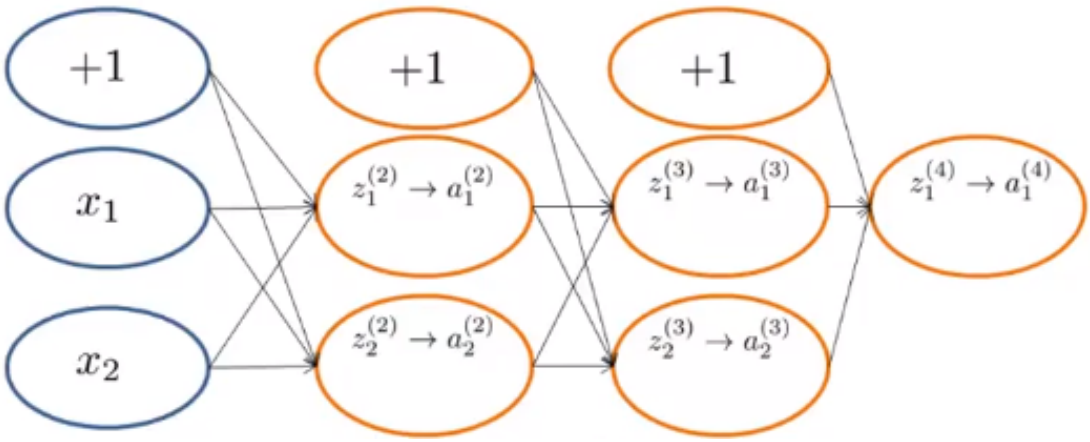


Figure 15: Backpropagation

5.1.4 Backpropagation in Practice - Implementation Note: Unrolling Parameters

With neural networks we are working with sets of matrices:

$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$
$D^{(1)}, D^{(2)}, D^{(3)}$

In order to use optimizing functions such as "fminunc()", we will want to "unroll" all the elements and put them into one long vector:

thetaVector = [Theta1(:); Theta2(:); Theta3(:);]
deltaVector = [D(:); D(:); D(:);]

If the dimensions of Theta1 is 10×11, Theta2 is 10×11 and Theta3 1×11, then we can get back our original matrices from the "unrolled" versions as follows:

Theta1 = reshape(thetaVector(1:110),10,11)
Theta2 = reshape(thetaVector(111:220),10,11)
Theta3 = reshape(thetaVector(221:231),1,11)

So to summarize, we have our initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.
Unroll to get initialTheta to pass to fminunc(@costFunction,initialTheta,options)

function [jval, gradientVec] = costFunction(thetaVec)
From thetaVec, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.
Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$.
Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get gradientVec.

5.1.5 Backpropagation in Practice - Gradient Checking

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

$$\frac{\partial}{\partial \Theta} J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}. \tag{52}$$

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

$$\frac{\partial}{\partial \Theta} J(\Theta) \approx \frac{J(\Theta_1, \cdots, \Theta_j + \epsilon, \cdots, \Theta_n) - J(\Theta_1, \cdots, \Theta_j - \epsilon, \cdots, \Theta_n)}{2\epsilon} \tag{53}$$

A small value for ϵ 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. In MATLAB/Octave we can do this as follows:

```
for i = 1:n
    thetaPlus = theta;
    thetaPlus(i) = thetaPlus(i) + EPSILON;
    thetaMinus = theta;
    thetaMinus(i) = thetaMinus(i) - EPSILON;
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2 * EPSILON);
end;
```

Check that gradApprox $\approx$ DVec (from backprop.). Implementation note:

- Implement backprop to compute DVec (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$)
- Implement numerical gradient check to compute gradApprox
- Make sure they give similar values.
- Turn off gradient checking (computational expensive). Using backprop code for learning

Important:

- Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of cost function), your code will be very slow.

5.1.6 Backpropagation in Practice - Random Initialization

Initial value of Θ: for gradient descent and advanced optimization method, need initial value for Θ. Consider gradient descent where initialTheta = zeros(n,1)? Zero initialization does not always work properly, because after each update, parameters corresponding to inputs going into each of two hidden units are identical. In order to get around this problem is **random initialization**: Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$ (i.e. : $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$):

$Theta1 = rand(10, 11) * (2 * INIT\_EPSILON) - INIT\_EPSILON;$
$Theta2 = rand(1, 11) * (2 * INIT\_EPSILON) - INIT\_EPSILON;$

5.1.7 Backpropagation in Practice - Putting It Together

Training a neural network

Pick a network architecture (connectivity pattern between neurons):

- Number of input units: Dimension of features x^i
- Number of output units: Number of classes
- Reasonable default: 1 hidden layer, or if > 1 hidden layer, have some number of hidden units in every layer (usually the more the better)(e.g. 3 - 5 - 4, 3 - 5 - 5 -4, 3 - 5 - 5 - 5 - 4).

Then:

1. Randomly initialize weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement code to compute cost function $J(\Theta)$
4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

   for i = 1:m
   Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$
   (Get activations $a^{(l)}$ and delta terms $d^{(l)}$ for $l = 2, \cdots, L$).
5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backpropagation vs. using numerical estimate of gradient descent of $J(\Theta)$. The disable gradient checking code.
6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ.

5.1.8 Application of Neural Networks - Autonomous Driving

Example of an autonomous driving car (Dean Pomerleau)[2].

[2] https://www.youtube.com/watch?v=ilP4aPDTBPE

6 Week 6

6.1 Advice for Applying Machine Learning

6.1.1 Evaluating a Learning Algorithm - Deciding What to Try Next

Suppose you have implemented regularized linear regression to predict housing prices:

$$J(\Theta) = \frac{1}{2m}[\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 + \lambda\sum_{i=1}^{m}\Theta_j] \tag{46}$$

However, when you test your hypothesis on a new set of houses, you find that it makes unacceptably large errors in its predictions. What should you try next?

- Get more training examples
- Try smaller sets of features
- Try getting additional features
- Try adding polynomial features
- Try decreasing/increasing λ

A pretty simple technique with which you can rule out half of the things on the list is **machine learning diagnostic**. This includes a test that you can run to gain insight what is/isn't working with a learning logarithm, and gain guidance as to how best to improve its performance. Diagnostics can take time to implement, but doing so can be a very good use of your time.

6.1.2 Evaluating a Learning Algorithm - Evaluating a Hypothesis

An overfitting hypothesis fails to generalize to new examples that are not in the training set. The standard way to evaluate a hypothesis is as follows. We are going to split the data in two portions; one **randomly** sorted portion is going to be our training set (70%) and the second **randomly** portion is going to be our test set (30%). The procedure for **linear regression** is as follows:

- Learn parameter Θ from training data (minimizing training error $J(\Theta)$)
- Compute test set error:

$$J_{test}(\Theta) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}(h_\Theta(x_{test}^{(i)}) - y_{test}^{(i)})^2 \tag{54}$$

The procedure for **logistic regression** is as follows:

- Learn parameter Θ from training data
- Compute test set error:

$$J_{test}(\Theta) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}} y_{test}^{(i)} log h_\Theta(x_{test}^{(i)}) + (1 - y_{test}^{(i)}) log h_\Theta(x_{test}^{(i)}) \tag{55}$$

- Misclassification error (0/1 misclassification error):

$$err(h_\Theta(x), y) = \begin{cases} 1 & \text{if } h_\Theta(x) \geq 0.5, \quad y = 0 \\ & \text{or if } h_\Theta(x) < 0.5, \quad y = 1 \\ 0 & \text{otherwise} \end{cases} \tag{56}$$

$$\text{Test error } = \frac{1}{m_{test}}\sum_{i=1}^{m_{test}} err(h_\Theta(x_{test}^{(i)}), y^{(i)}) \tag{57}$$

6.1.3 Evaluating a Learning Algorithm - 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 overfit 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. In general, once parameters $\Theta_0, \Theta_1, \cdots, \Theta_4$ were fit to some set of data (training set), the error of the parameters as measured on that data (the training error $J(\Theta)$) is likely to be lower than the actual generalization error. One model selection problem is the degree of polynomial (d) you choose:

$$
\begin{aligned}
&h_\Theta(x) = \Theta_0 + \Theta_1 x \rightarrow \Theta^{(1)} \rightarrow J_{test}(\Theta^{(1)}) \\
&h_\Theta(x) = \Theta_0 + \Theta_1 x + \Theta_2 x^2 \rightarrow \Theta^{(2)} \rightarrow J_{test}(\Theta^{(2)}) \\
&h_\Theta(x) = \Theta_0 + \Theta_1 x + \cdots + \Theta_3 x^3 \rightarrow \Theta^{(3)} \rightarrow J_{test}(\Theta^{(3)}) \\
&\vdots \\
&h_\Theta(x) = \Theta_0 + \Theta_1 x + \cdots + \Theta_{10} x^{10} \rightarrow \Theta^{(10)} \rightarrow J_{test}(\Theta^{(10)})
\end{aligned}
$$

Now in order to select a model, you can see which model has the lowest test error ($J_{test}(\Theta)$). However, the problem is that the error of the test set is likely to be an optimistic estimate of generalization error. In other words, our extra parameter (d = degree of polynomial) is fit to the test set. So, what we usually do instead is split the out data set in three pieces:

- The training set denoted as $(x^{(i)}, y^{(i)})(60\%)$
- The cross validation (CV) set denoted as $(x_{cv}^{(i)}, y_{cv}^{(i)})$ (20%)
- The test set denoted as $(x_{test}^{(i)}, y_{test}^{(i)})$ (20%)

Now we can also set the train/validation/test error:

$$\textit{Training error:} \qquad J_{train}(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y^{(i)})^2 \tag{58}$$

$$\textit{Cross Validation error:} \qquad J_{cv}(\Theta) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_\Theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2 \tag{59}$$

$$\textit{Test error:} \qquad J_{test}(\Theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_\Theta(x_{test}^{(i)}) - y_{test}^{(i)})^2 \tag{60}$$

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. So when faced with a model selection problem like this, we are going to use the cross validation set instead of the test error. Concretely, we are going to take our first hypothesis/model and minimizing the error giving us a specific error for Θ to let's say, the 10^{th} polynomial. Now we are going to test each of these hypothesis on the validation set. Then we are going to take the hypothesis with the lowest cross validation error.

$$
\begin{aligned}
&h_\Theta(x) = \Theta_0 + \Theta_1 x \rightarrow \Theta^{(1)} \rightarrow J_{cv}(\Theta^{(1)}) \\
&h_\Theta(x) = \Theta_0 + \Theta_1 x + \Theta_2 x^2 \rightarrow \Theta^{(2)} \rightarrow J_{cv}(\Theta^{(2)}) \\
&h_\Theta(x) = \Theta_0 + \Theta_1 x + \cdots + \Theta_3 x^3 \rightarrow \Theta^{(3)} \rightarrow J_{cv}(\Theta^{(3)}) \\
&\vdots \\
&h_\Theta(x) = \Theta_0 + \Theta_1 x + \cdots + \Theta_{10} x^{10} \rightarrow \Theta^{(10)} \rightarrow J_{cv}(\Theta^{(10)})
\end{aligned}
$$

In summary, we can now calculate three separate error values for the three different sets using the following method and the training/cross validation/test set:

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

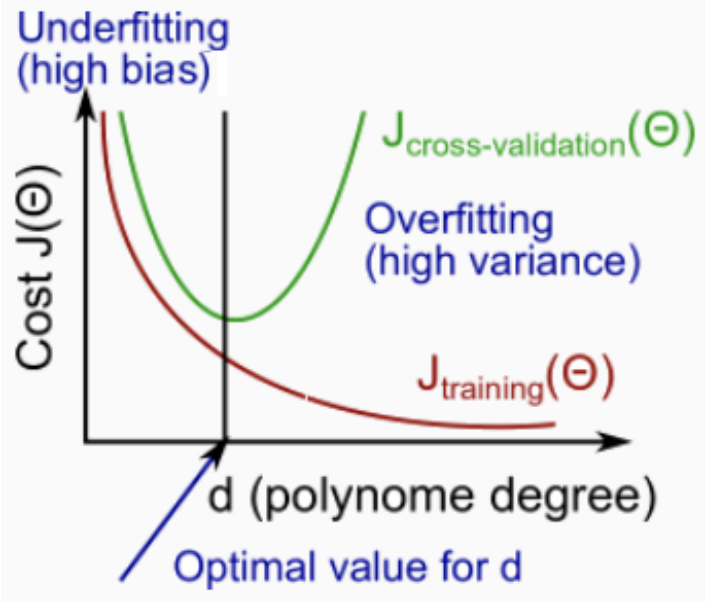


Figure 16: $J_{train}(\Theta)$ and $J_{cv}(\Theta)$ with an increasing d

6.1.4 Bias vs. Variance - Diagnosing Bias vs. Variance

When you have a hypothesis function that's not performing well you usually have the problem of high bias (underfit, d is 1) and high variance (overfit d is 4). But how do you determine which one this is? Let's look again at the training/cross validation error:

$$\textit{Training error:} \qquad J_{train}(\Theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})^2 \tag{58}$$

$$\textit{Cross Validation error:} \qquad J_{cv}(\Theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\Theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2 \tag{59}$$

The $j_{train}(\Theta)$ tends to decrease with an increasing polynomial. For $J_{cv/test}(\Theta)$ decreases to a certain value of d but increases again due to overfitting (see Figure 16). So, suppose your learning algorithm is performing less well than you were hoping ($J_{cv}(\Theta)$ or $J_{test}(\Theta)$ is high). Is it a bias problem or a variance problem? The regime on the left corresponds to a bias problem, whereas the regime on the right corresponds to a variance problem:

- Bias (underfit): Both $J_{train}(\Theta)$ and $J_{cv}(\Theta)$ will be high.
- Variance (overfit): $J_{train}(\Theta)$ will be low and $J_{cv}(\Theta)$ will be high.

