致读者：

本人仅仅作为一个执笔者对 Andrew Ng 在 Coursera 上的公开课 Machine Learning 进行学习的总结和整理。所有内容以 Andrew 的课程笔记为主，附加了一些补充信息，这些信息来源于 Wikipedia 和其他科研工作者的博客和论文，意在避免一些错误，在此谢谢以上各位的贡献。但是不可避免的是文中还是会存在一些错误，首先请读者海涵，其次请读者指出并乐意邮件于我 995509361@qq.com，以便后续改进，也可以相互交流(不可否认我只是菜鸟级别，哈哈)。对于本文采用英文语言，是因为本人的其他原因，但是单词和语法都比较简单，很容易理解。

本文重点涉及机器学习的基本方法和策略，可能更侧重于应用，所以非常适合初学者进行学习。本人强烈推荐各位去 Coursera 上观看 Andrew 的在线教学视频(不需要翻墙即可进入主页，对于国内观看视频的问题，大家可以网搜解决方法)，课后作业会让你对算法有更好的理解(有 Matlab 和 Octave 语言基础即可完成)。另外斯坦福大学的 CS229: Machine Learning 课程据说也不错，有网友推荐一起学习，可以补充相当多的理论知识，而且能下载课件，链接 http://cs229.stanford.edu/syllabus.html。另外 Andrew 也在 Coursera 上推出了 Deep Learning，本人还没开始学习，有心得体会的读者可以分享心得。

如果各位有时间和精力的话，还可以去了解数据处理的知识和方法，这对机器学习也很有帮助。当然这不会影响对本文的阅读，如果读者本身有数据的话，可以尝试将自己的数据应用到这些算法中。

本文旨在向各位普及机器学习，希望各位对人工智能有一个大概的认知，并因此产生兴趣。不得不说的是人工智能已经作为国家战略提上日程。无论各位抱着何种心态来了解和进入这个行业，人工智能正在成为生产力的一种工具，而不再是高高在上的理论了。如果你错过了计算机时代，错过了互联网时代，那你还有机会把握人工智能时代。但希望各位不要急功近利，摒弃“携人工智能以玩资本”浮躁心态。

有人说人工智能是纨绔子弟的资本生意，有人说人工智能是未来科技的理想殿堂，本人不给人工智能任何关于地位的评价，是否参与其中源于你想或者不想。在这个以变为不变的年代，任何科技都有可能湮灭和诞生。

我希望后续能去完善这份工作，但这不是一个确定的事情。也许有一天我自己也 out，who knows?

一句话：随心所欲。

行者

2017 年 11 月 26 日

Preface

中文维基百科：机器学习是一门多领域交叉学科，涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多门学科。机器学习理论主要是设计和分析一些让计算机可以自动“学习”的算法。机器学习算法是一类从数据中自动分析获得规律，并利用规律对未知数据进行预测的算法。

Arthur Samuel[1](1959), Machine Learning: Field of study that gives computers the ability to learn without being explicitly programmed.

Tom Mitchell[2](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**. (Where Experience ~ **E**, Task ~ **T**, Performance ~ **P**).

1. Classifications

In general, machine learning problem can be assigned to one of two broad classifications: supervised learning or unsupervised learning.

1.1. Supervised Learning

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

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

For example, we have the following sets of training data:

Table 1 A training data set: regression

Input x	**Output y**
0	4
1	7
2	7
3	8

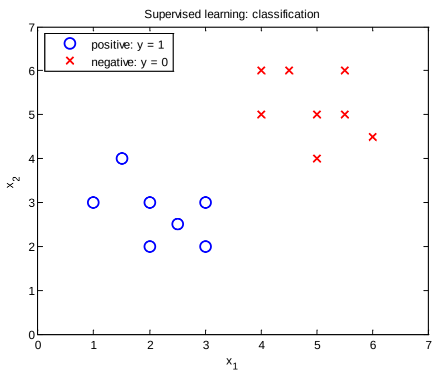


Figure 1 A training data set: classification

1.2. Unsupervised Learning

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

We can derive this structure by clustering the data based on relationships among the variables in the data. With unsupervised learning there is no feedback based on the prediction results, i.e., there is no teacher to correct you.

For example, we have the following set of training data:

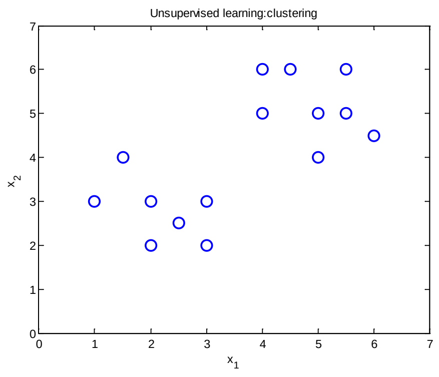


Figure 2 A training data set: clustering

2. Applications

1. Data Mining
2. Machine Vision
3. Handwriting recognition
4. Natural Language Processing(NLP)
5. Autonomous Systems
6. Recommendation System

3. Problems

When we address learning algorithms with data, sometimes we have to face problems like under-fitting problem or overfitting problem.

Under-fitting: If we don’t have enough features, the learned hypothesis may not fit the training set very well, and cannot predict new examples reasonably with high bias.

Overfitting: If we have too many features, the learned hypothesis may fit the training set very well, but fail to generalize to new examples with high variance.

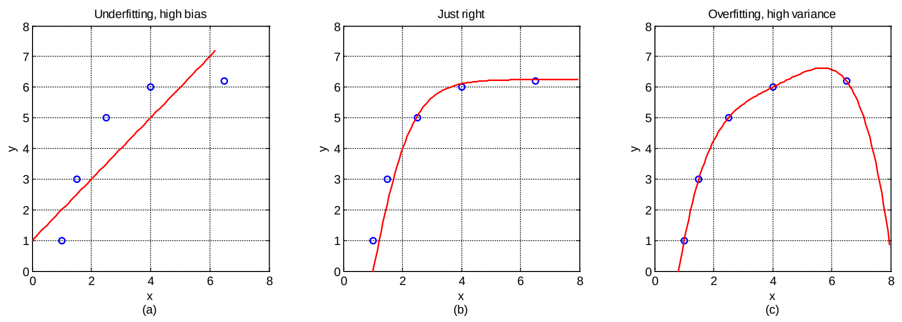


Figure 3 (a) Under-fitting problem; (b) Just right; (c) Overfitting problem

Annotations

1: Arthur Samuel is an American pioneer in the field of computer gaming and artificial intelligence, and coined “machine learning” in 1959. The Samuel Checks-playing Program appears to be the world’s first self-learning program.

*2: Tom Mitchell is an American computer scientist, known for his contributions to the advancement of machine learning, artificial intelligence, and cognitive neuroscience and he is the author of the textbook **Machine learning**.*

Regression Problem

In regression problems, we are taking input variables and trying to fit the output onto a continuous expected result function. Assuming that we have a data set including input variables or features *x* and output *y*, what we should do is that find a function that fits the data best.

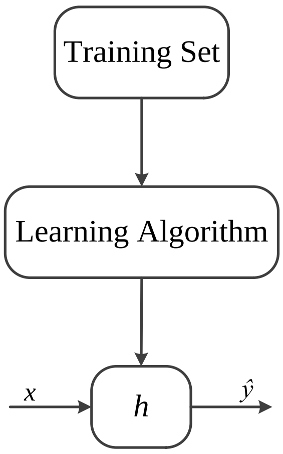


Figure 4 Regression Problem

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

x_j —	value of feature *j*
x_j^i —	value of feature *j* in the *i*th training example
x^i —	value of the *i*th training example
y^i —	value of the *i*th output
m —	the number of training example
n —	the number of features
θ_j —	parameter of x_j

1. Hypothesis Function

For linear regression problems, our hypothesis function has the general form:

$$\hat{y} = h_\theta\left(x\right) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = x^T\theta \tag{1.1}$$

Where: $x = \left[x_0 \quad x_1 \quad x_2 \quad \dots \quad x_n\right]^T$, $\theta = \left[\theta_0 \quad \theta_1 \quad \theta_2 \quad \dots \quad \theta_n\right]^T$.

Note that we give to $h_\theta\left(x\right)$ values for θ to get our estimated output $\hat{y}$. In other words we are trying to create a function called $h_\theta\left(x\right)$ to map our input data *x* to our output data *y*. And we always give $x_0 = 1$.

If *x* is set to be $x = \left[x_0 \quad x_1\right]^T$ and then $\theta = \left[\theta_0 \quad \theta_1\right]^T$, the hypothesis function is like the equation of a straight line, that is

$$\hat{y} = h_\theta\left(x\right) = \theta_0 + \theta_1 x_1$$

Then choose parameters θ so that $h_\theta\left(x\right)$ is close to *y* for our training examples(*x*, *y*).

2. Cost Function

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

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right)^2 \tag{1.2}$$

$h_\theta\left(x^i\right) - y^i$ is the difference between the predicted value and the actual value.

This function is otherwise called the “Squared error function”, or “Mean squared error”. The mean is halved $\frac{1}{2m}$ 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.

Now we are able to concretely measure the accuracy of our predictor function against the correct results we have so that we can predict new results we don’t have.

3. Gradient Descent[1]

We have our hypothesis function and we have a way of measuring how well it fits into the data. Now we need to estimate the parameters in hypothesis function when the cost function is the minimum. That’s where gradient descent comes in.

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 forward. We make steps down the cost function in the direction with the steepest descent, and the size of each step is determined by the parameter α, which is called the **learning rate**. In general, α is larger than zero, that’s $\alpha > 0$.

The gradient descent algorithm is that update the value of θ:

$$\theta_j := \theta_j - \alpha\frac{\partial J(\theta)}{\partial\theta_j}\left(j = 0,1,2,\cdots,n\right) \tag{1.3}$$

From the equation (1.1) and (1.2), we conclude that

$$\nabla J\left(\theta_j\right) = \frac{\partial J(\theta)}{\partial\theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right)x_j^i \tag{1.4}$$

4. Linear Regression Algorithm

According to the above content, the linear regression algorithm can be formed as:

$$\begin{gathered}\min_{\theta} J(\theta)\\ s.t.\ \ h_\theta(x) = x^T\theta\end{gathered} \tag{1.5}$$

When specially applied to the case of linear regression, a new form of the gradient descent equation can be derived. We can substitute our actual cost function and our actual hypothesis function and modify the equation to:

```
Set initial $\theta$;
Repeat until convergence:
  {
        $h_\theta = x^T\theta$
        $J = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right)^2$
        $\theta_j := \theta_j - \alpha\frac{\partial J(\theta)}{\partial\theta_j}\left(j = 0,1,2,\cdots,n\right)$
  }
```

If X contains all input variables or features in training data set and a column of ones and y is the set of all outputs, we get that

$$X = \begin{bmatrix} 1 & \left(x^1\right)^T \\ 1 & \left(x^2\right)^T \\ & \vdots \\ 1 & \left(x^m\right)^T \end{bmatrix}, y = \begin{bmatrix} y^1 \\ y^2 \\ \vdots \\ y^m \end{bmatrix}$$

Then sum operations in these equations will be transformed into matrix operations, and it will be more faster for computer to calculate because of avoiding loops, so (1.2) and (1.3) are calculated as[2]:

$$\begin{aligned} &H = X\theta \\ &J(\theta) = \frac{1}{2m}\left(X\theta - y\right)^T\left(X\theta - y\right) \\ &\nabla J(\theta) = \frac{1}{m}X^T\left(X\theta - y\right) \\ &\theta := \theta - \alpha\nabla J(\theta) \end{aligned} \tag{1.6}$$

Tips:(1)Debugging gradient descent. It will be useful to make a plot of J with number of iterations on the x-axis. If J ever increases or converges slowly, then you probably need to decrease α. (2)Automatic convergence test. Declare convergence if J decreases less than E in one iteration, where E is some small value such as 10^{-3}. However in practice it's difficult to choose this threshold value.

5. Example

We have a set of training data where n = 1, m = 7, meaning that the hypothesis function is like the equation of a straight line. Training data is presented in Table 2 and marked in Figure 5.

Table 2 A set of training data

x	y
0.5	2.1
0.8	2.5
1.0	3.2
1.3	3.5
1.5	3.8
1.8	4
2.0	5

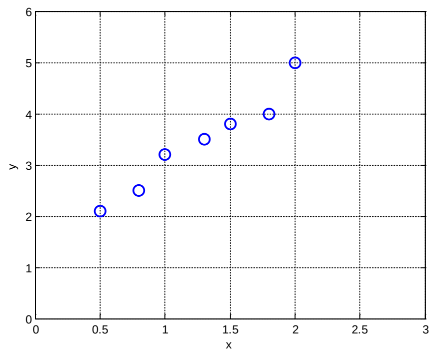


Figure 5 Marked training data

After repeating the algorithm (α = 0.1, iterations = 50), the parameters are estimated as θ = [1.2351, 1.7371] and J converges from 3.3112 to 0.0203.

As we know, the learning rate α affects convergence of this algorithm, and we choose another two values of the learning rate (α = 0.01 andα = 0.73) to estimate parameters, but the cost function converges more slowly when α = 0.01 and it is non-convergent when α = 0.73. We compare the situations by plotting the *J* curves in Figure 8.

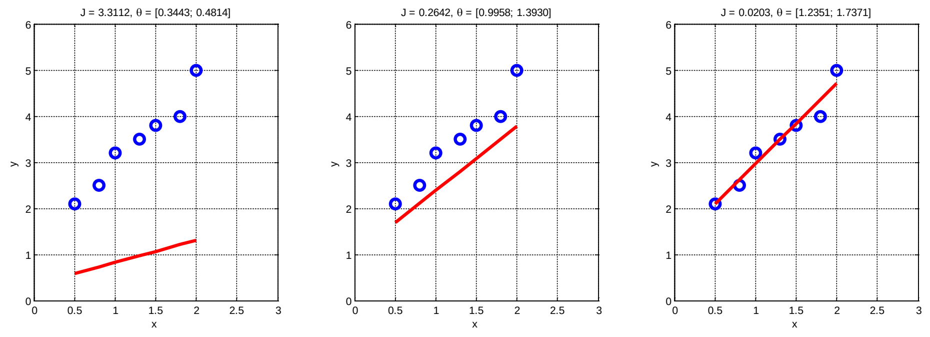


Figure 6 Hypothesis function varies with *θ*

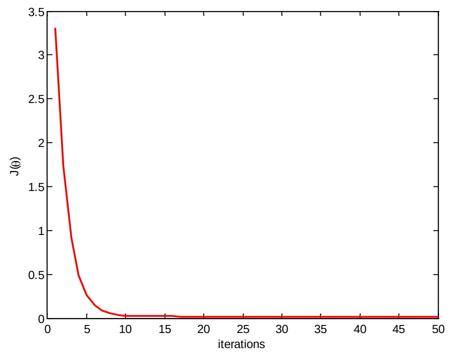


Figure 7 Cost function varies with iterations

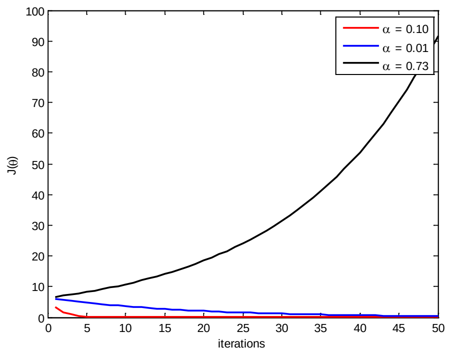


Figure 8 *J* curves vary with *α*

6. Feature Normalization

In the above example, *x* has only one feature. However it is common for us to meet linear regression problems with multiple variables. So we can speed up gradient descent by having each of our input values in roughly the range. This is because *θ* will descent quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven.

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

$$-1 \le x_j \le 1 \quad or \quad -0.5 \le x_j \le 0.5 \tag{1.7}$$

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

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

$$x_j := \frac{x_j - \mu_j}{s_j} \tag{1.8}$$

Where μ_j is the average of all the values for feature j and s_j is the range of values (i.e. max – min) or the standard deviation. Note that dividing by the range or dividing by the standard deviation, give different results. Selecting the standard deviation is more common.

7. Features and Polynomial Regression

We can improve our features and the form of our hypothesis function in a couple different ways. We can combine multiple features into one. For example, we can combine x_1 and x_2 into a new feature x_3 by taking $x_1 \cdot x_2$.

Our hypothesis function need not be linear if that does not fit the data well. We can change the behavior or curve of our hypothesis function by making it a quadratic, cubic or square root function (or any other form).

For example, if our hypothesis function is $h_\theta(x) = \theta_0 + \theta_1 x_1$ which cannot fit data well, then we can create additional features based on x_1, to get the quadratic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$ or the cubic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$. In these versions, we have created new features $x_2 = x_1^2$ and $x_3 = x_1^3$. But note that it is useless if there is a linear relationship between new features and original features and feature scaling becomes very important if you choose new features in this way.

8. Normal Equation

The "Normal Equation" is a method of finding the optimum theta without iteration. We have known from the equation (1.6) that $J(\theta)$ is a convex function[3], so it has a global minimum when $\nabla J(\theta) = 0$. From the equation

$$\nabla J(\theta) = \frac{1}{m} X^T (X\theta - y) = 0$$

we deduce that

$$\theta = (X^T X)^{-1} X^T y \tag{1.9}$$

Note that there is no need to do feature scaling with the normal equation. But sometimes $X^T X$ may be non-invertible, and the common causes are:

(1) Redundant features, where two features are very closely related (i.e. they are linearly dependent).

(2) Too many features (e.g. $m \leq n$). In this case, delete some features or use "regularization".

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

The following table is a comparison of gradient descent and the normal equation:

Table 3 Comparison of gradient descent and normal equation

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

We can use the above example in section 5 to calculate the normal equation and the result is θ = [1.2261, 1.7435] which is the analytical solution to which the solution of gradient descent algorithm is very close when the number of iteration is large enough (e.g. iterations = 1000).

Annotations

1. Gradient Descent

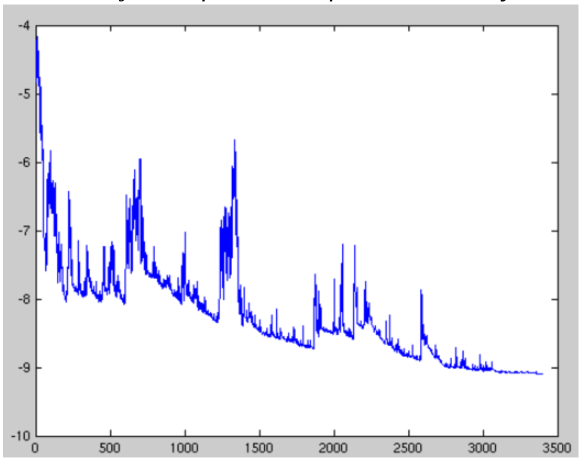
Datasets can often approach such sizes as m = 100000000. In this case, our gradient descent step will have to make a summation over all one hundred million examples. We will want to try to avoid this – the approaches for doing so are described below.

1.1 Batch Gradient Descent

From the formula(1.3), *we know that this gradient descent method computes the gradient using the whole dataset. This is called Batch Gradient Descent. It is relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution, either local or global. But it has disadvantages of time consuming and mass computation because we need to sum the cost of each sample in one parameters update.*

1.2 Stochastic Gradient Descent

Basically, in SGD, we are using the cost gradient of 1 example in each iteration, instead of using the sum of the cost gradient of all examples. However, SGD performs frequent updates with a high variance that causes the objective function to fluctuate heavily.



The cost function and gradient descent can be treated as:

$$\text{cost}^i(\theta) = \left(h_\theta\left(x^i\right) - y^i\right)^2$$

$$\theta_j := \theta_j - \alpha \frac{\partial \text{cost}^i(\theta)}{\partial \theta_j} = \theta_j - \alpha\left(h_\theta\left(x^i\right) - y^i\right)x_j^i$$

And the stochastic gradient descent algorithm's pseudocode is:

```
Repeat until convergence
{
    (1) Randomly shuffle examples in the training set;
    (2) For i in 1:m
        {
            $\theta_j := \theta_j - \alpha\left(h_\theta\left(x^i\right) - y^i\right)x_j^i \quad (j = 0,\ldots,n);$
        }
}
```

The advantages of SGD: (1) It is much faster than BGD and can be used to learn online; (2) BGD converges to the minimum of the basin, but SGD's fluctuation enables it to jump to new and potentially better local minima or maybe global minima; (3) If we decrease the learning rate, SGD will show the same convergence behavior as BGD.

1.3 Mini-Batch Gradient Descent

We have introduced two extreme algorithms in gradient descent, and there is a middlebrow to handle the problem, which is called mini-batch gradient descent.

Mini-batch gradient descent uses limited number of examples (e.g. b=10) in each iteration rather than all or only one example. So it has advantages of both algorithms and performs an update for every mini-batch of b training examples.

Its cost function and gradient descent are deduced as:

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

$$\theta_j := \theta_j - \alpha\frac{\partial J_b}{\partial \theta_j} = \theta_j - \frac{\alpha}{b}\sum_{i=1}^{b}\left(h_\theta\left(x^i\right) - y^i\right)x_j^i$$

The pseudocode is written as:

```
Repeat until convergence
{
    (1) Randomly shuffle examples in the training set;
    (2) Initialize i=1;
    (2) While (i<m)
        {
            $\theta_j := \theta_j - \frac{\alpha}{b}\sum_{i=1}^{b}\left(h_\theta\left(x^i\right) - y^i\right)x_j^i \quad (j = 0,\ldots,n);$
            $i = i + b;$
        }
}
```

The advantages of MBGD are: (1) to reduce the variance of the parameter updates and lead to more stable convergence; (2) to make use of matrix optimizations to computing gradient with mini-batch very efficient; (3) to train a neural network quickly; (3) to compute faster than BGD but slower than SGD.

These algorithms are used for minimizing not only convex functions but also non-convex problems.

1.4 Stochastic Gradient Descent with momentum

Because SGD cannot compute the exact derivate of loss function, which means the gradients update in the non-optical direction due to noisy derivatives, exponentially weighed averages can provide us a better estimate which is closer to the actual derivate than noisy calculations.

SGD with momentum helps accelerate gradients in the right direction when there are ravine areas.

General formula SGD with momentum:

$$V_j := \beta V_j + (1-\beta)\nabla J_{SGD}\left(\theta_j\right)$$

$$\theta_j := \theta_j - \alpha V_j$$

And it can be simplified as follows:

$$V_j := \beta V_j + \alpha\nabla J_{SGD}\left(\theta_j\right)$$

$$\theta_j := \theta_j - V_j$$

Another popular version of the momentum update is Nesterov Momentum:

$$\theta_j' = \theta_j - \beta V_j$$

$$V_j := \beta V_j + \alpha\nabla J_{SGD}\left(\theta_j'\right)$$

$$\theta_j := \theta_j - V_j$$

So the pseudocode is like:

```
Repeat until convergence
{
    (1) Randomly shuffle examples in the training set;
    (2) For i in 1:m
        {
            $V_j := \beta V_j + (1-\beta)\left(h_\theta\left(x^i\right) - y^i\right)x_j^i \ ,(j = 0,\ldots,n);$
            $\theta_j := \theta_j - \alpha V_j;$
            Or
            $V_j := \beta V_j + \alpha\left(h_\theta\left(x^i\right) - y^i\right)x_j^i \ ,(j = 0,\ldots,n);$
            $\theta_j := \theta_j - V_j;$
            Or
            $\theta_j' = \theta_j - \beta V_j;$
            $V_j := \beta V_j + \alpha\nabla J_{SGD}\left(\theta_j'\right);$
            $\theta_j := \theta_j - V_j;$
        }
}
```

2. Derivation of $\nabla J\left(\theta\right)$

From the cost function

$$J(\theta) = \frac{1}{2m}\left(X\theta - y\right)^T\left(X\theta - y\right) = \frac{1}{2m}\left(\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^T y\right)$$

We can deduce that

$$\nabla J\left(\theta\right) = \frac{1}{2m}\left(2X^T X\theta - 2X^T y\right) = \frac{1}{m}\left(X^T X\theta - X^T y\right) = \frac{1}{m}X^T\left(X\theta - y\right)$$

3. Convex Function

3.1 Definition

Convex set: Interval D is a convex set if

$$\forall x_1, x_2 \in D\text{, constant } \lambda \in [0,1]\text{, we have } \lambda x_1 + (1-\lambda)x_2 \in D$$

Convex function: f is a convex function on a nonempty convex set D if

$$\forall x_1, x_2 \in D \text{ and } \forall\lambda \in [0,1]\text{, we have } f\left(\lambda x_1 + (1-\lambda)x_2\right) \le \lambda f\left(x_1\right) + (1-\lambda)f\left(x_2\right)$$

A strictly convex function is defined if the inequality is strictly unequal. Any local minimum of a convex function is also a global minimum.

3.2 Decision Theorems

(1) If f is differentiable on convex set D, f is a convex function if and only if

$$\forall x \in D, x + \Delta x \in D, f\left(x + \Delta x\right) \ge f\left(x\right) + \nabla f^T\left(x\right)\Delta x$$

(2) If f is twice differentiable on convex set D, f is a convex function if and only if its Hessian matrix of second partial derivatives is positive semidefinite on the interior of the convex set.

Back to the cost function

$$J(\theta) = \frac{1}{2m}\left(X\theta - y\right)^T\left(X\theta - y\right)$$

$$\nabla J(\theta) = \frac{1}{m}X^T\left(X\theta - y\right)$$

So its Hessian matrix is a symmetric matrix as:

$$\nabla^2 J(\theta) = \frac{1}{m}X^T X$$

Set vector $v \in R^n$*, and* $v^T X^T Xv = \left(Xv\right)^T Xv \ge 0$*, then* $X^T X$ *is semidefinite and Hessian matrix is semidefinite. According to the 2nd decision theorem, we can conclude that J is a convex function. We can also prove it with the 1st decision theorem.*

3.3 Positive Semi-definite Matrix

Generally, an n×n Hermitian matrix M is positive semidefinite if the scalar z'Mz is non-negative for all non-zero column vectors z of n complex numbers. Hermitian matrix is a complex square *matrix that is equal to its own conjugate transpose. So a matrix is positive semidefinite if and only if* $z'Mz \geq 0$ *; and a matrix is positive definite if and only if* $z'Mz > 0$.

Some theorems to prove M is positive semidefinite: (1) M is positive semidefinite if and only if all eigenvalues of M are non-negative. (2) If M equals $C^T C$ *when C is real matrix, then M is positive semidefinite.*

Logistic Regression

Logistic regression is actually an approach to classification problems, not regression problem. It is named that way for historical reasons. In a classification problem, we have a training data set in which its outputs y's are discrete (e.g. $y \in \{0, 1\}$), and we try to map input variables into discrete outputs.

We firstly discuss about binary classification, then multiclass classification is analyzed.

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

x_j — value of feature *j*

x_j^i — value of feature *j* in the *i*th training example

x^i — value of the *i*th training example

y^i — value of the *i*th output

m — the number of training example

n — the number of features

θ_j — parameter of x_j

1. Binary Classification

Instead of our output vector y being a continuous range of values, it will only be 0 or 1, meaning that $y \in \{0, 1\}$ where 0 is usually taken as the "negative class" and 1 as the "positive class", but you are free to assign any representation to it.

One method is to use linear regression and map all predictions greater than 0.5 as a 1 and all less than 0.5 as a 0. This method doesn't work well because classification is not actually a linear function.

1.1. Hypothesis Function

Like linear regression problems, we firstly represent our hypothesis function which should satisfy the condition $0 \le h_\theta(x) \le 1$. Our new form uses the "Sigmoid Function", also called the "Logistic Function":

$$\begin{aligned} h_\theta(x) &= g\left(\theta^T x\right) \\ z &= \theta^T x \\ g(z) &= \frac{1}{1+e^{-z}} \end{aligned} \tag{2.1}$$

We plot curves of g varying with z as:

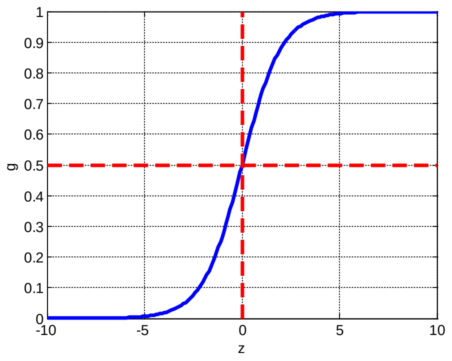


Figure 9 Curves of sigmoid function

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

From Figure 9, we find that

$$z = 0, e^0 = 1 \Rightarrow g(z) = 0.5$$
$$z \to \infty, e^{-\infty} \to 0 \Rightarrow g(z) = 1$$
$$z \to -\infty, e^{\infty} \to \infty \Rightarrow g(z) = 0$$

The h_θ give us the probability where our output is 1. For example, $h_\theta = 0.7$ give us the probability of 70% that output is 1 and the probability of 30% that output is 0. Our probability that our prediction is 0 is just the complement of our probability that it is 1.

$$\begin{aligned} &h_\theta = P(y=1 \mid x;\theta) \\ &P(y=1 \mid x;\theta) + P(y=0 \mid x;\theta) = 1 \end{aligned} \tag{2.2}$$

1.2. Decision Boundary

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

$$\begin{aligned} &h_\theta(x) \ge 0.5 \Rightarrow y = 1 \\ &h_\theta(x) < 0.5 \Rightarrow y = 0 \end{aligned} \tag{2.3}$$

The way our logistic function g behaves is that when its input is greater than or equal to zero, its output is greater than or equal to 0.5:

$$g(z) \ge 0.5 \ (z \ge 0)$$

It means when $z = \theta^T x$, we get that

$$h_\theta(x) = g(\theta^T x) \ge 0.5 \ \ (\theta^T x \ge 0) \tag{2.4}$$

Above all, we can now say:

$$\begin{aligned} &\theta^T x \ge 0 \Rightarrow y = 1 \\ &\theta^T x < 0 \Rightarrow y = 0 \end{aligned} \tag{2.5}$$

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

Again, the input to sigmoid function $g(z)$ doesn't need to be linear, and could be a function that describes a circle or any shape to fit our data.

1.3. Cost Function

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

Instead, our cost function for logistic regression looks like:

$$\begin{aligned} &J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost\left(h_\theta(x^i), y^i\right) \\ &Cost\left(h_\theta(x), y\right) = -\log\left(h_\theta(x)\right) \qquad if\ y = 1 \\ &Cost\left(h_\theta(x), y\right) = -\log\left(1 - h_\theta(x)\right) \quad if\ y = 0 \end{aligned} \tag{2.6}$$

Let us talk about the function $y = -\log(x)$ when $x \in (0,1]$, and the function curve is plotted as:

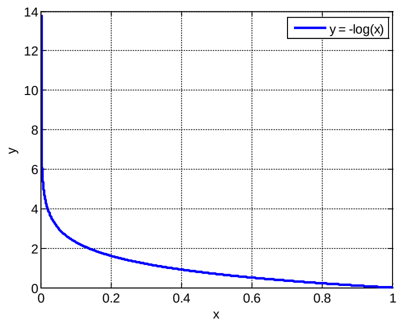


Figure 10 y varies when y=-log(x)

So $Cost\left(h_\theta(x), y\right)$ will approach infinity if $h_\theta(x) \to 0$ and will approach zero if $h_\theta(x) \to 1$ when y = 1; and $Cost\left(h_\theta(x), y\right)$ will approach infinity if $h_\theta(x) \to 1$ and will approach zero if $h_\theta(x) \to 0$ when y = 0.

[1]Note that writing the cost function in this way guarantees that $J(\theta)$ is convex for logistic regression.

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

$$Cost\left(h_\theta(x), y\right) = -y\log\left(h_\theta(x)\right) - (1-y)\log\left(1-h_\theta(x)\right) \tag{2.7}$$

Notice that when y = 1, then the second term $-(1-y)\log\left(1-h_\theta(x)\right) = 0$ and will not affect the result. If y = 0, then the first term $-y\log\left(h_\theta(x)\right) = 0$ and will not affect the result.

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

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^i \log\left(h_\theta\left(x^i\right)\right) + \left(1-y^i\right)\log\left(1-h_\theta\left(x^i\right)\right)\right) \tag{2.8}$$

In order to calculate faster, a vectorized implementation is:

$$\begin{aligned} h &= g\left(X\theta\right) \\ J(\theta) &= \frac{1}{m}\left(-y^T \log(h) - (1-y)^T \log(1-h)\right) \end{aligned} \tag{2.9}$$

where

$$X = \begin{bmatrix} 1 & \left(x^1\right)^T \\ 1 & \left(x^2\right)^T \\ & \vdots \\ 1 & \left(x^m\right)^T \end{bmatrix}, y = \begin{bmatrix} y^1 \\ y^2 \\ \vdots \\ y^m \end{bmatrix}$$

1.4. Gradient Descent

Remember that the general form of gradient descent is:

$$\theta_j := \theta_j - \alpha\frac{\partial J(\theta)}{\partial\theta_j}\left(j = 0,1,2,\cdots,n\right) \tag{2.10}$$

We need to calculate derivative of $J(\theta)$. But it will be useful to first calculate derivative of sigmoid function.

$$\begin{aligned} g'(z) &= \left(\frac{1}{1+e^{-z}}\right)' = \frac{e^{-z}}{\left(1+e^{-z}\right)^2} = \frac{e^{-z}}{\left(1+e^{-z}\right)^2} = \frac{1}{1+e^{-z}}\cdot\frac{1+e^{-z}-1}{1+e^{-z}} \\ &= g(z)\left[1-g(z)\right] \end{aligned} \tag{2.11}$$

Now we are ready to find out resulting partial derivative of J :

$$\begin{aligned} \frac{\partial J(\theta)}{\partial\theta_j} &= \frac{\partial}{\partial\theta_j}\frac{-1}{m}\sum_{i=1}^{m}\left(y^i \log\left(h_\theta\left(x^i\right)\right) + \left(1-y^i\right)\log\left(1-h_\theta\left(x^i\right)\right)\right) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial\theta_j}\left(y^i \log\left(h_\theta\left(x^i\right)\right) + \left(1-y^i\right)\log\left(1-h_\theta\left(x^i\right)\right)\right) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\left(y^i\frac{\partial}{\partial\theta_j}\log\left(h_\theta\left(x^i\right)\right) + \left(1-y^i\right)\frac{\partial}{\partial\theta_j}\log\left(1-h_\theta\left(x^i\right)\right)\right) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\left(y^i\frac{\frac{\partial}{\partial\theta_j}h_\theta\left(x^i\right)}{h_\theta\left(x^i\right)} + \left(1-y^i\right)\frac{\frac{\partial}{\partial\theta_j}\left(1-h_\theta\left(x^i\right)\right)}{1-h_\theta\left(x^i\right)}\right) \end{aligned} \tag{2.12}$$

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta_j} &= -\frac{1}{m}\sum_{i=1}^{m}\left(y^i \frac{h_\theta(x^i)\left(1-h_\theta(x^i)\right)x_j^i}{h_\theta(x^i)} + \left(1-y^i\right)\frac{-h_\theta(x^i)\left(1-h_\theta(x^i)\right)x_j^i}{1-h_\theta(x^i)} \right) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\left(y^i\left(1-h_\theta(x^i)\right)x_j^i - \left(1-y^i\right)h_\theta(x^i)x_j^i \right) \\ &= \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^i) - y^i \right)x_j^i\end{aligned}$$

And the vectorized version is:

$$\nabla J(\theta) = \frac{1}{m} X^T\left(g(X\theta) - y\right) \tag{2.13}$$

And the vectorized equation of gradient descent is followed:

$$\theta := \theta - \alpha\frac{1}{m} X^T\left(g(X\theta) - y\right) \tag{2.14}$$

There are some other advanced optimization algorithms such as "conjugate gradient[2]", "BFGS[3]" and "L-BFGS[4]" which are more sophisticated and faster to optimize θ.

1.5. Logistic Regression Algorithm

In summary, logistic regression algorithm operates as

```
Set initial $\theta$;
Repeat until convergence:
  {
      $h = g(X\theta)$
      $J(\theta) = \frac{1}{m}\left(-y^T \log(h) - (1-y)^T \log(1-h)\right)$
      $\theta := \theta - \alpha\frac{1}{m} X^T\left(g(X\theta) - y\right)$
  }
```

2. Multiclass Classification: One-vs-all

Now we will approach the classification of data into more than two categories. Instead of $y = \{0, 1\}$ we will expand our definition so that $y = \{0, 1, \dots, n\}$.

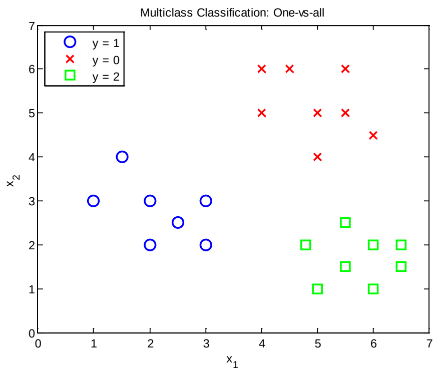


Figure 11 Multiclass Classification

In this case we divide our problem into n+1 binary classification problems; in each one, we predict the probability that 'y' is a member of one of our classes. 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 returns the highest value as our prediction.

$$\begin{aligned}
&y \in \{0,1,\dots,n\}\\
&h_\theta^0(x) = P(y=0 \mid x;\theta)\\
&h_\theta^1(x) = P(y=1 \mid x;\theta)\\
&\dots\\
&h_\theta^n(x) = P(y=n \mid x;\theta)\\
&prediction = \max_i \left(h_\theta^i(x)\right)
\end{aligned} \tag{2.15}$$

The graphic descriptions using the data in Figure 11 are:

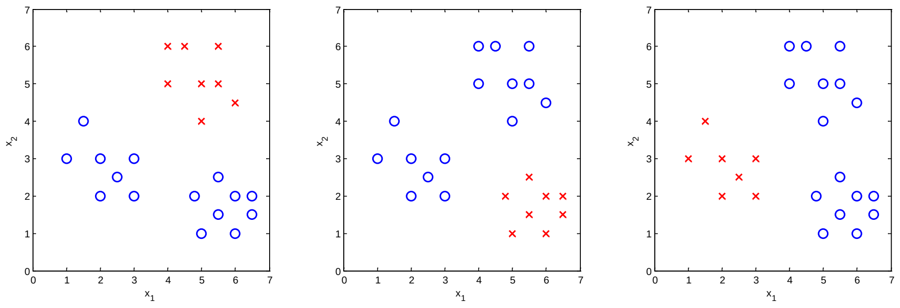


Figure 12 One-vs-all

Annotation

1. Now we prove that J(θ) is convex for logistic regression.

We have known that the derivative of J(θ) is (2.13), then it necessary to deduce its Hessian matrix:

$$\begin{aligned}
\nabla^2 J(\theta) &= \frac{1}{m}\frac{\partial}{\partial\theta}\left(X^T\left(g(X\theta)-y\right)\right)\\
&= \frac{1}{m}X^T\frac{\partial}{\partial\theta}\left(g(X\theta)-y\right)\\
&= \frac{1}{m}X^T\frac{\partial}{\partial\theta}g(X\theta)\\
&= \frac{1}{m}X^T diag(g)\,diag(\mathbf{1}-g)\,X\\
&= \frac{1}{m}X^T A^T A X
\end{aligned}$$

According to one of decision theorems of convex function, we can prove that J(θ) is convex.