6.1.5 Bias vs. Variance - Regularization and Bias/Variance

Now we're going to go deeper into the issue of bias and variances and talk about how it interacts with and is affected by the regularization of your learning algorithm:

$$J(\Theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{i})^2 + \lambda\sum_{j=1}^{m}\Theta_j^2 \tag{46}$$

A large λ gives a high bias (underfit) while a low value of λ gives us a high variance (overfit). Now how to choose the regularization parameter? For this we will redefine $J_{train}(\Theta)$ which was earlier defined as (without regularization term):

$$J_{train}(\Theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})^2 \tag{58}$$

$$J_{cv}(\Theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\Theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2 \tag{59}$$

$$J_{test}(\Theta) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}(h_\Theta(x_{test}^{(i)}) - y_{test}^{(i)})^2 \tag{60}$$

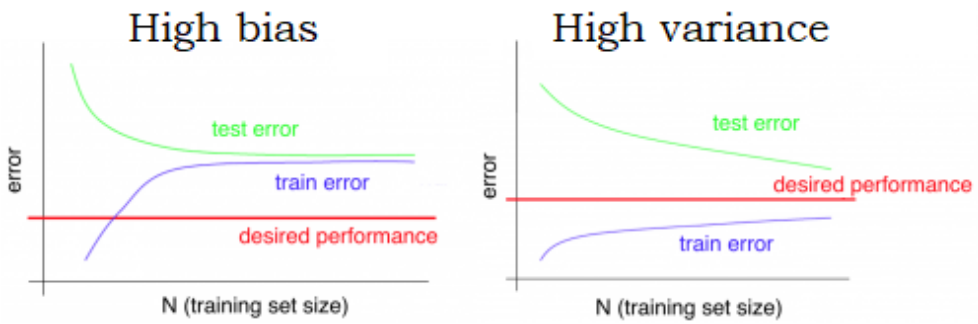


Figure 17: Learning Curve

Now we are going to choose different values for parameter λ:

$$\text{Try } \lambda = 0 \rightarrow \min_{\Theta} J(\Theta) \rightarrow \Theta^{(1)} \rightarrow J_{cv}(\Theta^{(1)})$$
$$\text{Try } \lambda = 0.01 \rightarrow \min_{\Theta} J(\Theta) \rightarrow \Theta^{(2)} \rightarrow J_{cv}(\Theta^{(2)})$$
$$\text{Try } \lambda = 0.02 \rightarrow \min_{\Theta} J(\Theta) \rightarrow \Theta^{(3)} \rightarrow J_{cv}(\Theta^{(3)})$$
$$\text{Try } \lambda = 0.04 \rightarrow \min_{\Theta} J(\Theta) \rightarrow \Theta^{(4)} \rightarrow J_{cv}(\Theta^{(4)})$$
$$\text{Try } \lambda = 0.08 \rightarrow \min_{\Theta} J(\Theta) \rightarrow \Theta^{(5)} \rightarrow J_{cv}(\Theta^{(5)})$$
$$\vdots$$
$$\text{Try } \lambda = 10 \rightarrow \min_{\Theta} J(\Theta) \rightarrow \Theta^{(10)} \rightarrow J_{cv}(\Theta^{(10)})$$

You choose then the value for λ which gives the lowers cross validation error $J_{cv}(\Theta^{(i)})$ and see how well it does on the test set (low $J_{test}(\Theta^{(i)})$). $J_{train}(\Theta)$ tends to be low with a small value of λ and increases with an increasing value of λ. Moreover, $J_{train}(\Theta)$ will be high with a low and high value of λ. So in order to choose the model and the regularization term λ, we need to:

- Create a list of values for λ
- Create a set of models with different degrees or any other variants
- Iterate through the λ's and for each λ go through all the models to learn some Θ
- Compute the cross validation error using the learned Θ (computed with λ) on the $J_{cv}(\Theta)$ **without** regularization or $\lambda = 0$
- Select the best combo that produces the lowest error on the cross validation set.
- Using the best combo Θ and λ, apply it on $J_{test}(\Theta)$ to see if it has a good generalization of the problem.

6.1.6 Bias vs. Variance - Learning Curves

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

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

Experiencing high bias (see Figure 17)

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

Large training 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.

Experiencing high variance (see Figure 17)

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

Large training size: causes $J_{train}(\Theta)$ to increase with training size and $J_{cv}(\Theta)$ continues to decrease without levelling 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.

6.1.7 Bias vs. Variance - Deciding What to Do Next Revisited

Debugging a learning algorithm: Suppose you have implemented regularized linear regression to predict housing prices. However, when you test your hypothesis in a new set of houses, you find that it makes unacceptably large errors in its prediction. What should you try next?

- Get more training examples → fixes high variance
- Try smaller sets of features → fixes high variance
- Try getting additional features → fixes high bias
- Try adding polynomial features → fixes high bias
- Try decreasing λ → fixes high bias
- Try increasing λ → fixes high variance

Let's relate this back to neural networks. In a "small" neural network you have fewer parameters (more prone to underfitting), but is computationally cheaper. In "large" neural networks you have more parameters (more prone to overfitting) and it computationally more expensive (use regularization λ do address overfitting). Another factor is the number of hidden layers; this you can try out with a training/cross validation/test set. Here are a few model complexity effects:

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

6.2 Machine Learning System Design

6.2.1 Building a Spam Classifier - Prioritizing What to Work On

We take as an example building a spam classifier, which classifies whether e-mails are spam (1) or not (0). In order to apply supervised learning we have to define x = features of email and y = spam (1) or not spam (0). We can than train a classifier for instance with a linear regression algorithm. So we could come up with some features x (e.g. 100 words) that indicate for spam/not spam. Given a piece of e-mail we can code it into a feature vector X,where a 0 corresponds to the absence and a 1 corresponds to the presence of a "spam" word. The length of the vector corresponds to the number of features (e.g. 100 words). Note: in practice, take most frequently occurring n words (10,000 to 50,000) in training set, rather than manually pick 100 words. Now when you building a spam classifier, how to spend your time to make it have a low error?

- Collect lots of data
  - E.g. "honeypot" project
- Develop sophisticated features based on email routing information (from email header)
- Develop sophisticated features for message body, e.g. should "discount" and "discounts" be treated as the same word? How about "deal" and "dealer"? Features about punctuation?
- Develop sophisticate algorithm to detect misspellings (e.g. m0rtage, med1cine, w4tches).

It is difficult to find out which of these options gives the best result!

6.2.2 Building a Spam Classifier - Error Analysis

The recommended approach is:

- Start with a simple algorithm that you can implement quickly. Implement it and test it on your cross validation data.
- Plot learning curves to decide if more data, more features, etc. are likely to help.
- Error analysis: Manually examine the examples (in cross validation set) that your algorithm made errors on. See if you spot any systematic trend in what type of examples it is making errors on.

The reason that this is a good approach is because it is hard to tell in advance what is the right thing to do and where you should spend your time. Premature optimization is based on "gut feeling", which is not considered a good approach. So for instance, look at the spam and not spam emails that the algorithm is misclassifying also known as **error analysis**. So let's look at the following example, where you have $m_{cv} = 500$ examples in your cross validation set and where the algorithm misclassified 100 emails. We could manually analyse 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.

1. What type of email it is (e.g. pharmaceutics, replica/fake, steal passwords).
2. What cues (features) you think would have helped the algorithm classify them correctly (e.g. deliberate misspellings, unusual email routing, unusual (spamming) punctuation).

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: 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.

6.2.3 Handling Skewed Data - Error Metrics for Skewed Classes

A challenge which machine learning practitioners often face, is how to deal with skewed classes in classification problems. Such a tricky situation occurs when one class is over-represented in the data set. A common example for this issue is fraud detection: a very big part of the data set, usually 90%, describes normal activities and only a small fraction of the data records should get classified as fraud. In such a case, if the model always predicts "normal", then it is correct 90% of the time. At first, the accuracy and therewith the quality of such a model might seem surprisingly good, but as soon as you perform some analysis and dig a little bit deeper, it becomes obvious that the model is completely useless. Now consider the example of the cancer classification example, where we train a logistic regression model $h_\Theta(x)$ ($y = 1$ if cancer, $y = 0$ otherwise). After running the algorithm you find out that you got 1% error on the test set (99% correct diagnoses). However, only 0.50% of patients have cancer.

```
function y = predictCancer(x)
    y = 0; % ignore x! → error = 0.5%
return
```

So this setting of when the ratio of positive to negative examples is very close to one of two extremes, where, in this case, the number of positive examples is much, much smaller than the number of negative examples because y equals one so rarely, this is what we call the case of skewed classes. If you have very skewed classes it becomes much harder to use just classification accuracy, because you can get very high classification accuracies or very low errors, and it's not always clear if doing so is really improving the quality of your classifier because predicting y equals 0 all the time doesn't seem like a particularly good classifier. When we're faced with such a skewed classes therefore we would want to come up with a different error metric or a different evaluation metric. One such evaluation metric are what's called precision recall (see Figure 18):

- **Precision**: of all patients where we predicted $y = 1$, what fraction actually has cancer?
  $\frac{True\ positives}{\#\ predicted\ positives} = \frac{True\ positives}{True\ positives + False\ positives}$
- **Recall**: of all patients that actually have cancer, what fraction did we correctly detect as having cancer?
  $\frac{True\ positives}{\#actual\ positives} = \frac{True\ positives}{True\ positives + False\ negatives}$

		Actual outcome y	
		Positive	Negative
Predicted outcome $h_\theta(x)$	Positve	True Positive	False positive
	Negative	False negative	True negative

Figure 18: Predicted and actual outcomes

So, if we have a learning algorithm that predicts $y = 0$ all the time, then this classifier will have a recall equal to 0, because there won't be any true positives. For settings where we have very skewed classes, it's not possible for an algorithm to sort of "cheat" and somehow get a very high precision and a very high recall by doing some simple thing like constantly predicting $y = 0$ or $y = 1$. And so we're much more sure that a classifier of a high precision or high recall actually is a good classifier, and this gives us a more useful evaluation metric that is a more direct way to actually understand whether the algorithm may be doing well. One final note in the definition of precision and recall, is that we usually use the convention that $y = 1$, in the presence of the more rare class.

6.2.4 Handling Skewed Data - Trading Off Precision and Recall

As a recap:

$$\text{Precision} = \frac{\textit{True positives}}{\textit{\# predicted positives}} = \frac{\textit{True positives}}{\textit{True positives + False positives}}$$

$$\text{Recall} = \frac{\textit{True positives}}{\textit{\#actual positives}} = \frac{\textit{True positives}}{\textit{True positives + False negatives}}$$

Trading off precision and recall
Logistic regression: $0 \geq h_\Theta(x) \geq 1$

Predict 1 if $h_\Theta(x) \geq 0.5$
Predict 0 if $h_\Theta(x) < 0.5$

Suppose we want to predict $y = 1$ (cancer) only if very confident. Then we change the predictions as follows:

Predict 1 if $h_\Theta(x) \geq 0.7$
Predict 0 if $h_\Theta(x) < 0.7$

This gives a **higher precision** (a higher fraction of the cancers that you predicts of having cancer actual have cancer). In contrast, this classifier will have **lower recall**. Now, suppose we want to avoid missing too many cases of cancer (avoid false negatives), we do the following:

Predict 1 if $h_\Theta(x) \geq 0.3$
Predict 0 if $h_\Theta(x) < 0.3$

In this case, there will be a **higher recall** classifier (we are going to correctly flagging a larger portion of the patients that actually do have cancer). However, there will be a **lower precision**.

More generally: Predict 1 if $h_\Theta \geq \textit{threshold}$ (see Figure 19). The form of this curve can vary. Now the question remains, how to compare precision/recall numbers? We now have two row numbers, namely precision (P) and recall (R) which makes it more complicated. So how can we get a single row number evaluation matrix? We could look at the average of both the precision and the recall number ($\frac{P+R}{2}$). However, this is not a good solution, because the two extremes will not be visible. A different way is calculating the F_1 score: $2\frac{PR}{P+R}$. This score takes a product of both the precision and recall, so this product must be pretty large. When either the precision or recall is 0, the F_1 will be really low as well:

- When $P = 0$ or $R = 0$ than the F_1-score will be 0.
- When $P = 1$ and $R = 1$ than the F_1-score will be 1.