2. Conjugate gradient algorithm

f(x) is continuous and differentiable, and its gradient $\nabla f(x)$ *indicates the direction of maximum increase. So the opposite direction means maximum decrease with an adjustable step length α and performs a line search in this direction until it reaches the minimum off:*

$$\begin{aligned}
&\Delta x_0 = -\nabla_x f(x_0)\\
&\alpha_0 := \arg\min_\alpha f(x_0 + \alpha\Delta x_0)\\
&x_1 = x_0 + \alpha_0 \Delta x_0
\end{aligned}$$

After this first iteration in the steepest direction Δx_0*, the following steps constitute one iteration of moving along a subsequent conjugate direction* s_n*, where* $s_0 = \Delta x_0$*:*

(1) Calculate the steepest direction: $\Delta x_n = -\nabla_x f(x_n)$

(2) Compute β_n

(3) Update the conjugate direction: $s_n = \Delta x_n + \beta_n s_{n-1}$

(4) Perform a line search: optimize $\alpha_n := \arg\min_\alpha f(x_n + \alpha s_n)$

$$\boxed{\textit{(5) Update the position: } x_{n+1} = x_n + \alpha_n s_n}$$

Some best known formulas for β_n are named after their developers:
(1) Fletcher-Reeves

$$\beta_n^{FR} = \frac{\Delta x_n^T \Delta x_n}{\Delta x_{n-1}^T \Delta x_{n-1}}$$

(2)Polak-Ribière:

$$\beta_n^{PR} = \frac{\Delta x_n^T \left(\Delta x_n - \Delta x_{n-1}\right)}{\Delta x_{n-1}^T \Delta x_{n-1}}$$

$$\beta_n = \max\left\{0, \beta_n^{PR}\right\}$$

(3)Hestenes-Stiefel:

$$\beta_n^{HS} = \frac{\Delta x_n^T \left(\Delta x_n - \Delta x_{n-1}\right)}{s_{n-1}^T \left(\Delta x_n - \Delta x_{n-1}\right)}$$

(4)Dai-Yuan:

$$\beta_n^{DY} = \frac{\Delta x_n^T \Delta x_n}{s_{n-1}^T \left(\Delta x_n - \Delta x_{n-1}\right)}$$

3. BFGS Algorithm(http://www.cnblogs.com/ljy2013/p/5129294.html)

BFGS is an optimization algorithm in the family of quasi-Newton methods created by Broyden, Fletcher, Goldfarb and Shanno.

For any function φ(x), its second-order Taylor expansion is:

$$\varphi(x) = f(x_k) + \nabla f(x_k)(x - x_k) + \frac{1}{2}(x - x_k)^T \nabla^2 f(x_k)(x - x_k)$$

Then when $\nabla\varphi(x) = 0$, we will minimize the function.

$$\nabla\varphi(x) = \nabla f(x_k) + \nabla^2 f(x_k)(x - x_k) \quad or$$

$$\nabla\varphi(x) = g_k + H_k(x - x_k) \quad when \quad g_k = \nabla f(x_k), H_k = \nabla^2 f(x_k)$$

If H_k *is nonsingular, we get the Newton's Method:*

$$x = x_k - H_k^{-1} \cdot g_k$$

But it is not so good because it needs to calculate Hessian matrix and costs more time and memory especially when addressing big data.

Quasi-Newton Method avoid calculation of Hessian matrix. Its principle is as follows:

$$\nabla\varphi(x) \approx g_k + H_k(x - x_k)$$

Set $x = x_{k+1}$, *we get that*

$$g_{k+1} - g_k \approx H_{k+1} \cdot (x_{k+1} - x_k)$$

or

$$y_k \approx H_{k+1} s_k \quad or \quad s_k = H_{k+1}^{-1} y_k$$

$$when \ y_k = g_{k+1} - g_k \ and \ s_k = x_{k+1} - x_k$$

We substitute B_{k+1} *for* H_{k+1} *and* D_{k+1} *for* H_{k+1}^{-1}, *and the equation is modified:*

$$y_k \approx B_{k+1} s_k \quad or \quad s_k = D_{k+1} y_k$$

Now we introduce the BFGS algorithm, and we set

$$B_{k+1} = B_k + \Delta B_k$$

$$\Delta B_k = \alpha u u^T + \beta v v^T$$

Then

$$y_k = B_k s_k + \left(\alpha u^T s_k\right) u + \left(\beta v^T s_k\right) v$$

If we have $\alpha u^T s_k = 1$ *and* $\beta v^T s_k = -1$, $u = y_k$ *and* $v = B_k s_k$, *α and β are deduced as:*

$$\alpha = \frac{1}{y_k^T s_k}, \beta = -\frac{1}{s_k^T B_k s_k}$$

So the representation of ΔB_k *will be:*

$$\Delta B_k = \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k}$$

BFGS Algorithm works by using B_k*:*

```
(1) Set $x_0$ and tolerance $\varepsilon$, $B_0=I$, $k:=0$
(2) Calculate descent direction $d_k = -B_k^{-1} g_k$
(3).Perform a line research to optimize $\lambda_k := \arg\min_{\lambda} f(x_n + \lambda s_n)$
(4) Set $s_k = \lambda_k d_k, x_{k+1} := x_k + s_k$
(5) If $\|g_{k+1}\| < \varepsilon$ , end loop
(6)Else: calculate $y_k = g_{k+1} - g_k$ and
        $B_{k+1} = B_k + \Delta B_k$
        k:=k+1,then go to 2nd step
```

BFGS Algorithm works by using D_k*:*

```
(1) Set $x_0$ and tolerance $\varepsilon$, $D_0=I$, $k:=0$
(2) Calculate descent direction $d_k = -D_k g_k$
(3).Perform a line research to optimize $\lambda_k := \arg\min_{\lambda} f(x_n + \lambda s_n)$
(4) Set $s_k = \lambda_k d_k, x_{k+1} := x_k + s_k$
(5) If $\|g_{k+1}\| < \varepsilon$ , end loop
(6)Else: calculate $y_k = g_{k+1} - g_k$ and
        $D_{k+1} = \left( I - \frac{s_k y_k^T}{y_k^T s_k} \right) D_k \left( I - \frac{y_k s_k^T}{y_k^T s_k} \right) + \frac{s_k s_k^T}{y_k^T s_k}$
        k:=k+1,then go to 2nd step
```

4.L-BFGS Algorithm

L-BFGS is also called Limited-memory BFGS that approximates BFGS algorithm using a limited amount of computer memory. We give a constant m according to our computer memory.

Set $\rho_k = \frac{1}{y_k^T s_k}, V_k = I - \rho_k y_k s_k^T$ *, and what we should do is to calculate* $D_k g_k$ *conveniently:*

```
(1) Set $\delta = \begin{cases} 0 & when\ k \le m \\ k-m & when\ k > m \end{cases}, L = \begin{cases} k & when\ k \le m \\ m & when\ k > m \end{cases}$ and $q_L = g_k$
(2) Backward Loop:
    for i=L-1:0
    {
        j= i+$\delta$;
        $\alpha_i = \rho_j s_j^T q_{i+1}$;
        $q_i = q_{i+1} - \alpha_i y_j$
    }
(3) Forward Loop:
    for i=0:L-1
    {
        j= i+$\delta$;
        $\beta_j = \rho_j y_j^T r_i$;
        $r_{i+1} = r_i + (\alpha_i - \beta_i) s_j$
    }
(4) $D_k g_k = r_L$
```

After having the value of $D_k g_k$ *, we can use BFGS algorithm to optimize our function.*

Regularization

Regularization is designed to address the problem of overfitting.

High bias or under-fitting is when the form of our hypothesis function maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features. eg. If we take $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$ then we are making an initial assumption that a linear model will fit the training data well and will be able to generalize but that may not be the case.

At the other extreme, overfitting or high variance is caused by a hypothesis function 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.

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

(1) Reduce the number of features:

a) Manually select which features to keep;

b) Use a model selection algorithm;

(2) Regularization

Keep all the features, but reduce the parameters θ_j. Regularization works well when we have a lot of slightly useful features.

1. Regularized Linear Regression

If we have overfitting from our hypothesis function, we can reduce the weight that some of the terms in our function carry by increasing their cost. Instead, we can modify our cost function:

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] \tag{3.1}$$

The λ, or lambda, is the regularization parameter. It determines how much the costs of our theta parameters are inflated. Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If lambda is chosen to be too large, it may smooth out the function too much and cause under-fitting.

Notice that the second term calculates without θ_0.

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

$$\begin{aligned}\theta_0 &:= \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right)x_0^i \\ \theta_j &:= \theta_j - \frac{\alpha}{m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right)x_j^i + \lambda\theta_j\right] \quad \left(j \in \{1,2,\dots,n\}\right)\end{aligned} \tag{3.2}$$

With some manipulation our update rule can also be represented as:

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

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

Set $\theta_0 = 0$, and the vectorized implements are as follows:

$$\begin{aligned}J(\theta) &= \frac{1}{2m}(h - y)^T(h - y) + \frac{\lambda}{2m}\theta^T\theta \\ \nabla J(\theta) &= \frac{1}{m}X^T(h - y) + \frac{\lambda}{m}\theta \\ \theta &:= \theta - \alpha\left(\frac{1}{m}X^T(h - y) + \frac{\lambda}{m}\theta\right)\end{aligned} \tag{3.3}$$

But $h = X\theta$ should be calculated before set $\theta_0 = 0$.

Now let's approach regularization using the alternate method of the non-iterative normal equation. To add in regularization, the equation is the same as our original, except that we add another term inside the parentheses and it can be deduced from (3.3) where $\nabla J(\theta) = 0$:

$$\theta = \left(X^T X + \lambda L\right)^{-1} X^T y \tag{3.4}$$

Where $L = \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}$ which should have dimension $(n+1)\times(n+1)$. Recall that if $m \le n$, then $X^T X$ is non-invertible. However, when we add the term λL, then $X^T X + \lambda L$ becomes invertible.

2. Regularized Logistic Regression

We can regularize logistic regression in a similar way that we regularize linear regression and modify our cost function by adding cost of parameters as follows:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^i \log\left(h_\theta\left(x^i\right)\right) + \left(1-y^i\right)\log\left(1-h_\theta\left(x^i\right)\right)\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{3.5}$$

Note that the second sum $\sum_{j=1}^{n}\theta_j^2$ means to explicitly exclude the bias term θ_0 by running from 1 to n.

Just like with linear regression, we will want to separately update θ_0 and the rest of the parameters because we do not want to regularize θ_0.

$$\begin{aligned} \theta_0 &:= \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right)x_0^i \\ \theta_j &:= \theta_j - \frac{\alpha}{m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y^i\right)x_j^i + \lambda\theta_j\right] \quad \left(j \in \{1, 2, \ldots, n\}\right) \end{aligned} \tag{3.6}$$

Equations can be vectorized as:

$$\begin{aligned} h &= g(X\theta) \\ J(\theta) &= \frac{1}{m}\left(-y^T \log(h) - (1-y)^T \log(1-h)\right) + \frac{\lambda}{2m}\theta^T\theta \ (Set\ \theta_0 = 0) \\ \theta &:= \theta - \alpha\frac{1}{m}\left(X^T(h-y) + \lambda\theta\right) \ (Set\ \theta_0 = 0) \end{aligned} \tag{3.7}$$

Neural Networks

Performing linear regression with a complex set of data with many features is very unwieldy. You have three features and create a hypothesis that includes all the quadratic terms like:

$$g\left(\theta_0+\theta_1x_1^2+\theta_2x_1x_2+\theta_3x_1x_3+\theta_4x_2^2+\theta_5x_2x_3+\theta_6x_3^2\right)$$

The exact way to calculate how many features of all polynomial terms is the combination function with repetition: $(n+r-1)!/\left[r!(n-1)!\right]$. The number of combination feature will be very large if n is large.

Neural networks offer an alternate way to perform machine learning when we have complex hypotheses with many features.

Neural networks are limited imitations of how our own brains work. They've had a big recent resurgence because of advances in computer hardware.

There is evidence that the brain uses only one "learning algorithm" for all its different functions. Scientists have tried cutting (in an animal brain) the connection between the cars and the auditory cortex and rewiring the optical nerve with the auditory cortex to find that the auditory cortex literally learns to see.

An ANN is based on a collection of connected units called artificial neurons. Each connection between neurons can transmit a signal to another neuron. The receiving neuron can process the signal(s) and then signal downstream neurons connected to it. Neurons may have state, generally represented by real numbers, typically between 0 and 1. Neurons have a weight that varies as learning proceeds, which can increase or decrease the strength of the signal that it sends downstream.

In general, a neural network includes three main layers: input layer, hidden layer and output layer. And the hidden layer may include a series of layers.

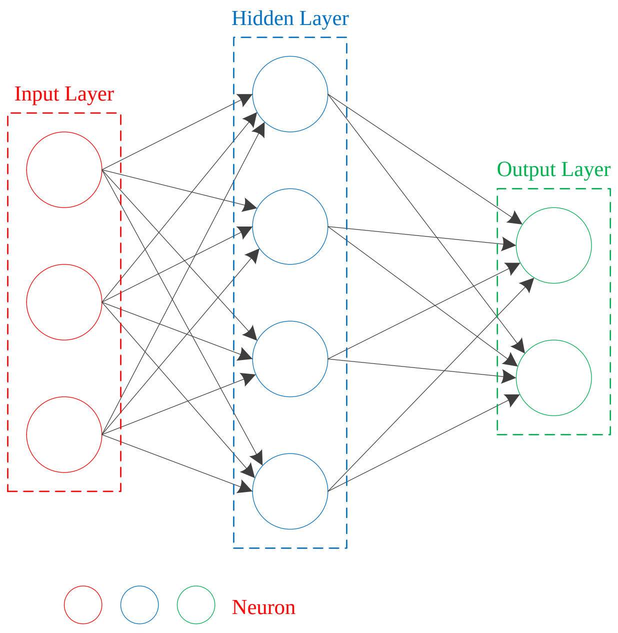


Figure 13 Neural networks

Notations for equations:

x_j — value of feature j

a_j^l — "activation" of unit j in l layer

Θ^l — matrix of weights controlling function mapping from layer l to layer l+1

$\Theta_{j,k}^l$ — weight from unit k in layer l to unit j in layer l+1

L — total number of layers in the network

s_l — number of units (not counting bias unit) in layer l

K — number of output units/classes

δ_j^l — “error” of node j in layer l

δ^l — “error” of nodes in layer l

Other notations have the same meaning as that before.

1. Model Representation

Activation Function: The first hidden layer can be regarded as the output of the input layer and so on. Every layer is output of its previous layer. So an activation function means how inputs work to outputs. In the following context, we choose the logistic function—sigmoid function that we use before as our activation function.

In neural networks, note that every layer should consider constant term (e.g. x_0), and it is always equal to 1. And our “theta” parameters are sometimes instead called “weights” in the neural networks model.

In fact, neural networks model can be represented as:

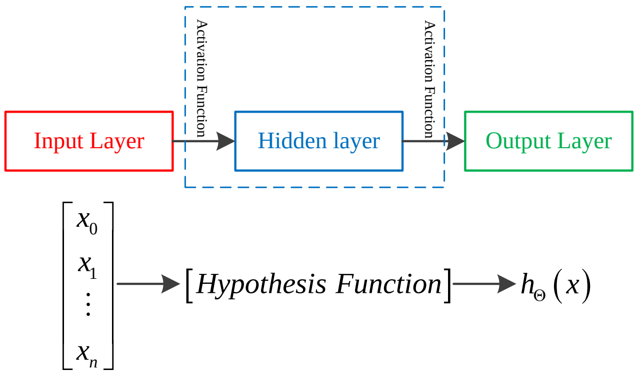
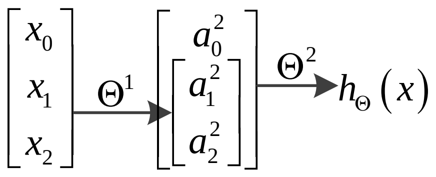


Figure 14 Neural networks model

But generally it is hard to get explicit hypothesis function for complex neural networks.

If we had one hidden layer, it would like visually something like ($x_0, a_0^2 = 1$):



The values for each of the “activation” nodes are obtained as follows (g is the logistic function):

$$a_1^2 = g\left(\Theta_{10}^1 x_0 + \Theta_{11}^1 x_1 + \Theta_{12}^1 x_2\right)$$
$$a_2^2 = g\left(\Theta_{20}^1 x_0 + \Theta_{21}^1 x_1 + \Theta_{22}^1 x_2\right)$$
$$h_\Theta(x) = a_1^3 = g\left(\Theta_{10}^2 a_0^2 + \Theta_{11}^2 a_1^2 + \Theta_{12}^2 a_2^2\right)$$

So we can get Θ^1 and Θ^2 matrix:

$$\Theta^1 = \begin{bmatrix} \Theta_{10}^1 & \Theta_{11}^1 & \Theta_{12}^1 \\ \Theta_{20}^1 & \Theta_{21}^1 & \Theta_{22}^1 \end{bmatrix}, \Theta^2 = \begin{bmatrix} \Theta_{10}^2 & \Theta_{11}^2 & \Theta_{12}^2 \end{bmatrix}$$

The dimensions of these matrices of weights is determined as: If network has s_j units in layer j and s_{j+1} units in layer j+1, then Θ^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”.

We define a new variable z_k^j that encompasses the parameters inside our g function. In our previous example if we replaced the variable z for all parameters we would get:

$$a_1^2 = g\left(z_1^2\right)$$
$$a_2^2 = g\left(z_2^2\right)$$
$$h_\Theta\left(x\right) = a_1^3 = g\left(z_1^3\right)$$

Setting $x = a^1$ so that it becomes more understandable:

$$x = \begin{bmatrix} x_0 & x_1 & \cdots & x_n \end{bmatrix}^T = \begin{bmatrix} a_0^1 & a_1^1 & \cdots & a_n^1 \end{bmatrix}^T = a^1$$

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

$$z_k^j = \Theta_{k,0}^{j-1} a_0^{j-1} + \Theta_{k,1}^{j-1} a_1^{j-1} + \cdots + \Theta_{k,s_{j-1}}^{j-1} a_{s_{j-1}}^{j-1}$$

The vector representation of *x* and z^j is:

$$a^{j-1} = \begin{bmatrix} a_0^{j-1} \\ a_1^{j-1} \\ \vdots \\ a_{s_{j-1}}^{j-1} \end{bmatrix}, z^j = \begin{bmatrix} z_1^j \\ z_2^j \\ \vdots \\ z_{s_j}^j \end{bmatrix}$$

We can rewrite the equation as:

$$z^j = \Theta^{j-1} a^{j-1} \tag{4.1}$$

Actually z^j starts from layer 2, so we can modify (4.1) as follows:

$$z^{j+1} = \Theta^j a^j \tag{4.2}$$

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

$$a^j = g\left(z^j\right) \tag{4.3}$$

Our function can be applied element-wise to our vector z^j. Note that add a bias unit (equal to 1) to layer *j* after we have computed a^j. In this last step, we are doing exactly the same thing as we did in logistic regression.

Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses.

2. Logical Operation

2.1. “AND”

A simple example of applying neural networks is by predicting x_1 AND x_2 which means is only true if both x_1 and x_2 are 1.

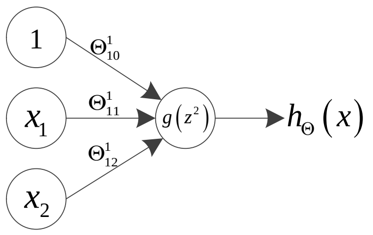


Figure 15 Logical Operation

Let’s set our first Θ matrix as: Θ^1 = [-30 20 20], and we get the results:

$$h_\Theta\left(x\right) = g\left(-30 + 20x_1 + 20x_2\right)$$
$$x_1 = 0, x_2 = 0 \Rightarrow g\left(-30\right) \approx 0$$
$$x_1 = 0, x_2 = 1 \Rightarrow g\left(-10\right) \approx 0$$
$$x_1 = 1, x_2 = 0 \Rightarrow g\left(-10\right) \approx 0$$
$$x_1 = 1, x_2 = 1 \Rightarrow g\left(10\right) \approx 1$$

So we have constructed one of the fundamental operations in computer by using a small neural network rather than using an actual AND gate.

2.2. "OR"

In Figure 15, if we set Θ matrix as: Θ^1 = [-10 20 20], we get the results:

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

So we have constructed "OR" operations by choosing right parameters.

2.3. "NOR"

In order to build "NOR" operation, we can set Θ^1 = [10 -20 -20], and the results are:

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

2.4. "XNOR"

A single layer is impossible to operate "XNOR", so we need another hidden layer to make it.

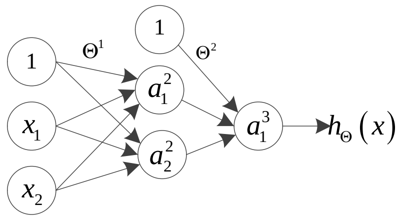


Figure 16 Logical Operation

Let's set Θ^1 = [-30 20 20; 10 -20 -20] and Θ^2 = [-10 20 20], and the results are:

$$\begin{aligned} &\begin{bmatrix} a_1^2 \\ a_2^2 \end{bmatrix} = \begin{bmatrix} g(-30 + 20x_1 + 20x_2) \\ g(10 - 20x_1 - 20x_2) \end{bmatrix} \\ &h_\Theta(x) = g(-10 + 20x_1 + 20x_2) \\ &x_1 = 0, x_2 = 0 \Rightarrow \begin{bmatrix} a_1^2 \\ a_2^2 \end{bmatrix} = \begin{bmatrix} g(-30) \approx 0 \\ g(10) \approx 1 \end{bmatrix} \Rightarrow g(10) \approx 1 \\ &x_1 = 0, x_2 = 1 \Rightarrow \begin{bmatrix} a_1^2 \\ a_2^2 \end{bmatrix} = \begin{bmatrix} g(-10) \approx 0 \\ g(-10) \approx 0 \end{bmatrix} \Rightarrow g(-10) \approx 0 \\ &x_1 = 1, x_2 = 0 \Rightarrow \begin{bmatrix} a_1^2 \\ a_2^2 \end{bmatrix} = \begin{bmatrix} g(-10) \approx 0 \\ g(-10) \approx 0 \end{bmatrix} \Rightarrow g(-10) \approx 0 \\ &x_1 = 1, x_2 = 1 \Rightarrow \begin{bmatrix} a_1^2 \\ a_2^2 \end{bmatrix} = \begin{bmatrix} g(10) \approx 1 \\ g(-30) \approx 0 \end{bmatrix} \Rightarrow g(10) \approx 1 \end{aligned}$$

3. Multiclass Classification

In logistical regression problems, we can solve multiclass classification by one-vs-all method. Now neural networks also can address these problems.

To classify data into multiple classes, we let our hypothesis function return a vector of values which means there are several outputs in output layer like Figure 13, that's:

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \rightarrow \begin{bmatrix} a_0^2 \\ a_1^2 \\ \vdots \\ a_l^2 \end{bmatrix} \rightarrow \begin{bmatrix} a_0^3 \\ a_1^3 \\ \vdots \\ a_m^3 \end{bmatrix} \rightarrow \cdots \rightarrow \begin{bmatrix} h_\Theta(x)_1 \\ h_\Theta(x)_2 \\ \vdots \\ h_\Theta(x)_s \end{bmatrix}$$

According to one-vs-all method, one element of outputs $h_\Theta(x)$ equals to 1 and others equal to 0. For example, there are for resulting classes if $s = 3$:

$$y^i = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Our final value of our hypothesis for a set of inputs will be one of the elements in y.

3.1. Cost Function

In neural networks, we may have many output nodes. We denote $h_\Theta(x)_k$ ($h_\Theta(x) \in \mathbb{R}^K$) as being a hypothesis that results in the kth output. Our cost function for neural networks is generalization of the one we used for logistic regression.

For neural networks, the cost function is slightly more complicated:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^i \log\left(h_\Theta\left(x^i\right)_k\right) + \left(1-y_k^i\right)\log\left(1-h_\Theta\left(x^i\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{j,i}^l\right)^2 \qquad (4.4)$$

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

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

3.2. Gradient Descent[1]

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

Our goal is to compute:

$$\min_\Theta J(\Theta)$$

It means using an optimal set of parameters in theta to minimize the cost function. We will look at the equations we use to compute the partial derivative of $J(\Theta)$:

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

In back propagation we are going to compute for every node δ_j^l. We define a_j^l is activation node j in layer l.

For the last layer, we can compute the vector of delta values with:

$$\delta^L = a^L - y \qquad (4.5)$$

Where L is our total number of layers and a^L is the vector of outputs of the activation units for the last layer.

To get the delta value of the layers before the last layer, we can use an equation that steps back from right to left:

$$\delta^l = \left(\Theta^l\right)^T \delta^{l+1} .* g'\left(z^l\right) \qquad (4.6)$$

As is known from Logistic Regression, the derivative of the logistic function $g(x)$ can be written out as $g'(x) = g(x).*(1-g(x))$. So the full back propagation equation for the inner nodes is then:

$$\delta^l = \left(\Theta^l\right)^T \delta^{l+1} .* a^l .* \left(1-a^l\right) \tag{4.7}$$

Notice that the multiplication sign is dot product in equation (4.6) and (4.7).

We can compute our partial derivative terms by multiplying our activation values and our error values for each training example *t*:

$$\frac{\partial J(\Theta)}{\partial \Theta_{i,j}^l} = \frac{1}{m}\sum_{t=1}^{m} a_j^{t,l} \delta_i^{t,l+1} \tag{4.8}$$

This however ignores regularization terms.

3.3. Backpropagation Algorithm

We take all these equations and put them together into a backpropagation algorithm:

```
(1) Set $\Delta_{i,j}^l := 0$ for all (l, i, j)
(2) for t = 1 : m
{
    Set $a^1 := x^t$ ;
    Perform forward propagation to compute $a^l \left(l = 2,3,\ldots,L\right)$ ;
    Using $y^t$ to compute $\delta^L = a^L - y^t$ ;
    Compute $\delta^{L-1}, \delta^{L-2}, \ldots, \delta^2$ by using $\delta^l = \left(\Theta^l\right)^T \delta^{l+1} .* a^l .* \left(1-a^l\right)$ ;
    Update $\Delta_{i,j}^l := \Delta_{i,j}^l + a_j^l \delta_i^{l+1}$ or with vectorization $\Delta^l := \Delta^l + \delta^{l+1}\left(a^l\right)^T$
    Compute $D_{i,j}^l := \begin{cases} \frac{1}{m}\Delta_{i,j}^l & (j=0) \\ \frac{1}{m}\left(\Delta_{i,j}^l + \lambda\Theta_{i,j}^l\right) & (j \neq 0) \end{cases}$
}
```

The capital-delta matrix is used as an "accumulator" to add up our values as we go along and eventually computer our partial derivative. In fact, the $D_{i,j}^l$ terms are the partial derivatives and the results we are looking for:

$$D_{i,j}^l = \frac{\partial J(\Theta)}{\partial \Theta_{i,j}^l} \tag{4.9}$$

This equation has considered the regularization term.

3.4. Gradient Checking

Gradient checking will assure that our backpropagation works as intended.

We can approximate the derivative of our cost function with:

$$\frac{\partial J(\Theta)}{\partial \Theta} \approx \frac{J(\Theta+\varepsilon) - J(\Theta-\varepsilon)}{2\varepsilon} \tag{4.10}$$

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

$$\frac{\partial J(\Theta)}{\partial \Theta_j} \approx \frac{J\left(\Theta_1,\cdots,\Theta_j+\varepsilon,\cdots,\Theta_n\right) - J\left(\Theta_1,\cdots,\Theta_j-\varepsilon,\cdots,\Theta_n\right)}{2\varepsilon} \tag{4.11}$$

Where ε is a small value.

We then check the approximate derivatives with the values in Section 3.3. We verify once that backpropagation algorithm is correct, then you don't need to compute them again because its code is very slow.

3.5. Random Initialization

Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly.

Instead we can randomly initialize our weights. Initialize each $\Theta_{i,j}^{l}$ to a random value between [-ε, ε] (ε<1):

$$\begin{aligned}\varepsilon &= \frac{\sqrt{6}}{\sqrt{Loutput + Linput}} \\ \Theta^{l} &= 2\varepsilon \cdot rand\left(Loutput, Linput+1\right) - \varepsilon\end{aligned} \tag{4.12}$$

3.6. Summary

Above all, when we train a neural network, we need:

(1) Randomly initialize the weights

(2) Implement forward propagation to get *h*

(3) Implement the cost function

(4) Implement backpropagation to compute partial derivatives

(5) Use gradient checking to confirm that backpropagation works. Then disable the gradient checking.

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

Annotation

1. Derivatives Used in Backpropagation

Without considering regularization term, the cost function will be:

$$J\left(\Theta\right) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^i \log\left(h_\Theta\left(x^i\right)_k\right) + \left(1-y_k^i\right)\log\left(1-h_\Theta\left(x^i\right)_k\right)\right]$$

A vectorized version is:

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

Now we ignore the summation natation in above equation, and set:

$$J_1\left(\Theta\right) = -\left(y\right)^T \log\left(h_\Theta\left(x\right)\right) - \left(1-y\right)^T \log\left(1-h_\Theta\left(x\right)\right)$$

We know that:

$$\frac{\partial J_1\left(\Theta\right)}{\partial \Theta^{L-1}} = \frac{\partial J_1\left(\Theta\right)}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial \Theta^{L-1}}$$

Now we compute:

$$\begin{aligned}\frac{\partial J_1\left(\Theta\right)}{\partial a^L} &= -y^T \cdot * \frac{1}{a^L} + \left(1-y\right)^T \cdot * \frac{1}{1-a^T} \\ \frac{\partial a^L}{\partial z^L} &= a^L \cdot * \left(1-a^L\right)\end{aligned}$$

Then we get the equation:

$$\begin{aligned}\frac{\partial J_1\left(\Theta\right)}{\partial a^L}\frac{\partial a^L}{\partial z^L} &= \left(-y \cdot * \frac{1}{a^L} + \left(1-y\right) \cdot * \frac{1}{1-a^T}\right) \cdot * \left(a^L \cdot * \left(1-a^L\right)\right) \\ &= a^L - y\end{aligned}$$

Which equals to (4.5) so that we conclude that:

$$\delta^L = \frac{\partial J_1\left(\Theta\right)}{\partial a^L}\frac{\partial a^L}{\partial z^L}$$

Then we modify the equation as:

$$\frac{\partial J_1\left(\Theta\right)}{\partial \Theta^{L-1}} = \delta^L \frac{\partial z^L}{\partial \Theta^{L-1}}$$

So we can also deduce equations:

$$\delta^{L-1} = \frac{\partial J_1(\Theta)}{\partial a^L} \frac{\partial a^L}{\partial z^L} \frac{\partial z^L}{\partial a^{L-1}} \frac{\partial a^{L-1}}{\partial z^{L-1}} = \delta^L \frac{\partial z^L}{\partial a^{L-1}} \frac{\partial a^{L-1}}{\partial z^{L-1}}$$

$$\frac{\partial J_1(\Theta)}{\partial \Theta^{L-2}} = \frac{\partial J_1(\Theta)}{\partial a^L} \frac{\partial a^L}{\partial z^L} \frac{\partial z^L}{\partial a^{L-1}} \frac{\partial a^{L-1}}{\partial z^{L-1}} \frac{\partial z^{L-1}}{\partial \Theta^{L-2}} = \delta^{L-1} \frac{\partial z^{L-1}}{\partial \Theta^{L-2}}$$

Given $z^L = \Theta^{L-1} a^{L-1}$ *in the last layer, the partial derivative is:*

$$\frac{\partial z^L}{\partial \Theta^{L-1}} = a^{L-1}$$

Put it together for the output layer:

$$\frac{\partial J_1(\Theta)}{\partial \Theta^{L-1}} = \delta^L a^{L-1}$$

$$\frac{\partial J_1(\Theta)}{\partial \Theta^{L-2}} = \delta^{L-1} a^{L-2}$$

Summarily, we get the derivatives of the cost function.

Support Vector Machines (SVMs)

In machine learning, support vector machines (also called support vector networks) are supervised learning models with associated learning algorithms that analyzed data used for classification and regression analysis. 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[1], implicitly mapping their inputs into high-dimensional feature spaces.

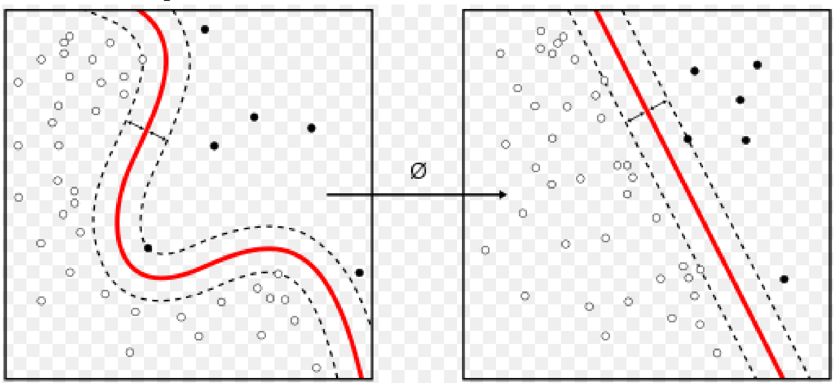


Figure 17 Kernel Trick

Linear classifier: In the case of support vector machines, a data point is viewed as a p-dimensional vector, and we want to know whether we can separate such points with a (p-1)-dimensional hyperplane. There are many hyperplane that might classify the data. One reasonable choice as the best hyperplane is the one that represents the largest separation, or margin, between the two classes. So we choose the hyperplane so that the distance from it to the nearest data point on each side is maximized. If such a hyperplane exists, it is known as the maximum-margin hyperplane and the linear classifier it defines is known as a maximum margin classifier. Samples on the margin are called the support vectors.

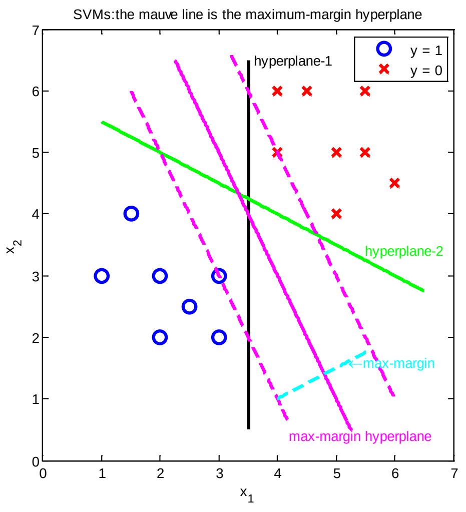


Figure 18 Max-margin Hyperplane

1. Cost Function

Recall that in logistic regression, we use the following rules:

If $y = 1$, then $h_\theta(x) \approx 1$ and $\theta^T x >> 0$

If $y = 0$, then $h_\theta(x) \approx 0$ and $\theta^T x << 0$

And the cost function for (non-regularized) logistic regression:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^i \log\left(h_\theta\left(x^i\right)\right) + \left(1-y^i\right)\log\left(1-h_\theta\left(x^i\right)\right)\right)$$
$$= \frac{1}{m}\sum_{i=1}^{m}\left(-y^i \log\left(\frac{1}{1+e^{-\theta^T x^i}}\right) - \left(1-y^i\right)\log\left(1-\frac{1}{1+e^{-\theta^T x^i}}\right)\right) \quad (5.1)$$

We now plot functions $y = -\log\left(\frac{1}{1+e^{-z}}\right)$ and $y = -\log\left(1-\frac{1}{1+e^{-z}}\right)$ as follows:

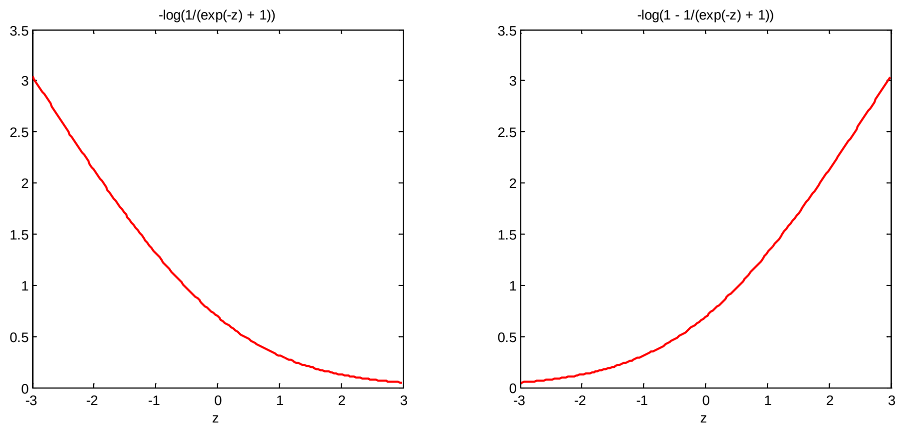


Figure 19 Curves of Both Functions

To make a support vector machine, we will modify the first term of the cost function so that when $z = \theta^T x$ is greater than 1, it outputs 0. Furthermore, for values of z less than 1, we shall use a straight decreasing line instead of the sigmoid curve. (In this literature, this is called a hinge loss function). Similarly, we modify the second term of the cost function so that when z is less than -1, it outputs 0. We also modify it so that for values of z greater than -1, we use a straight increasing line instead of the sigmoid curve.

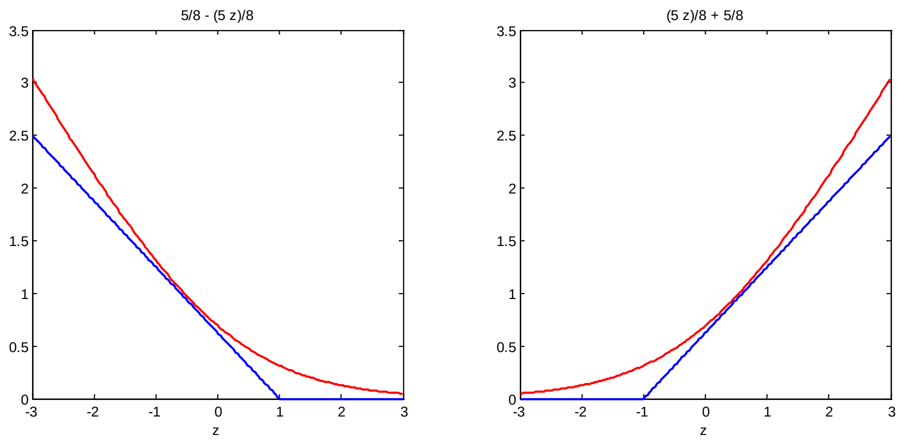


Figure 20 Blue curves represent modified functions

We shall denote these as $\text{cost}_1(z)$ when $y = 1$ and $\text{cost}_0(z)$ when $y = 0$, and we may define them as follows (where k is an arbitrary constant defining the magnitude of the slope of the line):

$$z = \theta^T x$$
$$\text{cost}_0(z) = \max\left(0, k(1+z)\right) \quad (5.2)$$
$$\text{cost}_1(z) = \max\left(0, k(1-z)\right)$$

We transform the cost function (5.1) for support vector machines by substituting $\text{cost}_1(z)$ and $\text{cost}_0(z)$:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[y^i \text{cost}_1\left(\theta^T x^i\right) + \left(1-y^i\right)\text{cost}_0\left(\theta^T x^i\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \quad (5.3)$$

Because m is a constant, it does not affect our optimization. Then we optimize this a bit by multiplying this by *m*:

$$J(\theta) = \sum_{i=1}^{m}\left[y^i \mathrm{cost}_1\left(\theta^T x^i\right) + \left(1-y^i\right)\mathrm{cost}_0\left(\theta^T x^i\right)\right] + \frac{\lambda}{2}\sum_{j=1}^{n}\theta_j^2 \qquad (5.4)$$

Furthermore, convention dictates that we regularize using a factor *C*, instead of λ, like so:

$$J(\theta) = C\sum_{i=1}^{m}\left[y^i \mathrm{cost}_1\left(\theta^T x^i\right) + \left(1-y^i\right)\mathrm{cost}_0\left(\theta^T x^i\right)\right] + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 \qquad (5.5)$$

This is equivalent to multiplying the equation by $C = \frac{1}{\lambda}$, and thus results in the same values when optimized. Now when we wish to regularize more, we decrease *C*; when we wish to regularize less, we increase *C*.