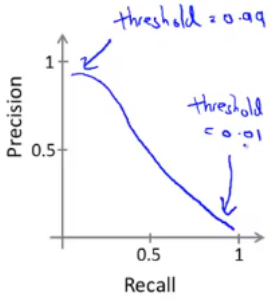


Figure 19: Precision and recall curve

6.2.5 Using Large Data Sets - Data For machine Learning

Another important aspect of machine learning is how much data us use to train your model on. Years ago Banko and Brill (2001)[3] designed a high accuracy learning system (e.g. classify between confusable words). They choose the following algorithms: Perceptron (Logistic regression), Winnow, Memory-based and Naive Bayes. They saw that for all algorithms accuracy increased with the training set size (in millions). Since this initial study, other studies showed similar results. Namely, that the type of algorithm has lesser effect with increasing data. So let's look at the following **large data rationale**: Assume feature $x \in \mathbb{R}^{n+1}$ has sufficient information to predict y accurately. Say for the following example "For breakfast I ate $\cdots$ eggs.", y can very well be predicted. however, for the following example, predicting y becomes much more difficult: "Predict housing price from only size (feet^2) and no other features". A useful test can be: Given the input x, can a human expert confidently predict y? Say, let's use a learning algorithm with many parameters (e.g. logistic regression/linear regression with many features; neural network with many hidden units) (low bias algorithms). The chances are that $J_{train}(\Theta)$ will be small. Now let's say with use a very large training set (unlikely to overfit). Then $J_{train}(\Theta) \approx J_{train}(\Theta)$. Now what these two imply is that $J_{test}(\Theta)$ will be small. So for a good learning algorithm we want to have a low bias (many parameters) and low variance (large training set).

[3] https://cdn-images-1.medium.com/max/800/0*3XNoR1xe4UOkwWzC.png

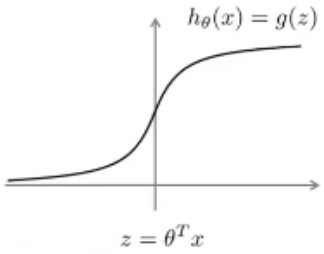


Figure 20: Sigmoid Function/Logistic Function. The function *g(z)* is shown here, which maps any real number to the *(0,1)* interval. z is defined as $\Theta^T x$.

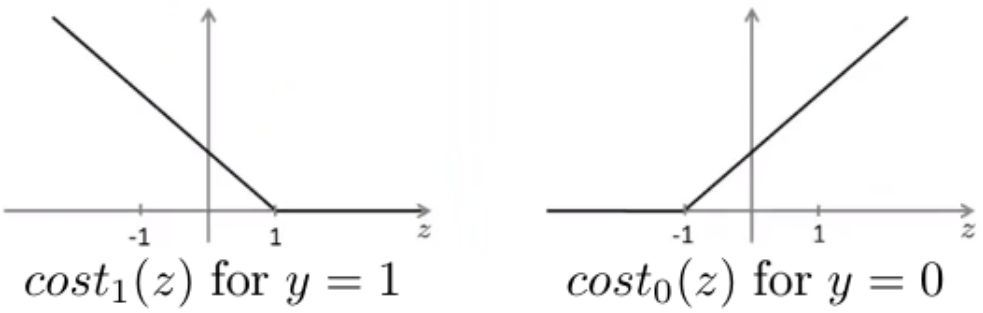


Figure 21: Support Vector Machines cost function.

7 Week 7

7.1 Support Vector Machines

7.1.1 Large Margin Classification - Optimization Objective

Support vector machines (SVMs) are supervised learning models with associated learning algorithms that analyse data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a **non-probabilistic binary linear classifier**. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall. In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the **kernel trick**, implicitly mapping their inputs into high-dimensional feature spaces. We'll come back to this later. We'll first focus on linear logistic regression (classification). An alternative view of logistic regression, with $h_\Theta(x) = \frac{1}{1+e^{-\Theta^T x}}$ is as follows (see Figure 20): We going to use z to denote $\Theta^T x$. If $y = 1$, we want $h_\Theta(x) \approx 1$, $\Theta^T x \gg 0$. In contrast, if $y = 0$, we want $h_\Theta(x) \approx 0$, $\Theta^T x \ll 0$. Now, when we look at the cost of example (x, y): $-(\ y\ log\ h_\Theta(x) + (1-y)\ log\ (1 - h_\Theta(x)))$

$= -y\ log\ \frac{1}{1+e^{-\Theta^T x}} - (1-y)\ log\ (1 - \frac{1}{1+e^{-\Theta^T x}})$

If $y = 1$ (want $\Theta^T x \gg 0$) only the first term is used ($-\ log\ \frac{1}{1+e^{-\Theta^T x}}$). So when z is fairly large, the contribution to the cost function will be small. Now we are going to modify this term slightly. The new cost function will be flat and grows in a straight line when z becomes smaller (two line segments) when $y = 1$. This will give us computational advantages that will be explained later. Now, when $y = 0$ (want $\Theta^T x \ll 0$) only the second term will apply ($-\ log\ (1 - \frac{1}{1+e^{-\Theta^T x}})$). Now for the support vector machine, we again split the function in two segments (first flat and then a linear line). The cost functions will be named the following: $cost_1(z)$ for $y = 1$ and $cost_0(z)$ for $y = 0$ (see Figure 21). Now the **support vector machine** will look as follows for logistic regression:

$$\min_\Theta \frac{1}{m}[\sum_{i=1}^{m} y^{(i)}(-\log h_\Theta(x^{(i)})) + (1-y^{(i)})((-\ log\ (1 - h_\Theta(x^{(i)})))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\Theta_j^2 \tag{50}$$

Support vector machine:

$$\min_{\Theta} \frac{1}{m} \sum_{i=1}^{m} y^{(i)} cost_1(\Theta^T x^{(i)}) + (1 - y^{(i)})\ cost_0(\Theta^T x^{(i)}) + \frac{\lambda}{2m} \sum_{j=1}^{n} \Theta_j^2 \tag{61}$$

Now we are going to write this a bit differently:

$$\min_{\Theta} C \sum_{i=1}^{m} y^{(i)} cost_1(\Theta^T x^{(i)}) + (1 - y^{(i)})\ cost_0(\Theta^T x^{(i)}) + \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2 \tag{62}$$

Remove $\frac{1}{m}$, because this is a constant and does not change the value of the minimization function ($J(\Theta)$). Earlier we wanted to minimize $A + \lambda B$. So what we did was by setting different values for this regularization parameter λ, we could trade off the relative weight between how much we wanted the training set well, that is, minimizing A, versus how much we care about keeping the values of the parameter small, so that will be, the parameter is B. So instead of using λ here to control the relative waiting between the first and second terms, we're instead going to use a different parameter which by convention is called C and is set to minimize $CA + B$. So for logistic regression, if we set a very large value of λ, that means you will give B a very high weight. In contrast, if we set C to be a very small value, then that responds to giving B a much larger rate than C, than A. So this is just a different way of controlling the trade off, it's just a different way of prioritizing how much we care about optimizing the first term, versus how much we care about optimizing the second term. So, this altered equation for the cost function will give the same value of Θ (i.e. the same value of Θ gives the optimal solution to both problems) if $C = \frac{1}{lambda}$. The SVM hypothesis is as follows:

$$h_\Theta(x) = \begin{cases} 1 & if\ \Theta^T x \geq 0 \\ 0 & otherwise \end{cases} \tag{63}$$

7.1.2 Large Margin Classification - Large Margin Intuition

Sometime people associate support vector machines with large margin classifiers. The cost function was defined as follows:

$$\min_{\Theta} C \sum_{i=1}^{m} y^{(i)} cost_1(\Theta^T x^{(i)}) + (1 - y^{(i)})\ cost_0(\Theta^T x^{(i)}) + \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2 \tag{62}$$

The cost functions were plotted earlier (see Figure 21). Now what does it take to make this cost functions small (approaching the flat line)?

- If $y = 1$, we want $\Theta^T x \geq 1$ (not just ≥ 0)
- If $y = 0$, we want $\Theta^T x \leq -1$ (not just < 0)

This is an interesting property of the SVM, because it demands that $\Theta^T x$ is extreme, which serves as a safety margin into the SVM. Logistic regression does something similar too of course, but let's see what the consequences are, in the context of the support vector machine. As an example, we set C to a very large value. Now we are going to be highly motivated to set the first term close to 0. For the fist term to approach 0, we have to:

- Whenever $y^{(i)} = 1$, then $\Theta^T x^{(i)} \geq 1$, so that $cost_1(z) \approx 0$
- Whenever $y^{(i)} = 0$, then $\Theta^T x^{(i)} \geq -1$, so that $cost_0(z) \approx 0$

If we've accomplished this, the first term is equal to 0: $\min_{\Theta} C \times 0 + \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2$. When you solve this, you get a very interesting decision boundary. When you have a linearly separable case of a SVM decision boundary, the decision boundary segregated the examples with a larger minimum distance from any of the training examples. This distance is called the margin of the SVM and this margin gives the SVM a certain robustness, because it tries to separate the data with as a large a margin as possible. That's why the SVM is sometimes called a large margin classifier. If C is very large, it will take account for outliers, when C is not too large, outliers will not be accounted for.

7.1.3 Large Margin Classification - Mathematics Behind Large Margin Classification

Vector Inner Product Let's say you want to calculate the inner product of two vectors, namely $u = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$ and $v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$ (see Figure 22, left). Then you could say that the inner product is $u^T v = p \cdot ||u|| = u_1 v_1 + u_2 v_2$, where

- p = length of the projection of v onto u (if angle between v and u is more than 90°, p will be a negative value).
- $||u|| =$ *length of vector* u $= \sqrt{u_1^2 + u_2^2}$ ($p \in \mathbb{R}$)

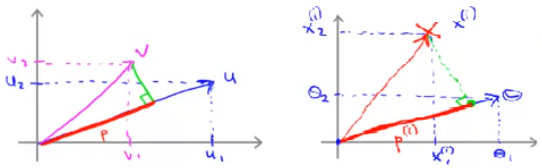


Figure 22: Inner product of vector v and vector u.

Now we go back to the SVM decision boundary where

$$\min_{\Theta} \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2$$

with

- $\Theta^T x^{(i)} \geq 1$ if $y^{(i)} = 1$
- $\Theta^T x^{(i)} \leq -1$ if $y^{(i)} = 0$

We are going to simplify this by ignoring Θ_0 with $n = 2$. Now the cost function can be written as follows:

$$\min_{\Theta} \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2 = \frac{1}{2}(\Theta_1^2 + \Theta_2^2 = \frac{1}{2}(\sqrt{\Theta_1^2 + \Theta_2^2})^2 \tag{64}$$

The term inside the parenthesis $\sqrt{\Theta_1^2 + \Theta_2^2}$ is equal to $||\Theta||$ (the length of the vector Θ). Finally this means that:

$$\min_{\Theta} \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2 = \frac{1}{2}(\Theta_1^2 + \Theta_2^2 = \frac{1}{2}(\sqrt{\Theta_1^2 + \Theta_2^2})^2 = \frac{1}{2}||\Theta||^2 \tag{64}$$

Now we are going to set $\Theta^T x^{(i)}$ to $u^T v$ (see Figure 22, right). Now $\Theta^T x^{(i)} = p^{(i)} \cdot ||\Theta|| = \Theta_1 x_1^{(i)} + \Theta_2 x_2^{(i)}$. Writing this down in our optimization objective is:

$$\min_{\Theta} \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2 = \frac{1}{2}||\Theta||^2$$

with

- $p^{(i)} \cdot ||\Theta|| \geq 1$ if $y^{(i)} = 1$
- $p^{(i)} \cdot ||\Theta|| \leq -1$ if $y^{(i)} = 0$

where $p^{(i)}$ is the projection of $x^{(i)}$ onto the vector Θ with the simplification: $\Theta_0 = 0$ (the decision boundary has to pass the origin $0, 0$). If p will be small, $||\Theta||$ should be very large. On the other hand, when p has a large value, $||\Theta||$ can be smaller, which is in line with the the optimization objective where we try to find values of Θ that are small. The values of p are corresponding to the margin, have a maximum value in SVM. If you allow $\Theta_0 \neq 0$, the decision boundary does not passes the origin. However, the VSM works similar as when it does cross the origin.

7.1.4 Large Kernels - Kernels I

Now we are going to discuss adapting SVMs for developing complex non-linear classifiers. One way to do this is adding polynomial features, where:

$$h_\Theta(x) = \begin{cases} 1 & \textit{if } \Theta_0 + \Theta_1 x_1 + \cdots \geq 0 \\ 0 & \textit{otherwise} \end{cases}$$

A alternative way to denote this is $\Theta_0 + \Theta_1 f_1 + \Theta_2 f_2 + \Theta_3 f_3 + \cdots$. However, is there a different/better choice of the features $f_1, f_2, f_3, \cdots$?

Kernel

Given x, compute new feature depending on proximity to landmarks $l^{(1)}, l^{(2)}, l^{(3)}$ (picked manually). Now we are going to define our features as follows given x:

- $f_1 = \text{similarity } (x, l^{(1)}) = \exp(-\frac{||x - l^{(1)}||^2}{2\sigma^2})$

- $f_2 = \text{similarity } (x, l^{(2)}) = \exp(-\frac{||x-l^{(2)}||^2}{2\sigma^2})$
- $f_3 = \text{similarity } (x, l^{(3)}) = \exp(-\frac{||x-l^{(3)}||^2}{2\sigma^2})$

This similarity function is considered a **Gaussian Kernel function**. Sometimes we also write $k(x, l^{(i)})$ (instead of $similarity(x, l^{(i)})$). Now let's look a bit closer.

$$f_1 = \text{similarity } (x, l^{(1)}) = \exp(-\frac{||x-l^{(1)}||^2}{2\sigma^2}) = \exp(-\frac{\sum_{j=1}^{n}(x-l_j^{(1)})^2}{2\sigma^2})$$

We now have the following cases:

- If $x \approx l^{(1)}$: $||x - l^{(1)}||^2$ will be close to 0: $f_1 \approx \exp(-\frac{0^2}{2\sigma^2}) \approx 1$
- If x is far from $l^{(1)}$: $||x - l^{(1)}||^2$ will have a large value: $f_1 \approx \exp(-\frac{(largenumber)^2}{2\sigma^2}) \approx 0$

Each landmark defines a new feature ($l^{(1)} \rightarrow f_1, l^{(2)} \rightarrow f_2, l^{(3)} \rightarrow f_3$). Now let's look a bit closer to the exponential function. So the f_1 calculates how close x is to the landmark l. When σ is set to a smaller value, than the feature f_1 falls to 0 much more rapidly. On the other hand, when σ is set to a larger value, than the feature f_1 falls to 0 much more slowly. So for $x^{(i)}$ you calculate $f_1, f_2, f_3, \cdots$ where eventually you predict y where $y = 1$ if $\Theta_0 + \Theta_1 f_1 + \Theta_2 f_2 + \Theta_1 f_1 \geq 0$ and $y = 0$ if $\Theta_0 + \Theta_1 f_1 + \Theta_2 f_2 + \Theta_1 f_1 < 0$. So this is one way to use Kernels where we predict 1 if we are close to the landmarks and 0 if we are far away from the landmarks.

7.1.5 Large Kernels - Kernels II

Where do we get the landmarks from ($l^{(1)}, l^{(2)}, l^{(3)}$)? For every training example that we have we are going to define landmarks at the same locations as the training examples ($l^{(1)} = x^{(1)}, l^{(2)} = x^{(2)}, l^{(3)} = x^{(3)}$). So, in the end we have n landmarks (equal to the number of training examples). To write this out more concretely:

- Given $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(n)}, y^{(n)})$
- choose $l^{(1)} = x^{(1)}, l^{(2)} = x^{(2)}, \cdots, l^{(m)} = x^{(m)}$
- Given example x:
  $f_1 = \text{similarity}(x, l^{(1)})$
  $f_2 = \text{similarity}(x, l^{(2)})$
  $\vdots$
  $f_n = \text{similarity}(x, l^{(n)})$
- This will give a vector $f = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix}$ where $f_0 = 1$
- For training example $(x^{(i)}, y^{(i)})$:
  $f_1^{(i)} = \text{similarity}(x^{(i)}, l^{(1)})$
  $f_2^{(i)} = \text{similarity}(x^{(i)}, l^{(2)})$
  $\vdots$
  $f_n^{(i)} = \text{similarity}(x^{(i)}, l^{(n)})$
- This will give a vector $f^{(i)} = \begin{pmatrix} f_0^{(i)} \\ f_1^{(i)} \\ f_2^{(i)} \\ \vdots \\ f_n^{(i)} \end{pmatrix}$ where $f_0 = 1$

Now we have the hypothesis that given x, compute features $f \in \mathbb{R}^{m+1}$ ($\Theta \in \mathbb{R}^{m+1}$) predict $y = 1$ if $\Theta^T f \geq 0$ ($\Theta^T f = \Theta_0 f_0 + \Theta_1 f_1 + \cdots + \Theta_m f_m$). We get the Θ by using the following equation:

$$\min_{\Theta} C \sum_{i=1}^{m} y^{(i)} cost_1(\Theta^T f^{(i)}) + (1 - y^{(i)})\ cost_0(\Theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^{n=m} \Theta_j^2 \tag{65}$$

By solving this minimization problem you'll get the parameters for your SVM. n defines the number of features which is equal to m. Furthermore, when implementing SVM the last term ($\frac{1}{2}\sum_{j=1}^{n=m}\Theta_j^2$) is written a bit differently $\Theta^T\Theta$, where $\Theta = \begin{pmatrix} \Theta_1 \\ \Theta_2 \\ \Theta_3 \\ \vdots \\ \Theta_n \end{pmatrix}$ (IF ignoring Θ_0). Most SVM replace $\Theta^T\Theta$ with $\Theta^T M \Theta$. By using this term, algorithms can run more efficiently. M is a matrix and when M becomes very large, running the algorithm for all landmarks becomes more expensive. In order to optimize this, Θ is multiplied by M. If you want, you can apply the kernel's idea and define the source of features using landmarks for logistic regression. But the computational trick that apply for SVM don't generalize well to other algorithms like logistic regression (very slow). Now lastly we talk about **SVM parameters**:

- C ($= \frac{1}{\lambda}$)
  - Large C: Lower bias, high variance
  - Small C: Higher bias, low variance.
- σ^2
  - Large σ^2: Features f_i vary more smoothly. Higher bias, low variance
  - Small σ^2: Features f_i vary less smoothly. Lower bias, high variance

7.1.6 SVMs in Practice - Using An SVM

Strongly recommended is to use SVM software packages (e.g. liblinear, libsvm) to solve for parameters Θ. You still need to specify several parameters:

- Choice for parameter C
- Choice for kernel (similarity function)

  - No kernel ("linear kernel") where predict $y = 1$ if $\Theta^T x \geq 0$. This is a good option when n (number of features) is large, and m (number of training examples) is small ($x \in \mathbb{R}^{n+1}$).
  - Gaussian kernel: $f_i = \exp(-\frac{||x - l^{(i)}||^2}{2\sigma^2})$. This is a good option when n (number of features) is small, and m (number of training examples) is large ($x \in \mathbb{R}^n$). Note, you still need to choose σ^2. For determining more complex non-linear decision boundaries. When you choose for a Gaussian kernel, you need to do the following:

    function f = kernel(x1,x2)
    where $(-\frac{||x1 - x2||^2}{2\sigma^2})$
    return

    Note: Do perform feature scaling before using the Gaussian kernel. If your features take on different ranges of values that $||x - l||$ different features will have varying effect based on the ranges (i.e. features with small values will have a low effect on f).
  - Other choices of kernel. Note, not all similarity functions make valid kernels. Need to satisfy technical condition called "Mercer's Theorem" to make sure SVM packages' optimizations run correctly, and do not diverge.
    - Polynomial kernel (e.g. similarity$(x, l) = (x^T l +$ constant $)^{degree}$).
    - More esoteric: String kernel, chi-square kernel, histogram intersection kernel,...

Now we'll discuss SVM in relation to multi-class classification. Many SVM packages already have built-in multi-class classification functionality. Otherwise, use one-vs.-all method: train K SVMs ($y \in \{1, 2, 3, \cdots, K\}$), one to distinguish $y = i$ from the rest, for $i = 1, 2, \cdots, K$, get $\Theta^1, \Theta^2, \cdots, \Theta^K$. Pick class i with largest $(\Theta^{(i)})^T x$. Finally, we developed SVMs

staring from logistic regression and modifying it a bit. Now let's state the following: n = number of features ($x \in \mathbb{R}^{n+1}$), m = number of training examples. If n is large (relative to m)(e.g $n \geq m$, $n = 10000$, $m = 1000$), we use logistic regression or SVM without a kernel ("linear kernel"). Subsequently, if n is small and m is intermediate ($n = 1-1000$, $m = 10-10000$) use a SVM with Gaussian kernel. Lastly, when n is small, and m is large ($n = 1 - 1000$, $m > 50000$) create/add more features, then use logistic regression or SVM without a kernel. Note: Logistic regression or SVM without a kernel are pretty similar algorithms/models. Finally, neural network is likely to work well for most of these setting, but might be slower to train. However, it turns out that the optimization problem that the SVM has a convex optimization problem and so SVM software packages will always find the global minimum or something close to it (i.e. you don't have to worry about local optima).

8 Week 8

8.1 Unsupervised Learning

8.1.1 Clustering - Unsupervised Learning: Introduction

Unsupervised learning includes learning from unlabelled data. As opposed to supervised learning where the data is labelled (see Figure 1). A training set with unsupervised learning looks as follows: $\{x^{(1)}, x^{(2)}, x^{(3)}, \cdots, x^{(m)}\}$ (no $y^{(i)}$ labels). A few applications of clustering (unsupervised learning) is market segmentation, social network analysis, organize computing clusters and astronomical data analysis.

8.1.2 Clustering - K-Means Algorithm

In the clustering problem we are given an unlabelled data set and we would like to have an algorithm automatically group the data into coherent subsets or into coherent clusters for us. The K Means algorithm is by far the most popular, by far the most widely used clustering algorithm. If we want to run the K Means clustering algorithm the first step is to randomly initialize two points called **the cluster centroids**. Subsequently, K Means is an iterative algorithm and it does two things. The first is a **cluster assignment step** (outer loop) and the second is a **move centroid step** (inner loop). The clustering assignment step runs through each of the examples and depending on whether it is closer to one of the two cluster centroids it assigns each of the data points to one of the two. The second part, namely the move centroid step, takes the average of the assigned points per cluster centroid and assigns the averaged values to the centroids. These two loops are iterated until the cluster centroids and assignment of data points will not change. To write this out in a more formal way, we take as input:

- K (number of clusters)
- Training set $\{x^{(1)}, x^{(2)}, x^{(3)}, \cdots, x^{(m)}\}$, where $x^{(i)} \in \mathbb{R}^n$ (drop $x_0 = 1$ convention)

Subsequently, randomly initialize K cluster centroids $\mu_1, \mu_2, \cdots, \mu_k \in \mathbb{R}^n$

Repeat {
    for $i = 1{:}m$
        $c^{(i)}$:= index (from 1 to K) of cluster centroid closest to $x^{(i)}$
    for $k = 1$ to K
        μ_k := average (mean) of points assigned to cluster k
}

A few points regarding the K-Means algorithm:

- **Cluster assignment step** Another way to define $c^{(i)}$ is the value of k that minimizes the distance between $x^{(i)}$ and μ_k: $\min_k ||x^{(i)} - \mu_k||^2$.
- **Move centroid step** μ_k will be an n-dimensional vector ($\in \mathbb{R}^n$). When a cluster has 0 points assign to it, this cluster is usually removed. Another option is to randomly assign the cluster to another value (less commmon).

K-Means algorithms can also be applied to dataset where you have non-separated clusters.

8.1.3 Clustering - Optimization Objective

K-Means also has an optimization objective. The K-Means optimization objective includes the following parameters, namely:

- $c^{(i)}$= index $(1, 2, \cdots, K)$ to which example $x^{(i)}$ is currently assigned.
- μ_k = cluster centroid k ($\mu_k \in \mathbb{R}^n$)
- $\mu_{c^{(i)}}$ = cluster centroid of cluster to which example $x^{(i)}$ has been assigned (e.g. when $x^{(i)}$ is assigned to cluster 5, then $c^{(i)} = 5$ and $\mu_{c^{(i)}} = \mu_5$).

Our optimization objective is as follows:

$$J(c^{(1)}, \cdots, c^{(m)}, \mu_1, \cdots, \mu_K) = \frac{1}{m} \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2 \tag{66}$$

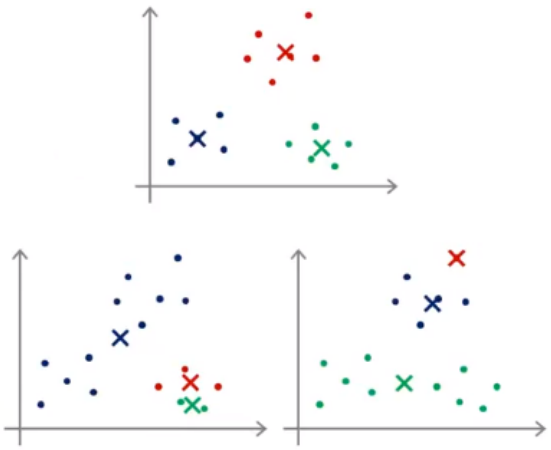


Figure 23: Possible global and local optima of K-mean algorithm.

$$\min_{c^{(1)},\cdots,c^{(m)},\mu_1,\cdots,\mu_K} J(c^{(1)},\cdots,c^{(m)},\mu_1,\cdots,\mu_K) \tag{67}$$

This cost function is sometimes also called the **distortion function**. Now we look again at the algorithm:

Repeat {
    for $i = 1{:}m$
        $c^{(i)}$:= index (from 1 to K) of cluster centroid closest to $x^{(i)}$
    for $k = 1$ to K
        μ_k := average (mean) of points assigned to cluster k
    }

So what the cluster assignment step does is minimizes $J(\cdots)$ with respect to the variables $c^{(1)}, c^{(2)}, \cdots, c^{(m)}$, while holding $\mu_1, \mu_2, \cdots, \mu_k$ fixed. Subsequently, the move centroid step, minimizes $J(\cdots)$ with respect to the variables $\mu_1, \mu_2, \cdots, \mu_k$.