A useful way to think about Support Vector Machines is to think of them as Large-Margin Classifiers.

If $y = 1$, then $\theta^T x \geq 1$

If $y = 0$, then $\theta^T x \leq -1$

Now when we set our constant *C* to a very large value, our optimizing function will constrain θ such that the summation of the cost function equals 0. We impose the following constraints on θ:

$\theta^T x \geq 1$ if $y = 1$ and $\theta^T x \leq -1$ if $y = 0$

If *C* is very large, we must choose θ parameters such that (My opinion: if the above condition is absolutely satisfied, the below equation always equals 0):

$$\sum_{i=1}^{m}\left[y^i \mathrm{cost}_1\left(\theta^T x^i\right) + \left(1-y^i\right)\mathrm{cost}_0\left(\theta^T x^i\right)\right] = 0$$

So the cost function is reduced to:

$$J(\theta) = \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 \qquad (5.6)$$

2. Maximum-Margin Classifier

In SVMs problems, we generally transform $y = \{0, 1\}$ to $y = \{-1, 1\}$ to simplify our problem. And we set (*w* is the normal vector of hyperplane).

$$\theta^T x = f(x) = w^T x + b \qquad (5.7)$$

The hyperplane is defined as $w^T x + b = 0$, and $\left|w^T x + b\right|$ is the representation of relative distance between *x* and the hyperplane. If the sign of $w^T x + b$ is the same as the sign of *y*, it means *x* is classified correctly, so the sign of $y(w^T x + b)$ can be used to decide whether examples are classified correctly.

We define the functional margin as:

$$\hat{\gamma} = y\left(w^T x + b\right) \qquad (5.8)$$

Then define the functional margin between the hyperplane and training data set *T* as the minimum relative distance of all examples in *T* to the hyperplane:

$$\hat{\gamma} = \min_{i=1,\ldots,m} \hat{\gamma}^i \qquad (5.9)$$

However, the functional margin will change if the values of *w* and *b* change, so we define another margin type—geometrical margin. In Figure 21 the sample *x* is a support vector.

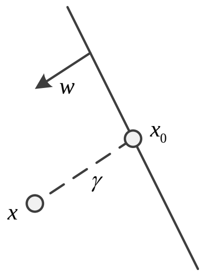


Figure 21 Geometrical margin

We know that

$$x = x_0 + \gamma \frac{w}{\|w\|} \qquad f\left(x_0\right) = w^T x_0 + b = 0 \tag{5.10}$$

And we get the result:

$$\gamma = \frac{w^T x + b}{\|w\|} = \frac{f\left(x\right)}{\|w\|} \tag{5.11}$$

The geometrical margin is defined as:

$$\tilde{\gamma} = y\gamma = \frac{\hat{\gamma}}{\|w\|} \tag{5.12}$$

Because *x* is a support vector, we know that *f*(*x*) = 1 and *y* = 1 or *f*(*x*) = -1 and *y* = -1, then $\hat{\gamma} = 1$. The equation (5.12) is modified as:

$$\tilde{\gamma} = \frac{1}{\|w\|} \tag{5.13}$$

We hope maximize $\tilde{\gamma}$ in order to classify data more confidently.

The optimized model is represented as:

$$\max\left(\frac{1}{\|w\|}\right) \qquad s.t.\ y^i\left(w^T x^i + b\right) \geq 1\left(i = 1, \ldots, m\right) \tag{5.14}$$

Which equals to minimizing the cost function:

$$\min \frac{1}{2}\|w\|^2 \qquad s.t.\ y^i\left(w^T x^i + b\right) \geq 1\left(i = 1, \ldots, m\right) \tag{5.15}$$

Which is a QP problem (Quadratic Programming).

But it can be transformed into a Lagrange Dual Problem: Lagrange function is created by the previous objective function subtracting the constraints terms with Lagrange multiplier:

$$L\left(w, b; \alpha\right) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{m} \alpha^i \left[y^i\left(w^T x^i + b\right) - 1\right]\left(\alpha^i \geq 0\right) \tag{5.16}$$

For a particular value of *x*, if the constraints are all satisfied then the best we can do is to set α^i = 0, so $L\left(w, b; \alpha\right) = \frac{1}{2}\|w\|^2$. If any of the constraints are not satisfied, then we can make $L\left(w, b; \alpha\right) \to \infty$ by taking $\alpha^i \to \infty$. So we have

$$\max_{\alpha} L\left(w, b; \alpha\right) = J\left(w\right) \tag{5.17}$$

Remember that we want to minimize $J\left(w\right)$, which we now know means finding:

$$\min_{w,b} \max_{\alpha} L\left(w, b; \alpha\right) \tag{5.18}$$

And the primal problem is transformed into dual problem:

$$\max_{\alpha} \min_{w,b} L\left(w, b; \alpha\right) \tag{5.19}$$

So we have

$$\frac{\partial L}{\partial w} = 0 \Rightarrow w = \sum_{i=1}^{m} \alpha^i y^i x^i \qquad \frac{\partial L}{\partial b} = 0 \Rightarrow \sum_{i=1}^{m} \alpha^i y^i = 0 \tag{5.20}$$

Substitute the equation to (5.16), we have:

$$\max_{\alpha} L\left(w, b; \alpha\right) = \max_{\alpha}\left[\sum_{i=1}^{m} \alpha^i - \frac{1}{2}\sum_{i,j=1}^{m} \alpha^i \alpha^j y^i y^j \left(x^i\right)^T x^j\right] \tag{5.21}$$

3. Kernels

The kernel trick avoids the explicit mapping that is needed to get linear learning algorithms to learn a nonlinear function or decision boundary.

For all x and x' in the space Ω, a kernel function $k(x,x')$ can be expressed as an inner product in another space V. I conclude its main idea as:

$$\begin{aligned} &x_\Omega \to \varphi(x)_V \\ &x'_\Omega \to \varphi(x')_V \\ &k(x,x') = \langle \varphi(x), \varphi(x') \rangle_V \end{aligned} \tag{5.22}$$

Where" $\to$ "means mapping, and $\langle a,b \rangle$ means dot product between a and b. In general, the dimension of V is higher than that of Ω, which means that the dimension of the kernel function is more higher, resulting in the opinion that kernel trick is a mapping from low dimension to high dimension (This explanation looks reasonable but not exact, and I think that it may be explained that the mapping function create more features to facilitate nonlinear problems).

We substitute w in (5.20) into (5.7):

$$f(x) = \left(\sum_{i=1}^{m} \alpha^i y^i x^i \right)^T x + b = \sum_{i=1}^{m} \alpha^i y^i \langle x^i, x \rangle + b \tag{5.23}$$

If it is impossible to linear classification in the space Ω, we need to map to the space V, which means that we have:

$$\begin{aligned} f(x) &= \sum_{i=1}^{m} \alpha^i y^i \langle \varphi(x^i), \varphi(x) \rangle + b \\ &= \sum_{i=1}^{m} \alpha^i y^i k(x^i, x) + b \end{aligned} \tag{5.24}$$

Some common kernels include:

(1) Polynomial (homogeneous): $k(x^i, x^j) = (x^i \cdot x^j)^d$

(2) Polynomial (inhomogeneous): $k(x^i, x^j) = (x^i \cdot x^j + 1)^d$

(3) Gaussian radial basis function: $k(x^i, x^j) = \exp\left(-\xi \left\| x_i - x_j \right\|^2\right) \; \left(\xi = 1/2\sigma^2\right)$

(4) Hyperbolic tangent: $k(x^i, x^j) = \tanh(\kappa x_i \cdot x_j + c) \; (\kappa > 0, c < 0)$

In this chapter, we use the Gaussian function as our kernel function. But we should remember that it is really important and necessary to choose the right kernel function to make a better classification because they have different effect to the same classification problems.

3.1. Gaussian Kernel

Given x, compute new feature depending on proximity to landmarks l^1, l^2, l^3.

To do this, we find the "similarity" of x and some landmark l^i:

$$f_i = \exp\left(-\frac{\left\| x - l^i \right\|^2}{2\sigma^2} \right) = \exp\left(-\sum_{j=1}^{n} \left(x_j - l_j^i \right)^2 / 2\sigma^2 \right) \tag{5.25}$$

There are a couple properties of the similarity function:

(1) If $x \approx l^i$, then $f_i \approx 1$; (2) If x is far from l^i, then $f_i \approx 0$.

Each landmark gives us the features in our hypothesis:

$$\begin{aligned}
&l^1 \to f_1\\
&l^2 \to f_2\\
&l^3 \to f_3\\
&\dots\\
&h_\theta(x) = \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 + \cdots
\end{aligned} \tag{5.26}$$

Combined with looking at the values inside θ, we can choose these landmarks to get the general shape of the decision boundary.

One way to get the landmarks is to put them in the exact same locations as all the training examples. This gives us m landmarks, with one landmark per training example:

$$l^1 = x^1, l^2 = x^2, \cdots, l^m = x^m$$

We may also set $f_0 = 1$ to correspond with θ_0. Thus we have the kernel trick of the given training example x^i:

$$x^i \to f^i = \begin{bmatrix} f_1^i = k\left(x^i, l^1\right) \\ f_2^i = k\left(x^i, l^2\right) \\ \vdots \\ f_m^i = k\left(x^i, l^m\right) \end{bmatrix} \tag{5.27}$$

Now to get the parameters $\Theta \in \mathbb{R}^{m+1}$ we can use the SVM minimization algorithm[1] with f^i substituted in for x^i:

$$J(\theta) = C\sum_{i=1}^{m}\left[y^i \mathrm{cost}_1\left(\Theta^T f^i\right) + \left(1-y^i\right)\mathrm{cost}_0\left(\Theta^T f^i\right)\right] + \frac{1}{2}\sum_{j=1}^{m}\Theta_j^2 \tag{5.28}$$

4. Parameters C and σ^2

Choosing C (recall that $C = 1 / \lambda$):

(1) If C is large, then we get higher variance/lower bias;

(2) If C is small, then we get lower variance/higher bias.

The other parameter we must choose is σ^2 from the Gaussian Kernel function:

(1) With a larger σ^2, the features f^i vary more smoothly, causing higher bias and lower variance.

(2) With a small σ^2, the features f^i vary less smoothly, causing lower bias and higher variance.

For example, $l^1 = [3,5]^T$, and $f_1 = \exp\left(-\left\|x - l^1\right\|^2 / 2\sigma^2\right)$, we get the results as following figures:

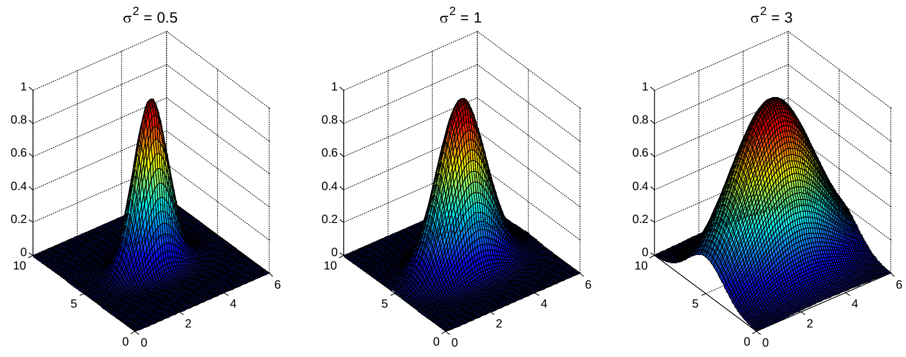


Figure 22 f^i vary with σ^2

In addition, if n is large (relative to m), then use logistic regression, or SVM without a kernel (the "linear kernel"); If n is small and m is intermediate, then use SVM with a Gaussian Kernel; If n is small and m is large, then manually create more features, then use logistic regression or SVM without a kernel.

In this first case, we don't have enough examples to need a complicated polynomial hypothesis.

In the second example, we have enough examples that we may need a complex non-linear hypothesis. In the last case, we want to increase our features so that logistic regression becomes applicable.

5. SVM algorithm

Now we give the SVM algorithm for classification:

```
(1) If necessary, perform feature scaling before optimization;
(2) Choose linear kernel or Gaussian kernel, maybe other kernels;
(3) Map initial data to the new space;
(4) Give a series of $C$ values as well as $\sigma^2$; Initialize parameters
(5) for C
    {
        for $\sigma^2$
        {
            Perform SVM minimization algorithm;
        }
    }
(6) Find $C$, $\sigma^2$ and corresponding parameters (We will discuss it in
    later chapter);
(7) Inversely transformation to get the boundary of data.
```

Annotation

1. SVM minimization algorithm

In this section, we just talk about SMO(Sequential minimal optimization) invented by John Platt in 1988 at Microsoft Research, which generated much excitement in the SVM community, as previously available methods for SVM training were much more complex and required expensive third-party QP solvers.

You had better to read the article "Sequential Minimal Optimization A Fast Algorithm for Training Support Vector Machines" written by John Platt to understand the theory of SMO: "Training a support vector machine requires the solution of a very large quadratic programming (QP) optimization problem. SMO breaks this large QP problem into a series of smallest possible QP problems. These small QP problems are solved analytically, which avoids using a time-consuming numerical QP optimization as an inner loop."

For linear classification in original space, we have ($\Psi(w,b;\alpha) = -L(w,b;\alpha)$):

$$\begin{array}{ll} \min \frac{1}{2}\|w\|^2 & \\ s.t.\ y^i\left(w^T x^i + b\right) \ge 1 (i = 1,\dots,m) \end{array} \Rightarrow \begin{array}{l} \min\limits_{\alpha} \Psi(w,b;\alpha) = \min\limits_{\alpha}\left[\frac{1}{2}\sum_{i,j=1}^{m} y^i y^j \left\langle x^i, x^j \right\rangle \alpha^i \alpha^j - \sum_{i=1}^{m} \alpha^i\right] \\ s.t.\ \ \alpha^i \ge 0, \sum_{i=1}^{m} y^i \alpha^i = 0 \end{array}$$

However, in 1995, Cortes and Vapnik ("Support Vector Networks") suggested a modification to the original optimization statement which allows, but penalizes, the failure of an example to reach the correct margin. That modification is:

$$\min \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i$$
$$s.t.\ y^i\left(w^T x^i + b\right) \ge 1 - \xi_i\ (i = 1,\dots,m)$$

Where ξ_i are slack variables that permit margin failure and C is a parameter which trades off wide margin with a small number of margin failures. When this new optimization problem is transformed into dual form, it simply changes the constraint into a box constraint:

$$\min_\alpha \Psi\left(w,b;\alpha\right) = \min_\alpha\left[\frac{1}{2}\sum_{i,j=1}^{m} y^i y^j \left\langle x^i, x^j \right\rangle \alpha^i \alpha^j - \sum_{i=1}^{m}\alpha^i\right]$$

$$s.t.\quad 0 \le \alpha^i \le \mathrm{C}, \sum_{i=1}^{m} y^i \alpha^i = 0$$

To generalize to non-linear classifiers, we have the dual objective function:

$$\min_\alpha \Psi\left(w,b;\alpha\right) = \min_\alpha\left[\frac{1}{2}\sum_{i,j=1}^{m} y^i y^j k\left(x^i, x^j \right) \alpha^i \alpha^j - \sum_{i=1}^{m}\alpha^i\right]$$

$$s.t.\quad 0 \le \alpha^i \le \mathrm{C}, \sum_{i=1}^{m} y^i \alpha^i = 0$$

The KKT conditions for the QP problem are particularly simple:

$$\alpha^i = 0 \Leftrightarrow y^i f^i \ge 1$$

$$0 < \alpha^i < C \Leftrightarrow y^i f^i = 1$$

$$\alpha^i = C \Leftrightarrow y^i f^i \le 1$$

SMO chooses two Langrange multipliers to jointly optimize, finds the optimal values for these multipliers at every step,treats other multipliers as constants. and we have

$$\alpha^1 y^1 + \alpha^2 y^2 = -\sum_{i=3}^{m} \alpha^i y^i = \zeta \quad or$$

$$\alpha^1 = \left(\zeta - \alpha^2 y^2\right) y^1$$

We substitute this equation to the objective function:

$$\Psi = \frac{1}{2}k_{11}\left(\zeta - \alpha^2 y^2\right)^2 + \frac{1}{2}k_{22}\left(\alpha^2\right)^2 + y^2 k_{12}\left(\zeta - \alpha^2 y^2\right)\alpha^2$$
$$-\left(\zeta - \alpha^2 y^2\right) y^1 - \alpha^2 + v^1\left(\zeta - \alpha^2 y^2\right) + y^2 v^2 \alpha^2 + \Sigma$$

Where $v^i = \sum_{j=3}^{m} \alpha^j y^j k\left(x^i, x^j\right),\ i = 1,2$, $k_{ij} = k\left(x^i, x^j\right)$ *and* Σ *is a constant. So* Ψ *is the minimum when its derivative is zero:*

$$\frac{\partial \Psi}{\partial \alpha^2} = \left(k_{11} + k_{22} - 2k_{12}\right)\alpha^2 - k_{11}\zeta y^2 + k_{12}\zeta y^2 + y^1 y^2 - 1 - v^1 y^2 + v^2 y^2 = 0$$

Before updating new values of α^1, α^2 *, we have:*

$$\zeta = \alpha^1 y^1 + \alpha^2 y^2$$

Because $v^i = \sum_{j=3}^{m} \alpha^j y^j k\left(x^i, x^j\right),\ i = 1,2$ *and* $f\left(x\right) = \sum_{i=1}^{m} \alpha^i y^i k\left(x^i, x\right) + b$ *, so the representation of* v^1 *and* v^2*:*

$$v^1 = f\left(x^1\right) - \sum_{j=1}^{2} y^1 \alpha^1 k_{1j} - b, v^2 = f\left(x^2\right) - \sum_{j=1}^{2} y^1 \alpha^1 k_{j2} - b$$

Then we define $E^i = f\left(x^i\right) - y^i$ *and* $\eta = k_{11} + k_{22} - 2k_{12}$ *, in this case, SMO computes the minimum along the direction of the constraint:*

$$\alpha^2_{new} = \alpha^2 + \frac{y^2\left(E^1 - E^2\right)}{\eta}$$

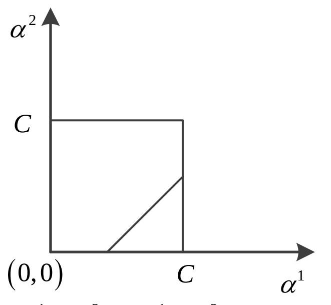
We have known that $y^i = \left\{-1,1\right\}$ *, so the two Lagrange multipliers must fulfill the constraints as:*

If $y^1 \ne y^2$ *, then the following bounds apply to* α^2 *:*

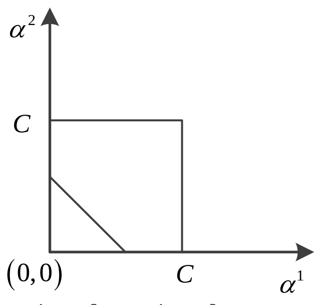
$$L = \max\left(0, \alpha^2 - \alpha^1\right), H = \min\left(C, C + \alpha^2 - \alpha^1\right)$$