8.1.4 Clustering - Random Initialization

Now we are going to discuss how to randomly initialize K cluster centroids $\mu_1, \mu_2, \cdots, \mu_k \in \mathbb{R}^n$.

- Should have $K < m$
- Randomly pick K training examples
- Set $\mu_1, \mu_2, \cdots, \mu_k$ equal to these K examples (e.g. $\mu_1 = x^{(i)}, \mu_2 = x^{(i)}$).

Depending on the initial randomized values, K can end up at different local optima (instead of the global optimum)(see Figure 23). Local optima correspond to $J(c^{(1)}, \cdots, c^{(m)}, \mu_1, \cdots, \mu_K)$ where K got stuck in a local optima. To optimize randomly initialization is to initialize multiple times. Concretely, this goes as follows:

For i = 1 to 100 (typical is 50-1000 iterations) {
    Randomly initialize K-means
    Run K-means. Get $c^{(1)}, \cdots, c^{(m)}, \mu_1, \cdots, \mu_K$
    Compute cost function (distortion)
        $J(c^{(1)}, \cdots, c^{(m)}, \mu_1, \cdots, \mu_K)$
    }

Finally, pick clustering that gave lowest cost $J(c^{(1)}, \cdots, c^{(m)}, \mu_1, \cdots, \mu_K)$. When you get a small number of clusters K (2 - 10), multiple initializations can sometimes help you find much better clustering of the data. However, if you are learning a large number of clusters K, multiple initializations is less likely to make a significant difference.

8.1.5 Clustering - Choosing the Number of Clusters

The most common way is to choose the number of clusters by hand. One method is the **Elbow method**. When using the Elbow method, the number of clusters K is varied with respect to the cost/distortion function J. The "Elbow" is

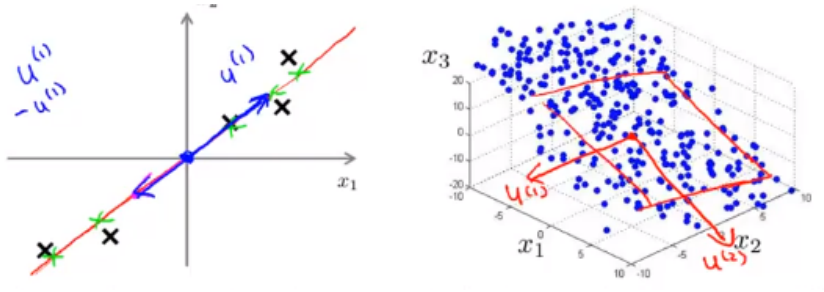


Figure 24: Principal Component Analysis (PCA) problem formulation.

the point in the curve where the distortion error goes down really slow. Often, the location of the "Elbow" is less clear. Sometimes, you're running K-Means to get clusters for use for some later/downstream purpose. Evaluate K-Means based on a metric for how well it performs for that later purpose.

8.2 Dimensionality Reduction

8.2.1 Motivation - Motivation I: Data Compression

A second type of unsupervised learning problem is **dimensionality reduction**. A reason for using this approach is data compression. Let's say we collected data with many many features two of them being length in centimetre en length in inches. This gives us a highly redundant representation. Consequently, what we can be done is reducing the data to one-dimension (instead of two) when features are highly correlated. The line on which the points lie can be seen as one feature (e.g. z_1):

$x^{(1)} \in \mathbb{R}^2 \rightarrow z^{(1)} \in \mathbb{R}$
$x^{(2)} \in \mathbb{R}^2 \rightarrow z^{(2)} \in \mathbb{R}$
$\vdots$
$x^{(m)} \in \mathbb{R}^2 \rightarrow z^{(m)} \in \mathbb{R}$

So to summarize, the examples of the original data set can be projected on a straight line, where you only need one real number to represent the location of each training example. Eventually, this reduction leads to a faster running algorithm. When you have a 3-dimensional data set, after data reduction, the dataset can be represented with a 2-dimensional vector ($z^{(i)} \in \mathbb{R}^2$).

8.2.2 Motivation - Motivation II: Visualization

The number of dimensions for reduced data is usually 2 or 3, since we can plot 2D or 3D but don't have ways to visualize higher dimensional data.

8.2.3 Principal Component Analysis - Principal Component Analysis Problem Formulation

Principal Component Analysis (PCA) problem formulation: How to determine the line (one-dimensional) representing the two-dimensional locations of each training example. PCA determines a line on which to project the data while minimizing the projection error (difference between values of training example and prediction/projection on the line). Before applying PCA, it's standard practice to first perform mean normalization and feature scaling (zero mean and comparable ranges of values). Let's write down the PCA a bit more formally:

- Reduce from 2-dimension to 1-dimension: Find a direction (a vector $u^{(1)} \in \mathbb{R}^n$) onto which to project the data so as to minimize the projection error (see Figure 24, left).
- *In general*: Reduce from n-dimension to k-dimension: Find k vectors $u^{(1)}, u^{(2)}, \cdots, u^{(k)}$ onto which to project the data so as to minimize the projection error (see Figure 24, right).

Note: PCA is not linear regression! With linear regression we fit a straight line where we minimize the squared error (vertical distance between training example and predicted value). With PCA the error includes the shortest distance to the predicted line (orthogonal). Moreover, with linear regression there is a special value y that we try to predict ($x \rightarrow y$). With PCA this is not the case, since al variables $(x_1, x_2, \cdots, x_n)$ are treated equally.

8.2.4 Principal Component Analysis - Principal Component Analysis Analysis Algorithm

Before applying PCA, there is a **data preprocesssing step**:

Training set: $x^{(1)}, x^{(2)}, \cdots, x^{(m)}$
Preprocessing (feature scaling/mean normalization):
$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$ (calculate mean of each feature)
Replace each $x_j^{(i)}$ with $x_j - \mu_j$.
If different features on different scales (e.g. x_1 = size of house, x_2 = number of bedrooms), scale features have to have comparable range of values ($x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{s_j}$)

With PCA we want to find n vectors on which to project our data. When reducing data from 2D to 1D we have $x^{(i)} \in \mathbb{R}^2 \rightarrow z^{(i)} \in \mathbb{R}$. When we reduce data from 3D to 2D we get $x^{(i)} \in \mathbb{R}^3 \rightarrow z^{(i)} \in \mathbb{R}^2$. The procedure for **PCA algorithm** is as follows:

Reduce data from n-dimensions to k-dimensions
Compute "covariance matrix":
$\Sigma = \frac{1}{m}\sum_{i=1}^{n}(x^{(i)})(x^{(i)})^T$
Compute "eigenvectors" of matrix Σ:
[U,S,V] = svd(Sigma);

Σ is used to denote a covariance matrix. This matrix Σ is then used to compute the "eigenvectors". "svd" stands for singular value decomposition. The Σ matrix will be an $n \times n$ matrix (number of dimensions). The output is an U matrix, where the columns represent the vectors $u^{(1)}, u^{(2)}, \cdots, u^{(n)}$ where $U \in \mathbb{R}^{n \times n}$. So, from [U,S,V] = svd(Sigma); we get:

$$\begin{pmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \cdots & u^{(n)} \\ | & | & & | \end{pmatrix} \in \mathbb{R}^{n \times n}$$

Where we take $u^{(1)}$ to $u^{(k)}$. Then we need to find $x \in \mathbb{R}^n \rightarrow z^{(i)} \in \mathbb{R}^k$. We do this by taking the first k columns of the U vector:

$$\begin{pmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \cdots & u^{(k)} \\ | & | & & | \end{pmatrix} \in \mathbb{R}^{n \times k} = U_{reduced}$$

and where:

$$z = \begin{pmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \cdots & u^{(k)} \\ | & | & & | \end{pmatrix}^T x = \begin{pmatrix} -(u^{(1)})^T - \\ \vdots \\ -(u^{(k)})^T - \end{pmatrix} x \text{ where } z \in \mathbb{R}^k$$

The first matrix is $n \times k$, after transposing this becomes are $k \times n$ matrix while x is an $n \times 1$ matrix, eventually yielding a $k \times 1$ matrix, where $z \in \mathbb{R}^k$. So to summarize:

- After mean normalization (ensure every feature has zero mean), and optionally feature scaling:
- Sigma = $= \frac{1}{m}\sum_{i-1}^{m}(x^{(i)})(x^{(i)})^T$ where $X = \begin{pmatrix} -(x^{(1)})^T - \\ \vdots \\ -(x^{(m)})^T - \end{pmatrix}$ and as a vectorized implementation in MATLAB/Octave:
  Sigma = (1/m) * X' * X;
- [U,S,V] = svd(Sigma);
- Ureduce = U(:,1:k); followed by z = Ureduce' * x;.

8.2.5 Applying PCA - Reconstruction from Compressed Representation

Reconstruction from compressed representation ($z \in \mathbb{R} \rightarrow x \in \mathbb{R}^2$) goes as follows: $X_{approx}^{(i)} = U_{reduce} \cdot z^{(i)}$.

8.2.6 Applying PCA - Choosing the Number of Principal Components

In the PCA algorithm we take n dimensional features and reduce them to k dimensional feature representation. This number k is a parameter of the PCA algorithm. The number k is called the principle components of the number of principle components we have trained. We will now discuss how to choose this parameter k of PCA. In order to choose k, we discuss a few useful concepts:

- PCA tries to minimize the average squared projection error: $\frac{1}{m}\sum_{i=1}^{m}||x^{(i)} - x_{approx}^{(i)}||^2$
- The total variation in the data is: $\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}||^2$
- Typically, choose k to be smallest value so that: $\frac{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}-x_{approx}^{(i)}||^2}{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}||^2} =\leq 0.01/0.05/0.10$ (1%/5%/10% where 99%/95%/90% of variance is retained).

How do you implement this? The algorithm is as follows:

Try PCA with $k = 1$
Compute $U_{reduce}, z^{(1)}, z^{(2)}, \cdots, z^{(m)}, x_{approx}^{(1)}, \cdots, x_{approx}^{(m)}$
Check if $\frac{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}-x_{approx}^{(i)}||^2}{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}||^2} \leq 0.01$?

Dis procedure is done for different values of k $(1, 2, 3, \cdots, 17)$. However, this procedure is not very efficient. However, when you call [U,S,V] = svd(Sigma), you get S which is an $n \times n$ diagonal matrix: $S = \begin{pmatrix} S_{11} & & & & \\ & S_{22} & & & \\ & & S_{33} & & \\ & & & \vdots & \\ & & & & S_{nn} \end{pmatrix}$ were everything of the diagonal will have the value 0. So the last term where the variance is calculated can be written as follows: $1 - \frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \geq 0.99$ where k is the number of dimensions and n the total amount of entrances. You can calculate this quantity for different values of k. Now you only need to call this function once, en keep changing the denominator. This allows you to select the value k without running PCA from scratch. So as we said earlier, PCA chooses a direction $u^{(1)}$ (or k directions $u^{(1)}, \cdots, u^{(k)}$) onto which to project data so as to minimize the (squared) projection error. Another way to say the same is that PCA tries to minimize: $\frac{1}{m}\sum_{i=1}^{m}||x^{(i)} - x_{approx}^{(i)}||^2$.

8.2.7 Applying PCA - Advice for Applying PCA

PCA can speed up the running time of an learning algorithm. This **supervised learning speedup** goes as follows. You have a dataset $(x^{(1)}, y^{(1)}, x^{(2)}, y^{(2)}, \cdots, x^{(m)}, y^{(m)})$ where $x^i \in \mathbb{R}^{10000}$ (high dimensional vector). For example, an image with a 100×100 pixels. This is a lot of data, resulting in a more slowly running algorithm. PCA helps to run the algorithm more efficiently by first extracting an unlabelled dataset $(x^{(1)}, x^{(2)}, \cdots, x^{(m)} \in \mathbb{R}^{10000})$ and put it into a new variable z $(x^{(1)}, x^{(2)}, \cdots, x^{(m)} \in \mathbb{R}^{1000})$ yielding a new training set $(z^{(1)}, y^{(1)}, z^{(2)}, y^{(2)}, \cdots, z^{(m)}, y^{(m)})$. Now the training examples are presented by a lower dimensional representation. Finally, we can take this reduced training set and feed it to a learning algorithm. Note: Mapping $x^{(i)} \rightarrow z^{(i)}$ should be defined by running PCA only on the training set. This mapping can be applied to the examples $x_{cv}^{(i)}$ and $x_{test}^{(i)}$ in the cross validation and test sets. To summarize:

- Compression (choose k by % of variance retained)
  - Reduce memory/disk needed to store data
  - Speed up learning algorithm
- Visualization ($k = 2$ or $k = 3$)

One bad use of PCA is to prevent an algorithm from overfitting: The reasoning behind this method is using $z^{(i)}$ instead of $x^{(i)}$ to reduce the number of features ($k < n$). Thus, fewer features, less likely to overfit. However, this might work OK, but it is not a good way to address overfitting. Use regularization instead ($\frac{\lambda}{2m}\sum_{j=1}^{n}\Theta_j^2$). With PCA you don't use the labels y, throwing away some useful information. Lastly, PCA is sometimes used where it shouldn't be. Often people design a Machine Learning system as follows:

- Get training set $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$
- Run PCA to reduce $x^{(1)}$ in dimension to get $z^{(1)}$

- Train logistic regression on $\{(z^{(1)}, y^{(1)}), (z^{(2)}, y^{(2)}), \cdots, (z^{(m)}, y^{(m)})\}$
- Test on test set: Map $x_{test}^{(i)}$ to $z_{test}^{(i)}$. Run $h_\Theta(z)$ on $\{(z_{test}^{(1)}, y_{test}^{(1)}), \cdots, (z_{test}^{(m)}, y_{test}^{(m)})\}$

How about doing the whole thing without using PCA? Before implementing PCA, first try running whatever you want to do with the original/raw data $x^{(i)}$. Only if that doesn't do what you want, then implement PCA and consider using $z^{(i)}$.