If $y^1 = y^2$ *, then the following bounds apply to* α^2 *:*

$$L = \max\left(0, \alpha^2 + \alpha^1 - C\right), H = \min\left(C, \alpha^2 + \alpha^1\right)$$



$y^1 \neq y^2 \Rightarrow \alpha^1 - \alpha^2 = \zeta$



$y^1 = y^2 \Rightarrow \alpha^1 + \alpha^2 = \zeta$

The constrained minimum is found by clipping the unconstrained minimum to the ends of the line segment($s = y^1 y^2$ *):*

$$\alpha_{new}^2 = \begin{cases} H & \alpha_{new}^2 \geq H \\ \alpha_{new}^2 & L \leq \alpha_{new}^2 \leq H \\ L & \alpha_{new}^2 \leq L \end{cases}, \qquad \alpha_{new}^1 = \alpha^1 + s\left(\alpha^2 - \alpha_{new}^2\right)$$

However, under unusual circumstances, we need $\alpha_{new}^2 = L$ *or H when* $\eta<0$ *if the kernel k does not obey Mercer's condition* or $\eta=0$ *if more than one training example has the same input. In addition, the second-order derivative of the objective function is η, so the objective function is convex and the minimum value is calculated at boundary point if* $\eta<0$*, or the objective function changes monotonously and the minimum value is calculated at boundary point too. In this case, we have:*

$$\begin{aligned} &\alpha_{new}^2 = L \\ &\alpha_{new}^1 = L_1 \\ &L_1 = \alpha^1 + s\left(\alpha^2 - \alpha_{new}^2\right) \\ &F_1 = y^1\left(E^1 - b\right) - s\alpha^2 k_{12} - \alpha^1 k_{11} \\ &F_2 = y^2\left(E^2 - b\right) - s\alpha^1 k_{12} - \alpha^2 k_{22} \\ &\Psi_L = L_1 F_1 + L F_2 + \frac{1}{2} L_1^2 k_{11} + \frac{1}{2} L^2 k_{22} + sLL_1 k_{12} \end{aligned}$$

or

$$\begin{aligned} &\alpha_{new}^2 = H \\ &\alpha_{new}^1 = H_1 \\ &H_1 = \alpha^1 + s\left(\alpha^2 - \alpha_{new}^2\right) \\ &F_1 = y^1\left(E^1 - b\right) - s\alpha^2 k_{12} - \alpha^1 k_{11} \\ &F_2 = y^2\left(E^2 - b\right) - s\alpha^1 k_{12} - \alpha^2 k_{22} \\ &\Psi_H = H_1 F_1 + H F_2 + \frac{1}{2} H_1^2 k_{11} + \frac{1}{2} H^2 k_{22} + sHH_1 k_{12} \end{aligned}$$

We set α_{new}^2 *as L if* $\Psi_L \leq \Psi_H$ *, or* α_{new}^2 *equals H.*

Now we update the value of b:

(1) If $0 < \alpha_{new}^1 < C$ *, then* $b_{new}^1 = -E^1 - y^1 k_{11}\left(\alpha_{new}^1 - \alpha^1\right) - y^2 k_{21}\left(\alpha_{new}^2 - \alpha^2\right) + b$ *;*

(2) If $0 < \alpha_{new}^2 < C$ *, then* $b_{new}^2 = -E^2 - y^1 k_{12}\left(\alpha_{new}^1 - \alpha^1\right) - y^2 k_{22}\left(\alpha_{new}^2 - \alpha^2\right) + b$ *;*

(3) If $0 < \alpha_{new}^i < C, i = 1,2$ *, then* $b_{new}^1 = b_{new}^2$ *;*

(4) Else, then $b_{new} = \left(b_{new}^1 + b_{new}^2\right)/2$.

The weight vector update is easy, due to the linearity of the SVM:

$$w_{new} = w + y^1\left(\alpha_{new}^1 - \alpha^1\right)x^1 + y^2\left(\alpha_{new}^2 - \alpha^2\right)x^2$$

How to choose α^1 *and* α^2 *? Platt suggested heuristics for choosing which multiplier to optimize. He created two separate choice heuristics: one for the first Lagrange multiplier and one for the second. The choice of the first heuristic provides the outer loop of the SMO algorithm. The outer loop first iterates over the entire training set, determining whether each example violates the KKT conditions. If an example violates the KKT conditions, it is then eligible for optimization. Once a first Lagrange multiplier is chosen, SMO chooses the second Lagrange multiplier to maximize the size the step taken during joint optimization which depends on the value of* $\left|E^1 - E^2\right|$*. If* E^1 *is positive, SMO chooses an example with minimum error* E^1*. If* E^1 *is negative, SMO chooses an example with maximum error* E^2*.*

Clustering

Clustering or cluster analysis is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups. Clustering is one of unsupervised learning tasks which use an unlabeled training set rather than a labeled one. In other words, we don’t have the vector *y* of expected results, we only have a dataset of features where we can find structure.

Clustering is good for:

(1) Market segmentation

(2) Social network analysis

(3) Organizing computer clusters

(4) Astronomical data analysis

1. K-Means Clustering

The K-Means Algorithm is the most popular and widely used algorithm for automatically grouping data into coherent subsets.

Notations for equations:

x^i — the *i*th training example

x^i_j — the value of feature *j* of x^i

S_k — the *k*th clustering set

x^{ki} — the *i*th feature x^i in S_k

μ_k — the centroid point of points in S_k

K — the number of centroid points

c^i — the index of the centroid that has minimal distance to x^i

$\mu_{c(i)}$ — cluster centroid of cluster to which example x^i has been assigned

1.1. Cost Function

Given a set of training examples $(x_1, x_2, \ldots, x_m)$ (In this case, examples are not labeled)where each example is a *n*-dimensional real vector, *K*-Means clusters aims to partition the *m* examples into *K* sets $S = \{S_1, S_2, \ldots, S_K\}$ so as to minimize the within-cluster sum of squares. Formally the objective is to find the minimum of the cost function:

$$J = \frac{1}{m}\sum_{k=1}^{K}\sum_{x \in S_i}\left\|x - \mu_k\right\|^2 \tag{6.1}$$

Which is equivalent to minimizing c^i :

$$c^i = \min_k \left\|x^i - \mu_k\right\|^2 \tag{6.2}$$

The cost function can also be defined as follows:

$$J\left(c^1, \ldots, c^m; \mu_1, \ldots, \mu_K\right) = \frac{1}{m}\sum_{i=1}^{m}\left\|x^i - \mu_{c(i)}\right\|^2 \tag{6.3}$$

Our optimization objective is to minimize all our parameters using the above cost function:

$$\min_{c,\mu} J\left(c; \mu\right) \tag{6.4}$$

That is, we are finding all values in set *S*, representing all our clusters, and μ, representing all our centroids that will minimize the average of the distance of every training example to its corresponding cluster centroid.

The above cost function is often called the distortion of the training example.

In the clustering assignment step, our goal is to:

Minimize *J* with $c^1, \ldots, c^m$ (holding $\mu_1, \ldots, \mu_K$ fixed) with the equation(6.2), and find set *S*.

In the move centroids step, our goal is to:

Minimize *J* with $\mu_1, \ldots, \mu_K$. We move each centroid to the average of its group. More formally, the equation for this loop is as follows:

$$\mu_k = \frac{1}{p}\left(x^{k1} + x^{k2} + \cdots + x^{kp}\right) \tag{6.5}$$

Where *p* training examples are assigned to group S_k and x^{ki} are the training examples in S_k.

If you have a cluster centroid with 0 points assigned to it, you can randomly re-initialize that centroid to a new point. You can also simply eliminate that cluster group. (But you must be careful, or you maybe get the “wrong” clustering results.)

After a number of iterations the algorithm will converge, where new iterations do not affect the clusters (Maybe set a small tolerance $\varepsilon = 10^{-4}$ to terminate your iterations.)

Note on non-separated clusters: some datasets have no real inner separation or natural structure. K-Means can still evenly segment your data into *K* subsets, so you can still be useful in this case.

1.2. Random Initialization

Choosing *K* can be quite arbitrary and ambiguous. But *K* needs to be less than *m*.

The elbow method: plot the cost *J* and the number of clusters *K*. The cost function should reduce as we increase the number of clusters, and then flatten out. Choose *K* at the point where the cost function starts to flatten out. However, fairly often, the curve is very gradual, so there’s no clear elbow.

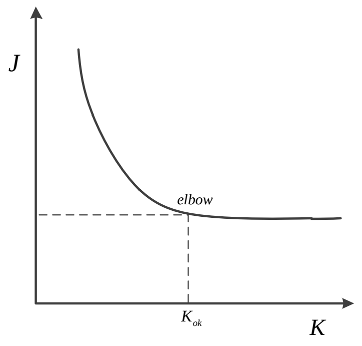


Figure 23 Choose *K* at the elbow point

Note that *J* will always decrease as *K* is increased. The one exception is if K-Means gets stuck at a bad local optimum.

Another way to choose *K* is to observe how well K-Means performs on a downstream purpose. In other words, you choose *K* that proves to be most useful for some goal you’re trying to achieve from using these clusters.

But if training examples are 1/2/3-dimensional vectors, they can be visualized to help you choose *K*.

Then initialize your *K* centroid points. There is one particular recommended method for randomly initializing your cluster centroids:

(1) Have $K < m$. That is, make sure the number of your clusters is less than the number of your training examples.

(2) Randomly pick *K* training examples. (Not mentioned in the lecture, but also be sure the selected examples are unique)

(3) Set $\mu_1, \ldots, \mu_K$ equal to these *K* examples.

K-Means can be get stuck in local optima. To decrease the chance of this happening, you can run the algorithm on many different random initializations. In cases where K<10 it is strongly recommended to run a loop of random initializations.

1.3. K-Means Algorithm

Above all, we conclude the K-Means Algorithm, and it may provide some help to your program.

```
(1) If necessary, perform feature scaling before optimization;
(2) Choose $K$ ($K<m$);
(3) Initialize $\mu_1,\ldots,\mu_K$ randomly;
(5) while ($\varepsilon$ > small value)
  {
      Assign clustering;
      Move centroids;
      Update $J$;
      Update $\varepsilon = J - J_{old}$;
  }
(6) Repeat (2)—(5) with different $K$, and determine the credible $K$;
(7) Determine the credible $\mu_1,\ldots,\mu_K$ ;
```

1.4. Example

We have an training data set X = [1, 3; 2, 2; 2, 3; 1.5, 4; 3, 2; 3, 3; 2.5, 2.5; 4, 5; 4, 6; 5, 4; 5, 5; 4.5, 6; 6, 4.5; 5.5, 5; 5.5, 6]. And we set the tolerance $\varepsilon = 10^{-4}$. We visualize these data in the figure and we find that K=2 is feasible. So we randomly choose two training examples(μ_1=[1, 3] and μ_2=[2, 2]). In common, we should perform feature scaling (it is really necessary), but in this case, I do not do it because I think the two features are the same order of magnitude (it is not a good idea). And we get the results as the following picture, and the centroid points are μ_1=[2.1429, 2.7857] and μ_2=[4.9375, 5.1875].

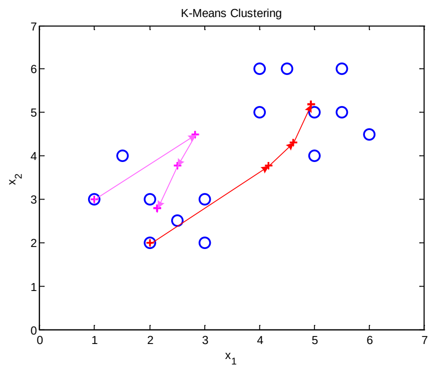


Figure 24 Moving Centroid Points Steps

Annotation

The drawbacks of K-Means:

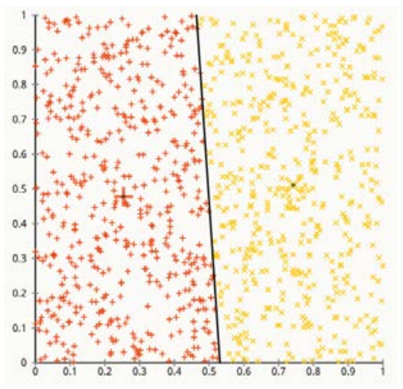
(1) K is uncertain without an assessment index. Maybe you have to try many different values of K to find a relative right one. But as we have known, the larger K is, the less the cost function is.

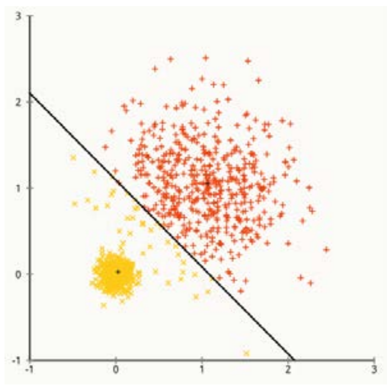
(2) K-Means can cluster non-clustered data because it just minimizes the SSE (sum of the square error) without considering the distribution states of data. (Left Upper Figure)

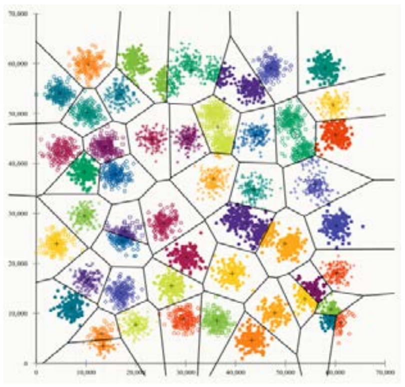
(3) Sensitive to scale, especially outlier points. Because the clusters have varying variances, and K-Means thus splits them incorrectly. (Right Upper Figure)

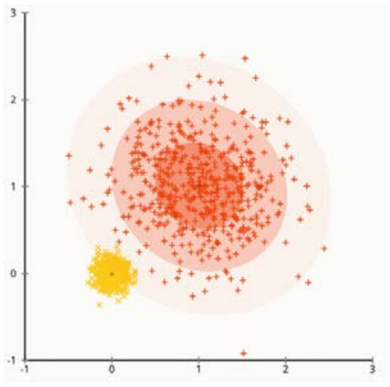
(4) K-Means can get stuck in a local Minimum. K-Means actually is a convex-optimization problem, and it cannot guarantee the global optima. (Left Lower Figure)

(5) In fact, before using K-Means to optimize problems, we have the assumptions: K-Means assumes the variance of the distribution of each cluster is spherical; All clusters have the same variance; The prior probability for all K clusters is the same, i.e. each cluster has roughly equal number of observations. (Right Lower Figure)









Advice for Applying Machine Learning

Suppose you have implemented regularized linear regression to predict housing prices with the cost function:

$$J(\theta)=\frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^i\right)-y^i\right)^2+\lambda\sum_{j=1}^{n}\theta_j^2\right] \tag{6.6}$$

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?

(1) Get more training examples;

(2) Try smaller sets of features;

(3) Try getting additional features;

(4) Try adding polynomial features

(5) Try decreasing λ;

(6) Try increasing λ.

Do not just pick one of these avenues at random. We will explore diagnostic techniques for choosing one of the above solutions in the following sections, that's "Diagnostic".

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

1. Evaluating a Hypothesis

A hypothesis may have low error for the training examples but still be inaccurate (because of overfitting). With a given dataset of training examples, we can split up the data into two sets: a training set and a test set.

The new procedure using these two sets is then:

(1) Learn θ and minimize $J_{train}(\theta)$ using the training set;

(2) Compute the test set error $J_{test}(\theta)$ (without regularization term).

$$J_{test}(\theta)=\frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}\left(h_\theta\left(x^i\right)-y^i\right)^2 \tag{6.7}$$

Table 4 Proportion of Data Sets

Data set	**Training data set**	**Test data set**
Proportion	70%	30%

2. Model Selection and Validation Test Set

Just because a learning algorithm fits a training set well, that does not mean it is a good hypothesis. The error of your hypothesis as measured on the data set with which you trained the parameters will be lower than any other data set.

In order to choose the model of your hypothesis, you can test each degree of polynomial and look at the error result.

2.1. Without the Validation Set

(1) Optimize the parameters in θ using the training set (e.g. for each polynomial degree);

(2) Find the credible model (e.g. polynomial degree) with the least error using the test set;

(3) Estimate the generalization error also using the test set with $J_{test}(\theta)$.

In this case, we have trained the model (e.g. the degree of the polynomial) using the test set. This will cause our error value to be greater for any other set of data.

2.2. Use the CV Set

To solve the problem, we introduce a third set, the Cross Validation Set, to serve as an intermediate set that we can train our model with. Then our test set will give us an accurate, non-optimistic error.

(1) Optimize the parameters in θ using the training set (e.g. for each polynomial degree);

(2) Find the credible model (e.g. polynomial degree) with the least error using the cross validation set with $J_{CV}(\theta)$

(3) Estimate the generalization error using the test set with $J_{test}(\theta)$.

$$J_{CV}(\theta) = \frac{1}{2m_{CV}} \sum_{i=1}^{m_{CV}} \left(h_\theta \left(x^i \right) - y^i \right)^2 \tag{6.8}$$

This way, the credible model has not been trained using the test set.

Note that using the CV set to select the credible model means that we cannot also use it for the validation curve process of setting the lambda value.

Table 5 Proportion of Data Sets

Data Set	**Training Data Set**	**Cross Validation Data Set**	**Test Data Set**
Proportion	60%	20%	20%

3. Diagnosing Bias vs. Variance

In this section we examine the relationship between the degree of the polynomial *d* and the under-fitting or overfitting of our hypothesis.

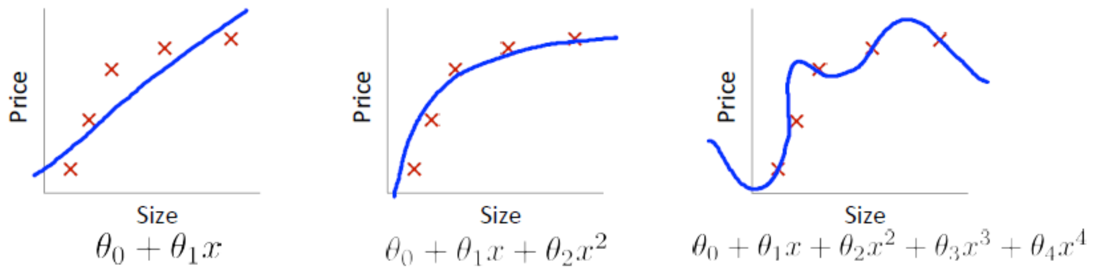


Figure 25 Relationship between *d* and Bias or Variance

(1) We need to distinguish whether bias or variance is the problem contributing to bad predictions.

(2) High bias is under-fitting and high variance is overfitting. We need to find a golden mean between these two.

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

High bias (under-fitting): both $J_{train}(\theta)$ and $J_{CV}(\theta)$ will be high. Also $J_{CV}(\theta) \approx J_{train}(\theta)$.