9 Week 9

9.1 Anomaly Detection

9.1.1 Density Estimation - Problem Motivation

Anomaly detection problem refers to an anomalous x_{test} given an unlabelled training set ($\{x^{(1)}, x^{(2)}, \cdots, x^{(m)}\}$). So we want to make an algorithm that tells us whether new data is anomalous. Formally we write this $p(x)$ given a certain *model*. Subsequently when $p(x_{test}) < \epsilon \rightarrow$ flag "anomaly", whereas when $p(x_{test}) \geq \epsilon \rightarrow$ flag "OK". Perhaps the most common application of anomaly detection is fraud detection:

$x^{(i)}$ = features of users i's activities
model $p(x)$ from data.
Identify unusual users by checking which have $p(x) < \epsilon$.

Another example is manufacturing. As well as monitoring computers in a data center:
$x^{(i)}$ = features of machine i
x_1 = memory use, x_2 = number of disk accesses/sec, x_3 = CPU load, x_4 = CPU load/network traffic $\cdots$
Identifying when unusual things happens on the machines.

9.1.2 Density Estimation - Gaussian Distribution

Gaussian (Normal) Distribution is as follows. Say $x \in \mathbb{R}$. If x is a distributed Gaussian with mean μ and variance σ^2, then we write this as x $N(\mu, \sigma^2)$ ("x is distributed as..."). The corresponding plot is a bell-shaped curve, where the centre is the mean μ and the width of the curve is the sigma σ^2 (i.e. variance, σ is called the standard deviation). The formula for the Gaussian distribution is:

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{(2\pi)}\sigma} exp(-\frac{(x-\mu)^2}{2\sigma^2}) \tag{68}$$

The area under the curve is equal to 1 (total probability). The **parameter estimation problem** is as follows. Given a dataset ($\{x^{(1)}, x^{(2)}, \cdots, x^{(m)}\}$) where $x^{(i)} \in \mathbb{R}$ and x $N(\mu, \sigma^2)$, we have to estimate parameters μ and σ^2. We estimate the parameters as follows:

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \tag{69}$$

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu)^2 \tag{70}$$

9.1.3 Density Estimation - Algorithm

Now we will apply a Gaussian distribution to develop a anomaly detection algorithm. Say we have a dataset ($\{x^{(1)}, x^{(2)}, \cdots, x^{(m)}\}$) where $x^{(i)} \in \mathbb{R}^n$. We are going to model $p(x)$, where $p(x) = p(x_1)p(x_2)p(x_3)\cdots p(x_n)$ (n are the features of our dataset). We are going to assume that every feature x_n $N(\mu, \sigma^2)$:

$$p(x) = p(x_1; \mu_1, \sigma_1^2) \cdot p(x_2; \mu_2, \sigma_2^2) \cdot p(x_3; \mu_3, \sigma_3^2) \cdot \cdots \cdot p(x_n; \mu_n, \sigma_n^2) \tag{71}$$

This can be written more compactly:

$$p(x) = \Pi_{j=1}^{n} p(x_j; \mu_j; \sigma_j^2) \tag{72}$$

where Π is the product (multiplication) of the different elements. So putting everything together will yield the following anomaly detection algorithm:

1. Choose features x_i that you think might be indicative of anomalous examples
2. Fit parameters $\mu_1, \cdots, \mu_n, \sigma_1^2, \cdots, \sigma_n^2$
   $\mu_j = \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)}$
   $\sigma_j^2 = \frac{1}{m} \sum_{i=1}^{m} (x_j^{(i)} - \mu_j)^2$
3. Given new example x, compute $p(x)$:
   $p(x) = \Pi_{j=1}^{n} p(x_j; \mu_j; \sigma_j^2) = \Pi_{j=1}^{n} \frac{1}{\sqrt{(2\pi)}\sigma_j} exp(-\frac{(x_j-\mu_j)^2}{2\sigma_j^2})$
   Anomaly if $p(x) < \epsilon$

9.1.4 Building an Anomaly Detection System - Developing and Evaluating an Anomaly Detection System

When developing a learning algorithm (choosing features etc.), making decisions is much easier if we have a way of evaluating our learning algorithm (i.e. **real-number evaluation**). Now let's assume we have some labelled data, of anomalous and non-anomalous examples ($y = 0$ if normal, $y = 1$ if anomalous). We also have some non-anomalous examples ($\{x^{(1)}, x^{(2)}, \cdots, x^{(m)}\}$) (assume normal examples/not anomalous). We are going to think of this data set as a training set (unlabelled). Next we are going to define a cross validation set and a test set, in which examples are present that are known to be anomalous. So we have:

- A training set (unlabelled, non-anomalous examples): $\{x^{(1)}, x^{(2)}, \cdots, x^{(m)}\}$ (it is okay if a few anomalous examples occur in the training set)
- A cross validation set (labelled with anomalous examples): $(x_{cv}^{(1)}, y_{cv}^{(1)}), \cdots, (x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})$
- A test set (labelled with anomalous examples): $(x_{test}^{(1)}, y_{test}^{(1)}), \cdots, (x_{test}^{(m_{test})}, y_{test}^{(m_{test})})$

with $y = 1$ typically around 2 - 50 examples. So let's take an example of aircraft engines. When we have 1000 good (normal) engines and 20 flawed (anomalous) engines. We divide the examples as follows:

- Training set: 6000 good engines.
- Cross validation set: 2000 good engines ($y = 0$) and 10 anomalous ($y = 1$)
- Test set: 2000 good engines ($y = 0$) and 10 anomalous ($y = 1$)

Using the same data for the cross validation and test set is not good practice. Next we fit model $p(x)$ on training set $\{x^{(1)}, x^{(2)}, \cdots, x^{(m)}\}$. On a cross validation/test example x, predict:

$$y = \begin{cases} 1 & \text{if } p(x) < \epsilon \text{ (anomaly)} \\ 0 & \text{if } p(x) \geq \epsilon \text{ (normal)} \end{cases} \tag{73}$$

The data is very skewed, therefore classification accuracy is not a good evaluation matrix. Other possible evaluation metrics are:

- True positive, false positive, false negative, true negative
- Precision/Recall
- F_1-score

Moreover, we can also use cross validation set to choose parameter ϵ.

9.1.5 Building an Anomaly Detection System - Anomaly Detection vs. Supervised Learning

Now when to use a anomaly detection or a supervised learning algorithm? An anomaly detection algorithm can be used when:

- Very small number of positive examples ($y = 1$)(0 - 20 is common).
- Large number of negative ($y = 0$) examples where with fit model $p(x)$.
- *Note*: There are many different "types" of anomalies. Hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far.
- *Applications*: fraud detection, manufacturing (e.g. aircraft engines) and monitoring machines in data center.

A supervised learning algorithm can be used when:

- Large number of positive and negative examples.
- *Note*: Enough positive examples for algorithm to get a sense of what positive examples are like, future positive examples likely to be similar to ones in training set.
- *Applications*: email spam classification, weather prediction (sunny/rainy/etc) and cancer classification.

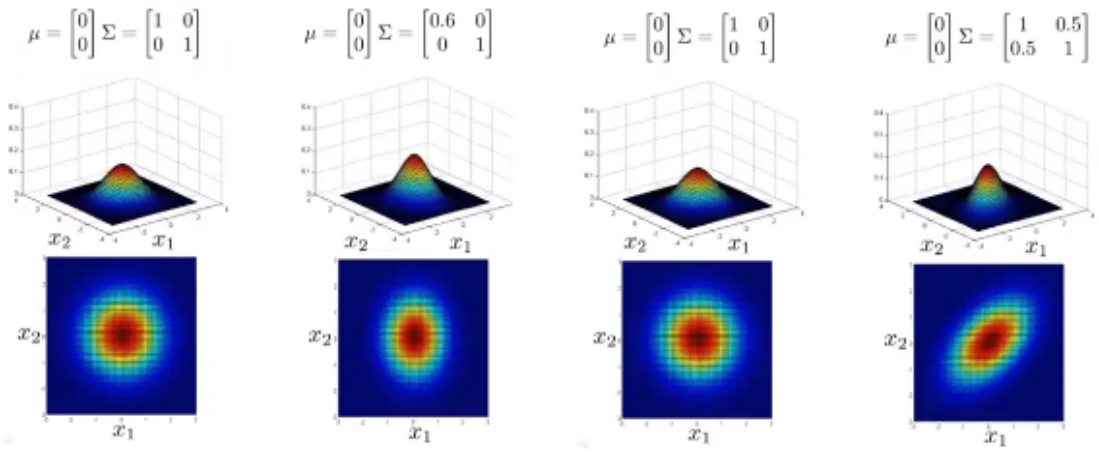


Figure 25: Examples for different values in Σ covariance matrix.

9.1.6 Building an Anomaly Detection System - Choosing What Features to Use

Sometimes data will have non-Gaussian features. This is solved by transformation (e.g. $log(x)$, $log(x+c)$, $\sqrt{x}$) eventually yielding a distribution that is Gaussian (normal). Another thing is how to come up with features for your anomaly detection algorithm (**error analysis for anomaly detection**). Eventually we want $p(x)$ to be large for normal examples x and $p(x)$ to be small for anomalous examples x. The most common problem is that $p(x)$ is comparable for normal and anomalous examples. We now want to come up with another feature (x_2) that enables us to distinguish between normal and anomalous examples. A tip is to choose features that might take on unusually large or small values in the event of an anomaly. Taking the example of monitoring computers in a data center with features: memory use of computer, number of disk accesses/sec, CPU load and network traffic. This case, you could expect that the CPU load will increase (e.g. stuck in the same loop), a new feature could be $\frac{CPUload}{networktraffic}$ or $\frac{(CPUload)^2}{networktraffic}$. With these types of variables you can better detect anomalous examples.

9.1.7 Multivariate Gaussian Distribution (Optional) - Multivariate Gaussian Distribution

When you have a dataset with two features (x_1 and x_2) and where x $N(x_i; \mu_i, \sigma_i^2)$, you may have an anomalous example of which the values for the separate features are not detected as anomalous (for instance when the dataset lies in an oval plane). This problem can be solved by a **Multivariate Gaussian (Normal) distribution** going as follows. When you have $x \in \mathbb{R}^n$, we don't model $p(x_1), p(x_2), \cdots$, etc. separately. Instead we model $p(x)$ all in one go with parameters $\mu \in \mathbb{R}^n, \Sigma \in \mathbb{R}^{n \times n}$ (covariance matrix). The formula for the multivariate Gaussian distribution is as follows:

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} exp(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)) \tag{74}$$

where $|\Sigma|$ = determinant of Σ. You can compute this in Octave/MATLAB with "det(Sigma)". Σ measures the variance between features, where lower values correspond to a smaller, narrower distribution (smaller variance) and higher value corresponds to a wider distribution (larger variance). You can also us the model to calculate correlations between features by increasing the off-diagonal values). When off-diagonal values are increased the distribution becomes more thinly peaked (more oval) corresponding to a $x_1 = x_2$ line. When the off-diagonal values are set to negative you will capture a negative correlation. You can also vary the values of μ which varies the peak/centre of the distribution is shifted. See Figure 25 for several examples.

9.1.8 Multivariate Gaussian Distribution (Optional) - Anomaly Detection using the Multivariate Gaussian Distribution

To recap, the Multivariate Gaussian distribution has two parameters, namely $\mu \in \mathbb{R}^n$ and $\Sigma \in \mathbb{R}^{n \times n}$, where:

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} exp(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)) \tag{72}$$

Next we want to perform datafitting giving a training set $\{x^{(1)}, x^{(2)}, \cdots, x^{(m)}\}$ where $x^{(i)} \in \mathbb{R}^n$.

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \tag{69}$$

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)(x^{(i)} - \mu)^T \tag{75}$$

So, the following steps enable you to develop an anomaly detection with the Multivariate Gaussian:

1. Fit model $p(x)$ by setting μ and Σ (with equation 69 and 75, respectively).
2. Given a new example, x (test example), compute $p(x)$ with the formula given earlier (see equation 72) and flag anomaly if $p(x) < \epsilon$.