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

The figure below illustrates the relationship between *d* and the hypothesis:

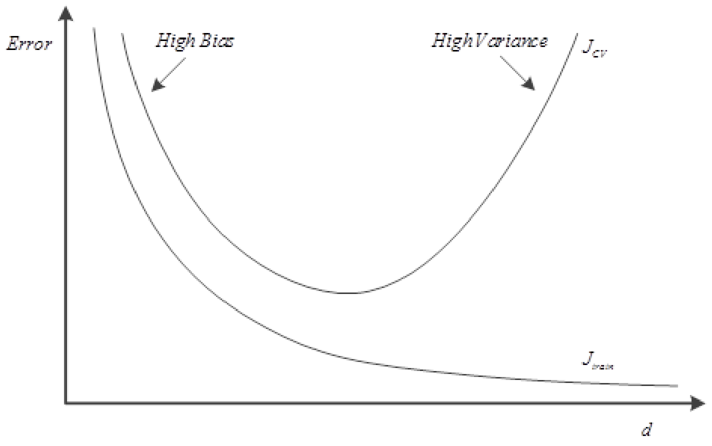


Figure 26 Relationship between *d* and the hypothesis

4. Regularization and Bias/Variance

Instead of looking at the degree *d* contributing to bias/variance, now we will look at the regularization parameter λ:

(1) Large λ: high bias (under-fitting)

(2) Intermediate λ: just right

(3) Small λ: high variance (overfitting)

A large λ heavily penalizes all the θ parameters, which greatly simplifies the line of resulting function, so causes under-fitting.

The relationship of λ to the training set and the variance set is as follows:

(1) Low λ: $J_{train}(\theta)$ is low and $J_{CV}(\theta)$ is high (high variance or overfitting)

(2) Intermediate λ: $J_{train}(\theta)$ and $J_{CV}(\theta)$ are somewhat low and $J_{train}(\theta) \approx J_{CV}(\theta)$

(3) Large λ: both $J_{train}(\theta)$ and $J_{CV}(\theta)$ will be high (high bias or under-fitting)

The figure below illustrates the relationship between λ and the hypothesis:

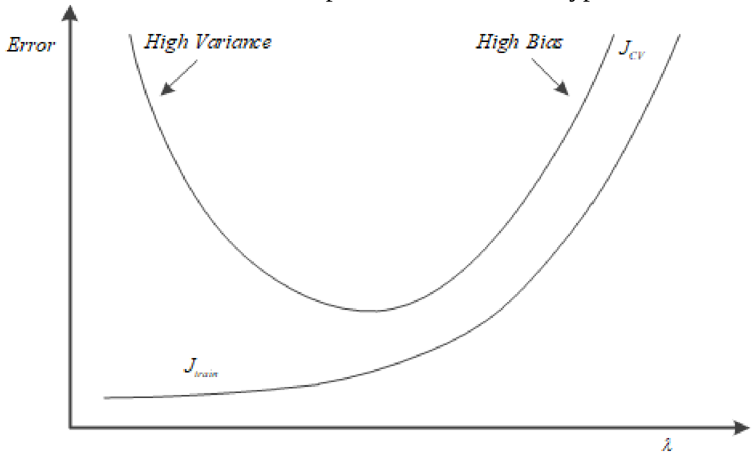


Figure 27 Relationship between λ and the hypothesis

In order to choose the model and regularization λ, we need:

(1) Create a list of λs (i.e. $\lambda \in \{0,0.01,0.02,0.04,0.08,0.16,0.32,0.64,1.28,2.56,5.12,10.24\}$);

(2) Choose a set of models with different degrees or any other variants;

(3) Iterate through the λs and for each λ go through all the models to learn some θ;

(4) Compute the cross validation error using the learned θ (computed with λ) on the $J_{CV}(\theta)$ without regularization or $\lambda = 0$;

(5) Select the best combo that produces the lowest error on the cross validation set;

(6) Using the best combo θ and λ, apply it on $J_{test}(\theta)$ to see if it has a good generalization of the problem.

5. Learning Curves

Training 3 examples will easily have 0 errors because we can always find a quadratic curve that exactly touches 3 points.

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.

For high bias, we have the following relationships in terms of training set size:

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

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

For high variance, the relationships in terms of the training set size are as:

(1) Low training set size: $J_{train}(\theta)$ will be low and $J_{CV}(\theta)$ will be high;

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

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

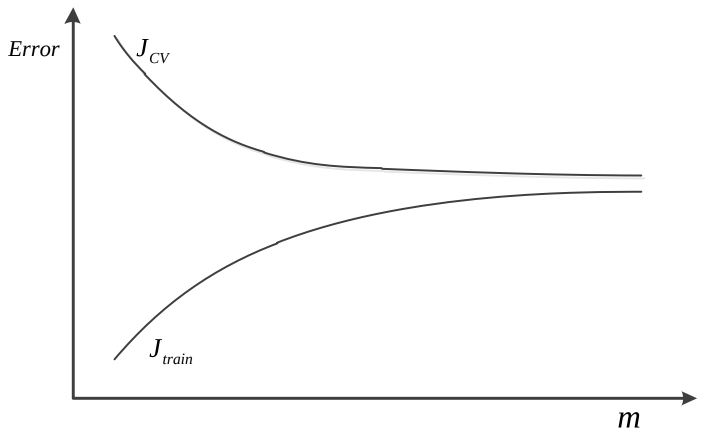


Figure 28 High bias

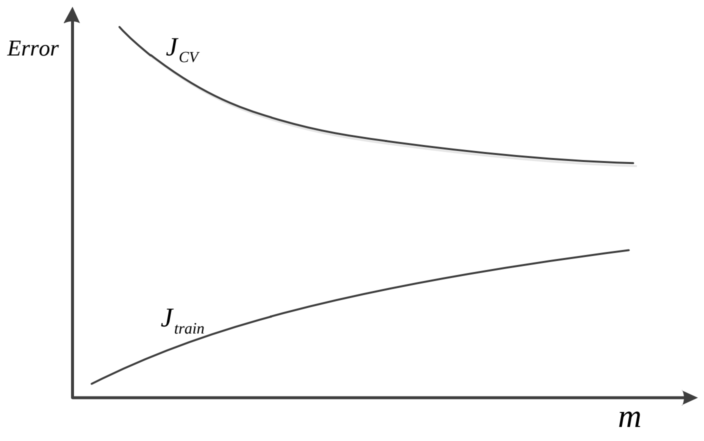


Figure 29 High variance

6. Deciding What to Do Next Revisited

Our decision process can be broken down as follows:

Table 6 Suggestions to fix high bias or variance

Getting more training examples:	Fix high variance
Trying smaller sets of features:	Fix high variance
Adding features:	Fix high bias
Adding polynomial features:	Fix high bias
Decreasing λ:	Fix high bias
Increasing λ:	Fix high variance

7. Diagnosing Neural Networks

A neural network with fewer parameters is prone to under-fitting. It is also computationally cheaper.

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

Using a single hidden layer is a good starting default. You can train your neural network on a number of hidden layers using your cross validation set.

8. Model Selection

How can we tell which parameters θ to leave in the model (known as “model selection”)?

There are several ways to solve this problem:

(1) Get more data (very difficult);

(2) Choose the model which best fits the data without overfitting (very difficult);

(3) Reduce the opportunity for overfitting through regularization.

Bias: approximation error (Difference between expected value and optimal value)

Variance: estimation error due to finite data

Intuition for the bias-variance trade-off:

(1) Complex → sensitive to data → much affected by changes in X → high variance, low bias;

(2) Simple model → more rigid → does not change as much with changes in X → low variance, high bias.

One of the most important goals in learning: finding a model that is just right in the bias-variance trade-off.

Regularization effects:

(1) Small values of λ allow model to become finely tuned to noise leading to large variance → overfitting;

(2) Large values of λ pull weight parameters to zero leading to large bias → under-fitting.

Model complexity effects:

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

(2) High model complexity (higher-order polynomials) fits the training data extremely well and the test data extremely poorly. These have low bias on the training data, but very high variance.

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

A typical rule of thumb when running diagnostics is:

(1) More training examples fix high variance but not high bias;

(2) Fewer features fix high variance but not high bias;

(3) Additional features fix high bias but not high variance;

(4) The addition of polynomial and interaction features fix high bias but not high variance;

(5) When using gradient descent, decreasing λ can fix high bias and increasing λ can fix high variance;

(6) When using neural networks, small neural networks are more prone to under-fitting and big neural networks are prone to overfitting. Cross-validation of network size is a way to choose alternatives.

Machine Learning System Design

If we design a spam classifier, how do you spend your time to make it have low error?

You maybe take actions as: (1) collect lots of data; (2) Develop sophisticated features based on email routing information (from email header); (3) 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? (4) Develop sophisticated algorithm to detect misspellings (e.g. m0rtgage, med1cine, w4tches.)

It is difficult to tell which of the options will be helpful. The recommended approach to solving machine learning problems is:

(1) Start with a simple algorithm, implement it quickly and test it early;

(2) Plot learning curves to decide if more data, more features, etc. will help;

(3) Error analysis: manually examine the errors on examples in the cross validation set and try to spot a trend.

It is important to get error results as a single, numerical value. Otherwise it is difficult to assess your algorithm's performance.

Before building a spam classifier, you may need to process your input, for example, if your input is a set of words, you may want to treat the same word with different forms (fail/failing/failed) as one word, so must use "stemming software" to recognize them all as one.

1. Error Metrics for Skewed Classes

It is sometimes difficult to tell whether a reduction in error is actually an improvement of the algorithm. For example, in predicting cancer diagnoses where 0.5% of the examples have cancer, we find our learning algorithm has a 1% error. However, if we were to simply classify every single example as a 0, then our error would reduce to 0.5% even though we did not improve the algorithm.

This is usually happens with skewed classes; that is, when our class is very rare in the entire data set. Or to say it another way, we have lot more examples from one class than from the other class. For this we can use Precision / Recall. We have a training set and its predicted results:

Table 7 Actual and predicted results

Data	**Actual 1**	**Actual 0**
Predicted 1	True positives	False positives
Predicted 0	False negatives	True negatives

Precision represents the proportion of true positives in total number of predicted positives:

$$precision = \frac{True\ Positives}{Total\ number\ of\ \ predicted\ \ positives} = \frac{True\ Positives}{True\ Positives + False\ Positives} \tag{6.9}$$

Recall represents the proportion of true positives in total number of actual positives:

$$precision = \frac{True\ Positives}{Total\ number\ of\ \ actual\ \ positives} = \frac{True\ Positives}{True\ Positives + False\ Negatives} \tag{6.10}$$

Accuracy represents the proportion of right predicted ones in total examples:

$$accuracy = \frac{True\ Positives + True\ Negatives}{Total\ examples} \tag{6.11}$$

These two metrics give us a better sense of how our classifier is doing. We want both precision and recall to be high. In the example at the beginning of the section, if we classify all patients as 0, then our recall will be 0, so despite having a lower error percentage, we can quickly see it has worse recall.

Note that if an algorithm predicts only negatives like it does in one of exercises, the precision is not defined, it is impossible to divide by 0.

2. Trading Off Precision and Recall

We might want a confident prediction of two classes using logistic regression. One way is to increase our threshold: Predict 1 if $h_\theta\left(x\right) \geq 0.7$ and predict 0 if $h_\theta\left(x\right) < 0.7$. This way, we only predict cancer if the patient has a 70% chance. Doing this, we will have higher precision but low recall.

In the opposite example, we can lower our threshold: Predict 1 if $h_\theta\left(x\right) \geq 0.3$ and predict 0 if $h_\theta\left(x\right) < 0.3$. This way, we get a very safe prediction. This will cause higher recall but lower precision.

The greater the threshold, the greater the precision and the lower the recall; The lower the threshold, the greater the recall and the lower the precision. In order to turn these two metrics into one single number, we can take the F value.

One way is to take the average:

$$F = \frac{P+R}{2} \tag{6.12}$$

But this does not work well. If we predict all $y = 0$ then we will bring the average up despite having 0 recall. If we predicted all examples as $y = 1$, then the very high recall will bring up the average despite having 0 precision.

A better way is to compute the F score (F1 score):

$$F_1 = 2\frac{PR}{P+R} \tag{6.13}$$

In order for the F score to be large, both precision and recall must be large.

We want to train precision and recall on the cross validation set so as not to bias our test set.

3. Data for Machine Learning

How much data should we train on?

In certain cases, an “inferior algorithm” if given enough data, can outperform a superior algorithm with less data. We must choose our features to have enough information. A useful test is: Given input *x*, would a human expert be able to confidently predict *y*?

Rationale for large data: if we have a low bias algorithm (many features or hidden units making a very complex function), then the larger the training set we use, the less we will have overfitting (and the more accurate the algorithm will be on the test set).

Dimensionality Reduction

There are some motivations to execute dimensionality reduction.

Motivation I: (1) We may want to reduce the dimension of our features if we have a lot of redundant data; (2) To do this, we find two highly correlated features, plot them, and make a new line that seems to describe both features accurately, and we place all the new features on this single line.

Doing dimensionality reduction will reduce the total data we have to store in computer memory and will speed up our learning algorithm.

Note that in dimensionality reduction, we are reducing our features rather than our number of examples. Our variable m will stay the same size; n, the number of features each example from x^1 to x^m carries, will be reduced.

Motivation II: Visualization.

It is not easy to visualize data that is more than three dimensions. We can reduce the dimensions of our data to 3 or less in order to plot it.

We need to find new features, z_1, z_2 and perhaps z_3 that can effectively summarize all the other features. For example, hundreds of features related to a county's economic system may all be combined into one feature that you call "Economic Activity".

1. Principal Component Analysis Problem Formulation

The most popular dimensionality reduction algorithm is Principal Component Analysis (PCA).

For example: given two features, x_1 and x_2, we want to find a single line that effectively describes both features at once. We then map our old features onto this new line to get a new single feature.

The same can be done with three features, where we map them to a plane.

The goal of PCA is to reduce the average of all the distances of every feature to the projection line. This is the projection error.

Reduce from 2d to 1d: find a direction (a vector $u^1 \in R^n$) onto which to project the data so as to minimize the projection error. If we are converting from 3d to 2d, we will project our data onto two directions (a plane), so k will be 2.

PCA is not linear regression. In linear regression, we are minimizing the squared error from every point to our predictor line. In PCA, we are minimizing the shortest distance, or shortest orthogonal distances, to our data points.

More generally, in linear regression we are taking all our examples in x and applying the parameters in θ to predict y.

In PCA, we are taking a number of features x_1, x_2, … , x_n, and finding a closet common dataset among them. We are not trying to predict any result and we are not applying any theta weights to the features.

2. Principal Component Analysis Algorithm

Before we can apply PCA, there is a data pre-processing step we must perform.

2.1. Data pre-processing

(1) Given training set: $(x_1, x_2, \dots, x_m)$;

(2) Pre-process (feature scaling or mean normalization):

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^i \tag{9.1}$$

(3) Replace each x_j^i with $x_j^i - \mu_j$. If different features on different scales, we need scale features to have comparable range of values. Above, we first subtract the mean of each feature from the original feature. Then we scale all the features:

$$x_j^i = \frac{x_j^i - \mu_j}{s_j} \tag{9.2}$$

We can define specifically what it means to reduce from 2d to 1d data as follows:

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}\left(x^i\right)\left(x^i\right)^T \tag{9.3}$$

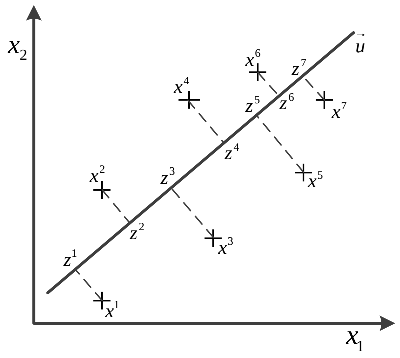


Figure 30 Projections from 2d to 1d

The z values are all real numbers and are the projections of our features onto u^1. So PCA has two tasks: figure out $u^1, \ldots, u^k$ and also find $z^1, \ldots, z^m$.

Now we computer "covariance matrix" as the formula(9.3), and the product of those will be an $n \times n$ matrix, which are the dimensions of Σ.

Then compute "eigenvectors" of covariance matrix Σ by using singular value decomposition[1], like svd function in Matlab or Octave: [U, S, V] = svd(Sigma), where Sigma is Σ matrix. U contains $u^1, \ldots, u^n$, which is exactly what we want.

After we get U, take the first k columns of the U matrix called "U_{reduce}" and compute z with:

$$z^i = \left(U_{reduce}\right)^T x^i \tag{9.4}$$

2.2. Reconstruction from Compressed Representation

If we use PCA to compress our data, how can we uncompress our data, or go back to our original number of features?

We can do this with the equation:

$$x_{approx}^i = U_{reduce} z^i \tag{9.5}$$

Note that we can only get approximations of our original data. It turns out that the U matrix has the special property that it is a Unitary Matrix[2]. One of the special properties of a Unitary Matrix is: $U^{-1} = U^*$ where U^* is the conjugate transpose matrix of U.

2.3. Choosing the Number of Principal Components

How do we choose k, also called the number of principal components? Recall that k is the dimension we are reducing to.

One way to choose k is by using the following formula:

(1) Given the average squared projection error: $E_{error} = \frac{1}{m}\sum_{i=1}^{m}\left\|x^i - x_{approx}^i\right\|^2$

(2) Also given the total variation in the data: $E = \frac{1}{m}\sum_{i=1}^{m}\left\|x^i\right\|^2$

(3) Choose k to be the smallest value such that: $E_{error} / E \le 0.01$

So you should try PCA with k = 1,2,… until it satisfies the unequal condition.

3. Advice for Applying PCA

The most common use of PCA is to speed up supervised learning.

Given a training set with a large number of features (e.g. $x^1, \ldots, x^m \in \mathrm{R}^{10000}$) we can use PCA to reduce the number of features in each example of the training set (e.g. $z^1, \ldots, z^m \in \mathrm{R}^{1000}$).

Note that we should define the PCA reduction from x^i to z^i only on the training set and not on the cross-validation or test sets. You can apply the mapping z^i to your cross-validation and test sets after it is defined on the training set.

Applications: (1) Compressions to reduce space of data and speed up algorithm; (2) Visualization of data if choose $k = 2$ or 3.

Bad use of PCA: Try to prevent overfitting although it can reduce the features of data. Note that you should try your full machine learning algorithm without PCA first.

Annotation

1. ***Sigular value decomposition (SVD)****: The singular value decomposition of an $m\times n$ real or complex matrix M is a factorization of the $U\Sigma V^*$, where U is an $m\times m$ real or complex unitary matrix, Σ is a $m\times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal, and V is an $n\times n$ real or complex unitary matrix.*

$$M = U\Sigma V^*$$

The diagonal entries σ_i of Σ are known as the singular values of M. The columns of U and the columns of V are called the left-singular vectors and right-singular vectors of M.

The singular value decomposition can be computed using the following observations:

(1) The left-singular vectors of M are a set of orthonormal eigenvectors of MM^*;

(2) The right-singular vectors of M are a set of orthonormal eigenvectors of M^*M;

(3) The non-zero singular values of M (found on the diagonal entries of Σ) are the square roots of the non-zero eigenvalues of both MM^* and M^*M.

*2.****Unitary Matrix****: In mathematics, a complex square matrix U is unitary if its conjugate transpose U^* is also its inverse — that is, if*

$$U^*U = UU^* = I$$

Where I is the identity matrix.

Properties: (1) Given two complex vectors x and y, multiplication by U preserves their inner product, that is, <Ux, Uy> = <x, y>; (2) U is normal ($U^*U=UU^*$); (3) U is diagonalizable, that is unitary similar to a diagonal matrix, as a consequence of the spectral theorem, and U has a decomposition of the form $U=VDV^*$ (V is unitary and D is diagonal and unitary); (4) $|det(U)| = 1$; (5) Its Eigen spaces are orthogonal; (6) The columns of U form an orthonormal basis of C^n with respect to the usual inner product; (7) The rows of U form an orthonormal basis of C^n with respect to the usual inner product.

Anomaly Detection

In data mining, anomaly detection (also outlier detection) is the identification of items, events or observations which do not conform to an expected pattern or other items in a dataset, such as bank fraud, a structural defect, medical problems or errors in a text.