Now what is the relationship to the original model which was:

$$p(x) = p(x_1; \mu_1, \sigma_1^2) \cdot p(x_2; \mu_2, \sigma_2^2) \cdot p(x_3; \mu_3, \sigma_3^2) \cdot \cdots \cdot p(x_n; \mu_n, \sigma_n^2) \tag{71}$$

Mathematically you can prove that the original model corresponds to Gaussian models where the contours of the probability density distribution are axis aligned (no ovally shaped distributions, i.e. no correlations). The only constrained is that the off-diagonal values are equal to 0. Now when would you use the original model or the multivariate Gaussian model? The original model is used when:

- Original model: $p(x) = p(x_1; \mu_1, \sigma_1^2) \cdot p(x_2; \mu_2, \sigma_2^2) \cdot p(x_3; \mu_3, \sigma_3^2) \cdot \cdots \cdot p(x_n; \mu_n, \sigma_n^2)$
- Manually create features to capture anomalies where x_1, x_2 take unusual combinations of values.
- Computationally cheaper (alternatively, scales better to large number of features n).
- OK even if m (training set size) is small.

The Multivariate Gaussian model is used when:

- Multivariate Gaussian Model: $p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} exp(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu))$
- Automatically captures correlations between different features.
- Computationally more expensive.
- Must have $m > n$, or else Σ is non-invertible ($m \geq 10n$ and make sure there are no redundant features).

9.2 Recommender Systems

9.2.1 Predicting Movie Ratings - Problem Formulation

Recommender systems is an important application of Machine Learning. Many websites these days (e.g. Netflix, Amazone, Ebay) offer you recommendations. Moreover, it has become apparent that the features you choose are important for the performance of learning algorithms. Recommender systems offer you recommendations which features to use. As an example we are going to predict movie ratings. Users rate movies using one to five stars (actually zero to five, making the math easier). Throughout, we will use the following notations:

- n_u = no. users
- n_m = no. movies
- $r(i,j) = 1$ if user j has rated movie i
- $y^{i,j}$ = rating given by user j to movie i (defined only if $r(i,j) = 1$).

In the recommender systems we have as input $r(i,j)$ and $y^{(i,j)}$. Now the recommender system problem is that given this dataset, how to predict other unrated movies. So, we have to come up with an algorithm that automatically recommends new movies based on previous watched items (see Figure 26).

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	0.9	0
Romance forever	5	?	?	0	1.0	0.01
Cute puppies of love	?	4	0	?	0.99	0
Nonstop car chases	0	0	5	4	0.1	1.0
Swords vs. karate	0	0	5	?	0	0.9

Figure 26: Recommender System Problem, how to predict missing ratings ("?"), based on known features.

9.2.2 Predicting Movie Ratings - Content Based Recommendations

How do we predict the missing values in Figure 26? Let's say we have two different features x_1 and x_2 which determine romance and action, respectively. Now each movie can be predicted with a feature vector $x^i = \begin{pmatrix} x_0^i \\ x_1^i \\ x_2^i \end{pmatrix}$ where $x_0 = 1$ (intercept). Next, for each user j, we will learn a parameter $\Theta^{(j)} \in \mathbb{R}^{n+1}$ (this case, x_0 as intercept, with two features x_1 and x_2). Subsequently, we will predict user j as rating movie i with $(\Theta^{(j)})^T x^{(i)}$ stars. More formally, we write down the problem as follows (**Problem formulation**):

- $r(i,j) = 1$ if user j has rated movie i (0 otherwise)
- $y^{(i,j)}$ = rating by user j on movie i (if defined)
- Θ^j = parameter vector for user j (preference of the user for different features)
- $x^{(i)}$ = feature vector for movie i
- For user j, movie i, predicted rating: $(\Theta^{(j)})^T x^{(i)}$ where $\Theta^{(j)} \in \mathbb{R}^{n+1}$
- *Temporarily*: $m^{(j)}$ = no. of movies rated by user j.

To learn $\Theta^{(j)}$ we will use a linear regression model. In order to learn the parameter vector $\Theta^{(j)} =$ we need to:

$$\min_{\Theta^{(j)}} \frac{1}{2} \sum_{i:r(i,j)=1} ((\Theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (\Theta_k^{(j)})^2 \tag{76}$$

where $i : r(i,j) = 1$ rates over all movies that user j has rated. However, we want to learn ratings for all our users. Consequently, the notation is changed slightly:

$$\min_{\Theta^1,\cdots,\Theta^{n_u}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} ((\Theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\Theta_k^{(j)})^2 \tag{77}$$

Now for the gradient descent update:

$$\Theta_k^{(j)} := \Theta_k^{(j)} - \alpha \sum_{i:r(i,j)=1} ((\Theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} \ \textit{(for } k = 0\textit{)} \tag{78a}$$

$$\Theta_k^{(j)} := \Theta_k^{(j)} - \alpha (\sum_{i:r(i,j)=1} ((\Theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \Theta_k^{(j)}) \ \textit{(for } k \neq 0\textit{)} \tag{78b}$$

This particular algorithm is called a content based recommendations or a content based approach because we assume that we different features available for different movies. In other words, these features capture the content of the movie. However, for many movies, we don't actually have such features. So, next we will talk about an approach to recommend systems that are not content based.

9.2.3 Collaborative Filtering - Collaborative Filtering

We will now discuss non-content based movies (features are unknown)(see Figure 27). Now let's say, we do know how the preference of the users for romantic or action movies ($\Theta^{(1)} = \begin{pmatrix} 0 \\ 5 \\ 0 \end{pmatrix}, \Theta^{(2)} = \begin{pmatrix} 0 \\ 5 \\ 0 \end{pmatrix}, \Theta^{(3)} = \begin{pmatrix} 0 \\ 0 \\ 5 \end{pmatrix}, \Theta^{(4)} = \begin{pmatrix} 0 \\ 0 \\ 5 \end{pmatrix}$). If we can

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	?	?
Romance forever	5	?	?	0	?	?
Cute puppies of love	?	4	0	?	?	?
Nonstop car chases	0	0	5	4	?	?
Swords vs. karate	0	0	5	?	?	?

Figure 27: Recommender System Problem, how to predict missing ratings ("?"), based on unknown features.

get these parameters of our users, we can infer x_1 and x_2 for each movie. In other words, what value should x_1 be so that $(\Theta^{(j)})^T x^{(1)} \approx y^{(1,j)}$. The formalized optimization algorithm is as follows. Given $\Theta^{(1)}, \cdots, \Theta^{(n_u)}$, to learn $x^{(i)}$:

$$\min_{x^{(i)}} \frac{1}{2} \sum_{i:r(i,j)=1} ((\Theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (x_k^{(i)})^2 \tag{79}$$

Given $\Theta^{(1)}, \cdots, \Theta^{(n_u)}$, to learn $x^{(1)}, \cdots, x^{(n_m)}$:

$$\min_{x^{(1)}, \cdots, x^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{i:r(i,j)=1} ((\Theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 \tag{80}$$

Now to summarize:

- In the previous video we were given $x^{(1)}, \cdots, x^{(n_m)}$ (and movie ratings), and estimated $\Theta^{(1)}, \cdots, \Theta^{(n_u)}$.
- In this video we were given $\Theta^{(1)}, \cdots, \Theta^{(n_u)}$, and estimated $x^{(1)}, \cdots, x^{(n_m)}$.

What you can do go back and forth between the methods ($\Theta \to x \to \Theta \to x \to \Theta \to \cdots$), which works pretty well.

9.2.4 Collaborative Filtering - Collaborative Filtering Algorithm

We'll now put the earlier discussed algorithms together and develop a **Collaborative Filtering algorithm**. Earlier we developed formula's that $\min_{\Theta^{(1)}, \cdots, \Theta^{(n_u)}} J(\Theta^{(1)}, \cdots, \Theta^{(n_u)})$ and $\min_{x^{(1)}, \cdots, x^{(n_m)}} J(x^{(1)}, \cdots, x^{(n_m)})$(equation 77 and 80, respectively). What you can do is going back and forth between both optimization objectives. Alternatively, a more efficient algorithm is where the separates optimization objectives are put into one formula, namely:

$$J(x^{(1)}, \cdots, x^{(n_m)}, \Theta^{(1)}, \cdots, \Theta^{(n_u)}) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\Theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (\Theta_k^{(j)})^2 \tag{81}$$

where $\min_{x^{(1)}, \cdots, x^{(n_m)}, \Theta^{(1)}, \cdots, \Theta^{(n_u)}} J(x^{(1)}, \cdots, x^{(n_m)}, \Theta^{(1)}, \cdots, \Theta^{(n_u)})$ (minimize both parameter simultaneously), where $x \in \mathbb{R}^n$ (no intercept).

So putting it altogether, here is our collaborative filtering algorithm:

1. Initialize $x^{(1)}, \cdots, x^{(n_m)}, \Theta^{(1)}, \cdots, \Theta^{(n_u)}$ to small random values.
2. Minimize $J(x^{(1)}, \cdots, x^{(n_m)}, \Theta^{(1)}, \cdots, \Theta^{(n_u)})$ using gradient descent (or an advanced optimization algorithm). E.g. for every $j = 1, \cdots, n_u$, $i = 1, \cdots, n_m$ (no intercept $k = 0$):

$$x_k^{(i)} := \Theta_k^{(i)} - \alpha \Big(\sum_{i:r(i,j)=1} ((\Theta^{(j)})^T x^{(i)} - y^{(i,j)}) \Theta_k^{(j)} + \lambda x_k^{(i)} \Big) \tag{82}$$

$$\Theta_k^{(j)} := \Theta_k^{(j)} - \alpha \Big(\sum_{i:r(i,j)=1} ((\Theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \Theta_k^{(j)} \Big) \tag{78}$$

3. For a user with parameters Θ and a movie with (learned) features x, predict a star rating of $\Theta^T x$.

9.2.5 Low Rank Matrix Factorization - Vectorization: Low Rank Matrix Factorization

An alternative way of writing your data is in a matrix Y where the rows represent the ratings and the columns represent the different users: $\begin{pmatrix} 5 & 5 & 0 & 0 \\ 5 & ? & ? & 0 \\ ? & 4 & 0 & ? \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 5 & 0 \end{pmatrix}$ and the predicted ratings are: $\begin{pmatrix} (\Theta^{(1)})^T(x^{(1)}) & (\Theta^{(2)})^T(x^{(1)}) & \cdots & (\Theta^{(n_u)})^T(x^{(1)}) \\ (\Theta^{(1)})^T(x^{(2)}) & (\Theta^{(2)})^T(x^{(2)}) & \cdots & (\Theta^{(n_u)})^T(x^{(2)}) \\ (\Theta^{(1)})^T(x^{(3)}) & (\Theta^{(2)})^T(x^{(3)}) & \cdots & (\Theta^{(n_u)})^T(x^{(3)}) \\ \vdots & \vdots & \vdots & \vdots \\ (\Theta^{(1)})^T(x^{(n_m)}) & (\Theta^{(2)})^T(x^{(n_m)}) & \cdots & (\Theta^{(n_u)})^T(x^{n_m}) \end{pmatrix}$. This can be written out in a vectorized manner where $X = \begin{pmatrix} -(X^{(1)})^T- \\ -(X^{(2)})^T- \\ \vdots \\ -(X^{(n_m)})^T- \end{pmatrix}$ and $\Theta = \begin{pmatrix} -(\Theta^{(1)})^T- \\ -(\Theta^{(2)})^T- \\ \vdots \\ -(\Theta^{(n_u)})^T- \end{pmatrix}$. Now given these definitions you can compute $X\Theta^T$. This algorithm is also called **low rank matrix factorization**. Now finding related movies goes as follows. For each product i, we learn a feature vector $x^{(i)} \in \mathbb{R}^n$ (for instance x_1 = romance, $_2$ = action, x_3 = comedy, etc.). Now how to find movies j related to movie i? You have to find a small value of $||x^{(i)} - x^{(j)}||$, where movie j and i are similar.

9.2.6 Low Rank Matrix Factorization - Implementational Detail: Mean Normalization

One last implementation detail is mean normalization. Let's suppose we have a new user 5 that has not rated any movies. The collaborative filtering algorithm will learn $\Theta^{(5)} \in \mathbb{R}^2$. The first term of equation 81 will have no effect since there are no rated movies (no movies with $r(i,j) = 1$). Only the regularization term will have an effect, which the algorithm wants to minimize. If your goal is to minimize this term you will end up with $\Theta^{(5)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. Also $(\Theta^{(5)})^T x^{(i)} = 0$ (new user will rate every movie with 0 stars), which is not realistic. This problem can be fixed with mean normalization. This is done by computing the average rating μ of every movie. Next, we are going to subtract the average μ of the original dataset (normalizing, with an average rating of 0). Subsequently, the newly formed normalized data set is going to be used for the collaborative filtering algorithm: for user j, on movie i predict $(\Theta^{(j)})^T x^{(i)} + \mu_i$. So specifically for user 5 $\Theta^{(5)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, where $(\Theta^{(5)})^T x^{(i)} + \mu_i$, meaning that we predict a rating based on the mean ratings given by other users.