Anomalies are also referred as outliers, novelties, noise, deviations and expectations. Three broad categories of anomaly detection techniques exist: unsupervised anomaly detection, supervised anomaly detection and semi-supervised anomaly detection. Unsupervised anomaly detection techniques detect anomalies in an unlabeled test data under the assumption that the majority of the instance in the data set are normal by looking for instances that seem to fit least to the remainder of the data set. Supervised anomaly detection techniques require a data set that has been labeled as “normal” and “abnormal” and involves training a classifier. Semi-supervised anomaly detection techniques construct a model representing normal behavior from a given normal training data set, and then testing the likelihood of a test instance to be generated by the learnt model.

Anomaly detection was proposed for intrusion detection systems (IDS) by Dorothy Denning in 1986.

1. Problem Motivation

Just like in other learning problems, we are given a dataset x^1, x^2, … , x^m. And we are then given a new example, x_{test}, and we want to know whether this new example is abnormal/anomalous.

We define a “model” $p(x)$ that tells us the probability the example is not anomalous. We also use a threshold ε as a dividing line so we can say which examples are anomalous and which are not.

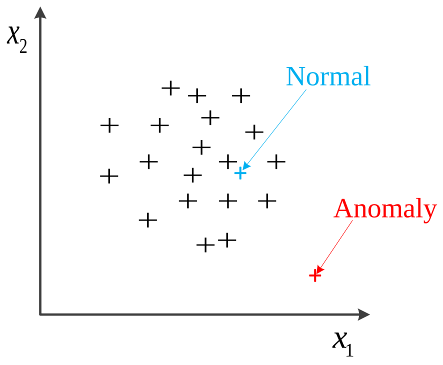


Figure 31 Anomaly Detection

2. Gaussian Distribution

The Gaussian distribution is a familiar bell-shaped curve that can be described by a function $\mathcal{N}(\mu, \sigma^2)$. If the probability distribution of x is Gaussian with mean μ, variance σ^2, then $x \sim \mathcal{N}(\mu, \sigma^2)$.

The Gaussian distribution is parameterized by a mean and a variance, where μ describes the center of the curve, called the mean, and σ called the standard deviation represents the width of the curve. The full function is as follows:

$$p\left(x;\mu,\sigma^2\right)=\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{10.1}$$

We can estimate the parameter μ from a given dataset by simply taking the average of all the examples:

$$\mu=\frac{1}{m}\sum_{i=1}^{m}x^i \tag{10.2}$$

And the σ^2 is estimated with the squared error formula:

$$\sigma^2=\frac{1}{m}\sum_{i=1}^{m}\left(x^i-\mu\right)^2 \tag{10.3}$$

3. Algorithm

Given a training set of examples, $\{x^1, x^2, \dots, x^m\}$ where each example is a vector, $x \in \mathbb{R}^n$, then:

$$p(x) = p(x_1; \mu_1, \sigma_1^2)\, p(x_2; \mu_2, \sigma_2^2) \cdots p(x_n; \mu_n, \sigma_n^2) \tag{10.4}$$

In statistics, this is called an "**independence assumption**" on the values of the features inside training example x. More compactly, the above expression can be written as follows:

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) \tag{10.5}$$

Its detailed algorithm is:

```
(1) Fit parameters $\mu_1, \mu_2, \dots, \mu_n$ and $\sigma_1^2, \sigma_2^2, \dots, \sigma_n^2$ using
        $\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^i, \ \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^i - \mu_j)^2$
(2) Calculate $p(x)$ of the new example $x$:
        $p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$
(3) Anomaly if $p(x) < \varepsilon$.
```

4. Supervised Anomaly Detection

The training examples are labeled and categorized into anomalous group (y = 1) and non-anomalous group (y = 0).

Among the data, take a large proportion of good, non-anomalous data for the training set on which to train $p(x)$. Then take a smaller proportion of mixed anomalous and non-anomalous examples for your cross validation and test sets.

For example, we may have a set where 0.2% of the data is anomalous. We take 60% of those examples, all of which are good (y = 0) for the training set. We then take 20% of the examples for the cross-validation set (with 0.1% of the anomalous examples) and another 20% from the test set (with another 0.1% of the anomalous). In other words, we split the data 60/20/20 training /CV/test and then split the anomalous examples 50/50 between the CV and test sets.

The algorithm is:

```
(1) Fit model $p(x)$ on training set $x^1, x^2, \dots, x^m$;
(2) On a cross validation/test example $x$, predict:
    If $p(x) < \varepsilon$ (anomaly), then $y = 1$;
    If $p(x) > \varepsilon$ (normal), then $y = 0$;
(3) Possible evaluation metrics: true positive, false positive,
    false negative, true negative; precision/recall; $F_1$ score
(4) repeat (2) and (3) steps to choose parameter $\varepsilon$ (using CV
    set)
```

The difference between anomaly detection and supervised classification learning:

Table 8 Difference between anomaly detection and supervised classification learning

Anomaly detection	**Supervised learning**
(1) Very small number of positive examples (y = 1), and large number of negative (y = 0) examples. (2) 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 before.	(1) A large number of positive and negative examples. (2) 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.

5. Feature Transformation

The feature will greatly affect how well your anomaly detection algorithm works.

We can check our features are Gaussian by plotting a histogram of our data and checking for the bell-shaped curve. Some transforms we can try on an example feature x that does not have the bell-shaped curve are: log(x + c), $\sqrt{x+c}$ and $(x + c)^{1/3}$ where c is a constant. We can play with each of these to try and achieve the Gaussian shape in our data. There is an error analysis procedure for anomaly detection that is very similar to the one in supervised learning. Our goal is for $p(x)$ to be large for normal examples and small for anomalous examples.

One common problem is when $p(x)$ is similar for both types of examples. In this case, you need to examine the anomalous examples that are giving high probability in detail and try to figure out new features that will better distinguish the data. In general, choose features that might take on unusually large or small values in the event of an anomaly.

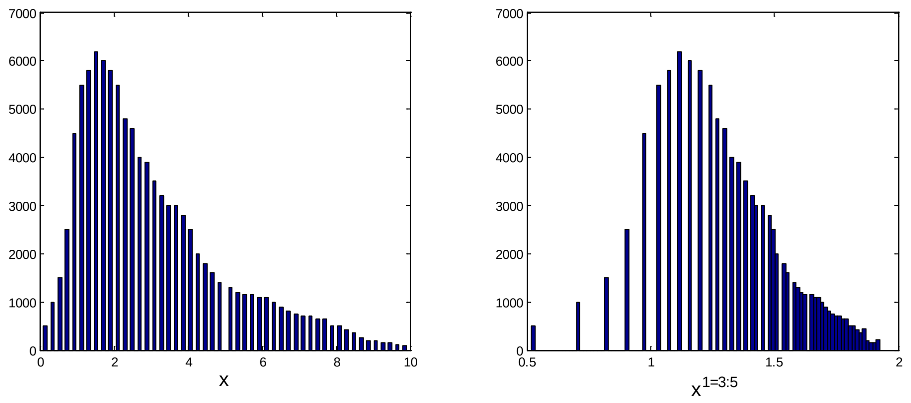


Figure 32 Transformation of x to be Gaussian distribution

6. Multivariate Gaussian Distribution

The multivariate Gaussian distribution is an extension of anomaly detection and may (or may not) catch more anomalies. But it does not depend on “independence assumption”, because of the usage of covariance matrix Σ. In fact, there may be some relations between features. So instead of modeling $p(x_i)$ separately, we will model all $p(x_i)$ in one go. Our parameters will be: $\mu\in\mathbb{R}^n$, and $\Sigma\in\mathbb{R}^{n\times n}$, and we have:

$$p(x;\mu,\Sigma)=\frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}}\exp\left[-1/2(x-\mu)^T\Sigma^{-1}(x-\mu)\right] \tag{10.6}$$

Where $|\Sigma|$ is the determinant of Σ. It is an anomaly if $p(x)<\varepsilon$.

The important effect is that we can model oblong Gaussian contours, allowing us to better fit data that might not fit into the normal circular contours.

Varying Σ changes the shape, and orientation of the contours. Changing μ will move the center of the distribution.

Given a training set of examples, $\{x^1, x^2, \dots, x^m\}$ where each example is a vector, $x\in\mathbb{R}^n$, then:

$$\mu=\frac{1}{m}\sum_{i=1}^{m}x^i,\Sigma=\frac{1}{m}\sum_{i=1}^{m}(x^i-\mu)(x^i-\mu)^T \tag{10.7}$$

The multivariate Gaussian model can automatically capture correlations between different features of x.

We know that Σ is a diagonal matrix where every diagonal element is σ_i^2 when all features are mutual independent. In other words, the original model $p(x)$ corresponds to a multivariate Gaussian where the contours of $p(x; \mu, \Sigma)$ are axis-aligned. However, the original model maintains some advantages: it is computationally cheaper (no matrix to invert, which is costly for large number of features) and it performs well even with small training set size (in multivariate Gaussian model, it should be greater than the number of features for Σ to be invertible).

Recommender System

A recommender system or a recommendation system is a subclass of information filtering system that seeks to predict the "rating" or "preference" that a user would give to an item. Recommender systems are very popular application of machine learning, and are utilized in a variety of areas including movies, music, news and so on.

Recommender systems typically produce a list of recommendations in one of two ways — "collaborative filtering" and "content-based filtering", sometimes through hybrid recommender systems. Collaborative filtering approaches build a model from a user's past behavior as well as similar decisions made by other users. This model is then used to predict items (or ratings for items) that the user may have interest in. Content-based filtering approaches utilize a series of discrete characteristic of an item in order to recommend additional items with similar properties. A hybrid recommender system comes from their combined approaches.

Assume that we are trying to recommend movies to customers. We can use the following definitions:

n_u —	number of users
n_m —	number of movies
$r(i,j)$ —	equal to 1 if user *j* has rated movie *i*
$y(i,j)$ —	rating given by user *j* to movie *i* (defined only if $r(i,j)=1$)
θ^j —	parameter vector for user *j*
θ_k^j —	the *k*th parameter of θ^j
x^i —	feature vector for movie *i*
m^j —	number of movies rated by user *j*

1. Content-Based Filtering

We can give a table to help us to understand this problem.

Table 9 A movie recommender system example

Movie/Users	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		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

In this table, we introduce two features of movies — x_1 and x_2 which represents how much romance and how much action a movie may have (on a scale of 0-1).

One approach is that we could do linear regression for every single user. For each user *j*, learn a parameter θ^j. Predict user *j* as rating movie *i* with x^i. To learn θ^j, we do the following:

$$\min_{\theta^j} \frac{1}{2} \sum_{i:r(i,j)=1} \left[\left(\theta^j\right)^T x^i - y^{(i,j)} \right]^2 + \frac{\lambda}{2} \sum_{k=1}^{n} \left(\theta_k^j\right)^2 \tag{11.1}$$

Where the base of the first summation is choosing all *i* such that $r(i,j)=1$. To get the parameters for all our users, we modify the formula as follows:

$$\min_{\theta^j (j=1,\dots,n_u)} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left[\left(\theta^j\right)^T x^i - y^{(i,j)} \right]^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(\theta_k^j\right)^2 \tag{11.2}$$

We can apply our linear regression gradient descent update using the above cost function:

$$\theta_k^j := \theta_k^j - \alpha \sum_{i:r(i,j)=1} \left[\left(\theta^j\right)^T x^i - y^{(i,j)} \right] x_k^i \quad (k=0)$$
$$\theta_k^j := \theta_k^j - \alpha \left[\sum_{i:r(i,j)=1} \left[\left(\theta^j\right)^T x^i - y^{(i,j)} \right] x_k^i + \lambda \theta_k^j \right] \quad (k \neq 0) \tag{11.3}$$

2. Collaborative Filtering

It can be very difficult to find features such as "amount of romance" or "amount of action" in a movie. To figure this out, we can use feature finders.

We can let the users tell us how much they like the different genres, providing their parameter vector immediately for us.

Given $\theta^1, \ldots, \theta^{n_u}$, we can predict x^i :

$$\min_{x^1,\ldots,x^{n_m}} \frac{1}{2} \sum_{i:r(i,j)=1} \left[\left(\theta^j\right)^T x^i - y^{(i,j)} \right]^2 + \frac{\lambda}{2} \sum_{k=1}^{n} \left(x_k^i\right)^2 \tag{11.4}$$

To infer the features from given parameters, we use the squared error function with regularization over all the users:

$$\min_{x^1,\ldots,x^{n_m}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{i:r(i,j)=1} \left[\left(\theta^j\right)^T x^i - y^{(i,j)} \right]^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} \left(x_k^i\right)^2 \tag{11.5}$$

From above content, we can randomly guess the values for θ to guess the features repeatedly to converge to a good set of features if we do not know the parameters. But to speed things up, we can simultaneously minimize our features and our parameters:

$$J(x,\theta) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} \left[\left(\theta^j\right)^T x^i - y^{(i,j)} \right]^2 + \frac{\lambda_x}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} \left(x_k^i\right)^2 + \frac{\lambda_\theta}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(\theta_k^j\right)^2 \tag{11.6}$$

Because the algorithm can learn them itself, the bias units where $x_0 = 1$ have been removed, therefore $x \in \mathbb{R}^n, \theta \in \mathbb{R}^n$.

We can also get the gradient descent formulas as:

$$x_k^i := x_k^i - \alpha_x \left[\sum_{j:r(i,j)=1} \left[\left(\theta^j\right)^T x^i - y^{(i,j)} \right] \theta_k^j + \lambda_x x_k^i \right]$$
$$\theta_k^j := \theta_k^j - \alpha_\theta \left[\sum_{i:r(i,j)=1} \left[\left(\theta^j\right)^T x^i - y^{(i,j)} \right] x_k^i + \lambda_\theta \theta_k^j \right] \tag{11.7}$$

There are the steps in the algorithm:

(1) Initialize $x^1, \ldots, x^{n_m}$ and $\theta^1, \ldots, \theta^{n_u}$ to small random values; This serves to break symmetry and ensures that the algorithm learns features $x^1, \ldots, x^{n_m}$ that are different from each other.
(2) Minimizing $J(x,\theta)$ using gradient descent (or an advanced optimization algorithm);
(3) With Parameters θ and features x, predict result by using $\theta^T x$.

Given matrices X (each row containing features of a particular movie) and Θ (each row containing the weights for those features for a given user), then the full matrix Y of all predicted rating of all movies by all users is given simply by: $Y = X\Theta^T$.

If you want to predict how similar two movies i and j are can be done using the distance between their respective feature vectors x which is $\left\| x^i - x^j \right\|$.

3. Mean Normalization

If the ranking system for movies is used from the previous lectures, then new users will be assigned new movies incorrectly. Specifically, they will be assigned θ with all components equal to zero due to the minimization of the regularization term. That is, we assume that the new user will rank all movies 0, which does not seem intuitively correct.

We rectify this problem by normalizing the data relative to the mean. First, we use a matrix *Y* to store the data from previous ratings, where the *i*th row of *Y* is the ratings for the *i*th movie and the *j*th column corresponds to the ratings for the *j*th user. We define a new vector: $\mu = \left[\mu_1, \ldots, \mu_{n_m}\right]$ such that

$$\mu_i = \frac{\sum_{j:r(i,j)=1} Y_{i,j}}{\sum_{j:r(i,j)=1} r(i,j)} \tag{11.8}$$

Which is effectively the mean of the previous ratings for the *i*th movie (where only movies that have been watched by users are counted). We now can normalize the data by subtracting μ, the mean rating, from the actual ratings for each user (column in matrix *Y*) to get Y':

As an example, consider the following matrix *Y* and mean ratings μ:

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 4 & ? & ? & 0 \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 5 & 0 \end{bmatrix}, \mu = \begin{bmatrix} 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix} \Rightarrow Y' = \begin{bmatrix} 2.5 & 2.5 & -2.5 & -2.5 \\ 2 & ? & ? & -2 \\ -2.25 & -2.25 & 2.75 & 1.75 \\ -1.25 & -1.25 & 3.75 & -1.25 \end{bmatrix}$$

Now we slightly modify the linear regression prediction to include the mean normalization term:

$$\left(\theta^j\right)^T x^i + \mu_i$$

For a new user, the initial predicted values will be equal to the μ term instead of simply being initialized to zero, which is more accurate.

The last Chapter

This chapter will introduce some technologies that are useful in your machine learning programs.

1. Online Learning

With a continuous stream of users to a website, we can run an endless loop that gets (x, y), where we collect some user actions for the features in x to predict some behavior y.

You can update θ for each individual (x, y) pair as you collect them. This way, you can adapt to new pools of users, since you are continuously updating θ.

2. Map-Reduce and Data Parallelism

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. It is a software architecture created by Google to handle big data problems. But in 2004, Google developed Apache Mahout replacing MapReduce as their primary big data processing model.

It works when you have many computers or a computer with multiple cores:

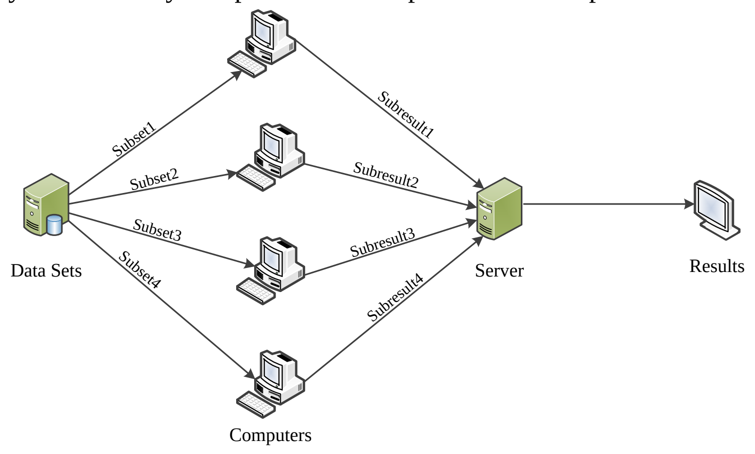


Figure 33 MapReduce with multiple computers

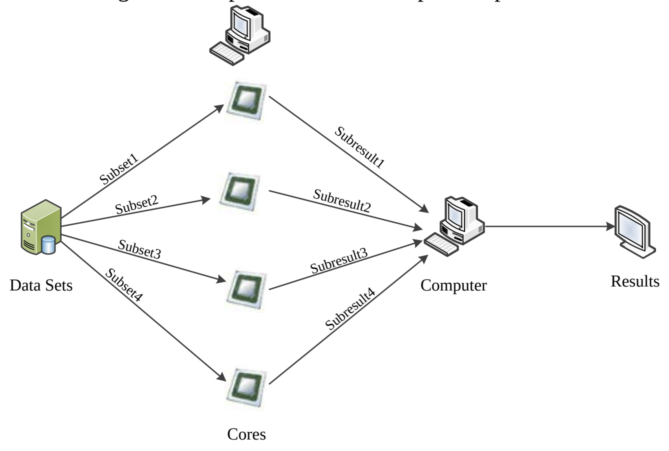


Figure 34 MapReduce with multiple cores

MapReduce generally performs a 5-step parallel and distribution computation:

(1) Prepare the Map() input: the “MapReduce system” designates Map processors, assigns the input key value *K1* that each processor would work on, and provide that processor with all the Map-generated data associated;

(2) Run the user-provided Map() code: Map() is run exactly once for each *K1* key value, generating output organized by key values *K2*;

(3) “Shuffle” the Map output to the Reduce processors: The MapReduce system designates

reduce processors, assigns the *K2* key value each processor should work on, and provide that processor with all the Map-generated data associated with that key value;

(4) Run the user-provided Reduce() code: Reduce() is run exactly once for each *K2* key value produced by the Map step;

(5) Produce the final output: the MapReduce system collects all the reduce output, and sorts it by *K2* to produce the final outcome.

You can split your training set into z subsets corresponding to the number of machines you have.

Your learning algorithm is MapReduceable if it can be expressed as computing sums of functions over the training set. Linear regression and logistic regression are easily parallelizable. For neural networks, you can compute forward propagation and back propagation on subsets of your data on many machines. Those machines can report their derivatives back to a “master” server that will combine them.