10 Week 10

10.1 Large Scale Machine Learning

10.1.1 Gradient Descent with Large Datasets - Learning with Large Datasets

We'll now discuss large scale machine learning, where large data sets are used to train learning algorithms. When you have a large dataset ($m = 100000000$) where you want to train a logistic regression model you compute the following:

$$\Theta_j := \Theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y_i) x_j^{(i)} \qquad \textit{(simultanously update all } \Theta_j\textit{)} \tag{12}$$

However, this is computationally really expensive and time consuming. You can ask yourself why not pick a random subset of the training set (say $m = 1000$). Next, when you plot the cost function for your training and cross validation set ($J_{train}(\Theta)$ and $J_{cv}(\Theta)$, respectively) against the number of training examples you can observe whether adding more training data is useful (see Figure 17). Adding more data is only useful when the algorithm has high variance when m is small. When your algorithm has high bias when m is small, adding more features might be a better approach.

10.1.2 Gradient Descent with Large Datasets - Stochastic Gradient Descent

As was mentioned earlier, gradient descent becomes a very expensive approach when dealing with large datasets. Now suppose you are training a linear regression model with gradient descent, with:

$$h_\Theta(x) = \sum_{j=0}^{n} \Theta_j x_j \tag{30}$$

and

$$J_{train}(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y^{(i)})^2 \tag{58}$$

and

Repeat {

$$\Theta_j := \Theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y_i) x_j^{(i)} \qquad \textit{(for every } j = 0, \cdots, n\textit{)} \tag{12}$$

}

This is also called a batch gradient descent. An alternative algorithm is the **stochastic gradient descent** which is as follows:

$$cost(\Theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\Theta(x^{(i)}) - y^{(i)})^2 \tag{83}$$

$$J_{train}(\Theta) = \frac{1}{m} \sum_{i=1}^{m} cost(\Theta, (x^{(i)}, y(i))) \tag{84}$$

Here, the cost function evaluates how well the hypothesis is doing on a single training example. Stochastic gradient descent performs gradient descent (modify parameters) after *each* training example. First, you need to randomly shuffle (reorder) the training examples. Next:

Repeat {
  for $i = 1, \cdots, m$ {
  $\Theta_j := \Theta_j - \alpha(h_\Theta(x^{(i)}) - y_i) x_j^{(i)}$ *(for every* $j = 0, \cdots, n$*)*
  }
}

where $(h_\Theta(x^{(i)}) - y_i) x_j^{(i)} = \frac{\partial}{\partial \Theta_j} cost(\Theta, (x^{(i)}, y^{(i)}))$ As a result, after many loops the algorithm will end up in some regions close to the global minimum. In contrast to batch gradient descent, it will never converge completely. However, for most practical purposes, this is not a problem. Depending on the size of your dataset, the inner loop only has to run a few times.

10.1.3 Gradient Descent with Large Datasets - Mini-Batch Gradient Descent

The algorithms we talked about so far are:

- Batch gradient descent: Use al m examples in each iterations
- Stochastic gradient descent: use 1 example in each iteration

An alternative approach is **Mini-batch gradient descent**,which uses b examples in each iteration (typically $b = 2-100$). Say when $b = 10$ and $m = 1000$.

Repeat {
  for $i = 1, 11, 21, 31, \cdots, 991$ {
  $\Theta_j := \Theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\Theta(x^{(k)}) - y_k) x_j^{(k)}$ (for every $j = 0, \cdots, n$)
  }
}

Mini-batch gradient descent can be faster than batch gradient descent. Mini-batch gradient descent can be implemented in a vectorized manner enabling parallel computations. A disadvantage is that you have an extra parameter b with which you have to work.

10.1.4 Gradient Descent with Large Datasets - Stochastic Gradient Descent Convergence

As discussed earlier, checking for convergence of batch gradient descent goes as follows. You plot $J_{train}(\Theta)$ as a function of the number of iterations of gradient descent, where you sum over the entire training set. For stochastic gradient descent on the other hand, we compute $cost(\Theta, (x^{(i)}, y^{(i)}))$ before updating Θ using $(x^{(i)}, y^{(i)})$. As the cost gradient descent is scanning through our data set we compute cost BEFORE updating Θ. Finally, every 1000 iterations (say), plot $cost(\Theta, (x^{(i)}, y^{(i)}))$ averaged over the last 1000 examples processed by algorithm. This allows use to check if the gradient descent is diverging. Increasing examples will result in a smoother curve (error plotted against number of iterations). When you see an increasing error over a larger number of iterations it is advised to use a smaller value of α. The learning rate α is typically held constant. You can slowly decrease α over time if we want Θ to converge (e.g. $\alpha \frac{cons1}{iterationNumber + const2}$).

10.1.5 Advanced Topics - Online Learning

The **online learning** setting allows us to model problems where we have a continuous flood of data coming in and we would like an algorithm to learn from that. Suppose you have a shipping service where user comes, specifies origin and destination, you offer to ship their package for some asking price, and users sometimes choose to sue your shipping service ($y = 1$), sometimes not ($y = 0$). Features x capture properties of user, of origin/destination and asking price. We want to learn $p(y = 1|x; \Theta)$ to optimize price. In order to model is we can use logistic regression.

Repeat forever {
  Get (x, y) corresponding to user.
  Update Θ using (x, y):
    $\Theta_j := \Theta_j - \alpha(h_\Theta(x^{(i)}) - y_i) x_j^{(i)}$ *(for every $j = 0, \cdots, n$)*
  }

In this situations, examples that have been used by the algorithm are discarded since a continuous stream of data is available to the algorithm). Moreover, this way an algorithm can also adapt to changing user preference. Another application is called "product search" (learning to search). Say a user searches for "Android phone 1080p camera". Suppose we have 100 phones in the store and we want to return 10 results. We want to develop a learning algorithm that determines the best 10 results. Say we have $x =$ features of phone; how many words in user query match of phone, how many words in query match description phone etc. $y = 1$ if user clicks on links and $y = 0$ otherwise. Now we want to model $p(y = 1|x; \Theta)$ (this is called CTR, "Click Through Rate"). With this model we can show the user the 10 phone they're most likely to click on. We will ultimately get $(\{(x^{(1)}, y^{(1)}), \cdots, (x^{(10)}, y^{(10)}\})$. Other examples are choosing special offers to show user, customized selection of new articles and product recommendation.

10.1.6 Advanced Topics - Map Reduce and Data Parallelism

Some machine learning problems are to large to run on one computer. To solve this we can use a map reduce approach. Say you use batch gradient descent:

$\Theta_j := \Theta_j - \alpha \frac{1}{400} \sum_{i=1}^{400} (h_\Theta(x^{(i)}) - y_i) x_j^{(i)}$

- Machine 1: Use: $(x^{(1)}, y^{(1)}), \cdots, (x^{(100)}, y^{(100)})$
  - $temp_j^1 = \sum_{i=1}^{100} (h_\Theta(x^{(i)}) - y_i) x_j^{(i)}$
- Machine 2: Use: $(x^{(101)}, y^{(101)}), \cdots, (x^{(200)}, y^{(200)})$
  - $temp_j^2 = \sum_{i=101}^{200} (h_\Theta(x^{(i)}) - y_i) x_j^{(i)}$
- Machine 3: Use: $(x^{(201)}, y^{(201)}), \cdots, (x^{(300)}, y^{(300)})$
  - $temp_j^3 = \sum_{i=201}^{300} (h_\Theta(x^{(i)}) - y_i) x_j^{(i)}$
- Machine 4: Use: $(x^{(301)}, y^{(301)}), \cdots, (x^{(400)}, y^{(400)})$
  - $temp_j^4 = \sum_{i=301}^{400} (h_\Theta(x^{(i)}) - y_i) x_j^{(i)}$

This temporarily summations will be combined and used to compute Θ_j as follows:

$$\Theta_j := \Theta_j - \alpha \frac{1}{400} (temp_j^1 + temp_j^2 + temp_j^3 + temp_j^4) \qquad \textit{(for every } j = 0, \cdots, n\textit{)} \tag{85}$$

In this manner, the algorithm is able to process larger datasets. Many learning algorithms can be expressed as computing sums of functions over the training set (e.g. for advanced optimization, with logistic regression). Another applications is dividing your training set over different multi-core in one machine. An advantage of this use is that you don't have to work about network latency.

Figure 28: Photo Optical Character Recognition pipeline.

11 Week 11

11.1 Application Example: Photo OCR

11.1.1 Photo OCR - Problem Description and Pipeline

Photo OCR stands for photo **Optical Character Recognition**, which is the mechanical or electronic conversion of images of typed, handwritten or printed text into machine-encoded text. In order to perform photo OCR we must do the following: text detection, character segmentation and character classification. This is also called a photo OCR pipeline, where you have multi-modules that act one after another (see Figure 28).

11.1.2 Photo OCR - Sliding Windows

Let's take another example like pedestrian recognition, where an algorithm gives the number of pedestrians on a certain image. To train an algorithm, let's say you have $x =$ pixels in 82×36 image patches where you have positive examples with pedestrians ($y = 1$) and negative examples without pedestrians ($y = 0$). Eventually a classifier can tell us whether a pedestrian is present on a new image, where you select a small part of the image also called a patch (rectangle). The step (or stride) size refers to the size with which you move the patch in one image. This rectangle slides through your image ("sliding window") where you start with small image patches and increase the image patch every time you process the same image. Eventually, the algorithm recognizes pedestrians at a certain patch size. Now, when we look at text detection, and when you train your network, you also have positive examples with text ($y = 1$) and negative examples without text ($y = 0$). Let's assume you run your sliding window using one patch size, you end up with an image where your classifier indicates where it has found text (say, text is indicated with white, and non-text with black). Next, we use an expansion operator where it takes the processed image and expands the text-related regions. Subsequently, when we look at the expanded regions that have proper aspect ratio (height-width ratio proper for text) we draw boundary rectangles around these regions. This is how text recognition works using sliding windows. Next, for each of the positive examples (examples with text) we are going to segment characters where positive examples can be split up in the middle of the patch (between characters) and negative examples not. This is called a "1D sliding window" because we only move the patch from the left to the right.

11.1.3 Photo OCR - Getting Lots of Data and Artificial Data

Artificial data synthesis gives an unlimited supply of labelled data to train a supervised learning algorithm for the photo OCR. Moreover, by introducing distortions can also amplify your artificial data. This applies to both visual and auditive artificial data. Distortion introduced should be representation of the type of noise/distortions in the test set. For audio this could be background noise. It usually does not help to add purely random/meaningless noise to your data. Depending which noise you expect depends on the test set you have. A few point can be mentioned regarding getting more data:

- Make sure you have low bias classifier before expending the effort (plot learning curves) (e.g. keep increasing the number of features/number of hidden units in neural network until you have a low bias classifier)
- "How much work would it be to get 10x as much data as we currently have?"
  - Artificial data synthesis
  - Collect/label it yourself
  - "Crowd source" (e.g. Amazon Mechanical Turk)

11.1.4 Photo OCR - Ceiling Analysis: What Part of the Pipeline to Work on Next

Ceiling analysis estimates the error due to each component. Let's take as an example the photo OCR (see Figure 28). Now the question remains, what part of the pipeline should you spend the most time trying to improve? What we do is label the data manually so that the accuracy of the text detection is 100%, where after hopefully the accuracy of the other modules also increases. Next, we will manually label the examples correctly in the character segmentation followed by doing the same for the character recognition. When you look at the overall system accuracy you can determine which module improvement results in the largest increase in accuracy. Another more complex example of ceiling analysis is face recognition from images. The pipeline includes the camera image, preprocess (remove background), face detection

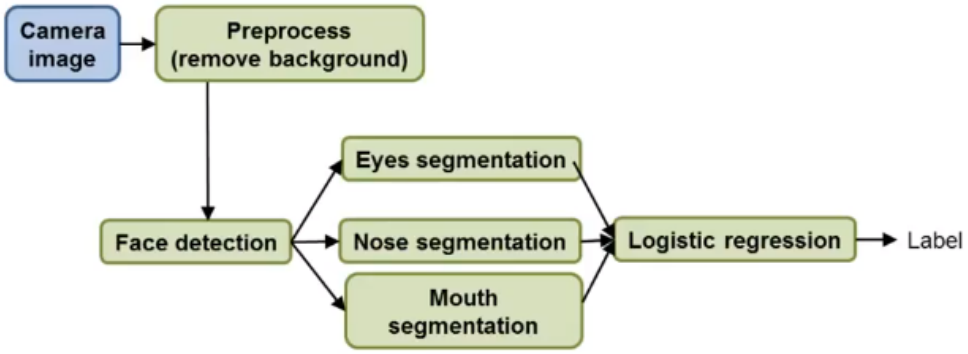


Figure 29: Pipeline for face recognition.

(eyes segmentation, nose segmentation and mouth segmentation) and logistic regression eventually yielding a label of the training example (see Figure 29).

11.1.5 Conclusion - Summary and Thank You

Main topics that have been discussed:

- Supervised Learning
  - Linear regression, logistic regression, neural networks, SVMs
- Unsupervised Learning
  - K-means, PCA, Anomaly detection
- Special applications,special topics
  - Recommender systems, large scale machine learning
- Advice on building a machine learning system
  - Bias/variance, regularization; deciding what to work on next: evaluation of learning algorithms, learning curves, error analysis, ceiling analysis.