This is my summary of course Machine Learning by Andrew Ng on Coursera. You can have a reference here after finishing the course. I'm glad to communicate with you and learn from each other. If you find any mistakes in the article, I would appreciate it if you pointed them out.(If there is anything wrong with the appearance of the formulas please refer to the pdf edition.)

[a pdf edition](http://www.luckycallor.com/wp-content/uploads/2016/06/Summary-of-course-Machine-Learning-by-Andrew-Ng-on-Coursera.pdf)

**1. Linear Regression**

For m examples with n features, we can use a matrix X (with m rows and n columns) to describe the data, where row vector {x^i}(with n+1 dimension including{x_0}) represents an example, while column vector {x_j}(with m dimension) represents a feature; or in matrix X, every element x_j^i(row i, column j) represent the jth feature of ith example.

For parameters, we use vector {\rm{\theta }} with n+1 elements to describe, where {\theta _j} is correspond to {x_j}.

For labels, we use a vector y (with m elements) to represent, where element {y_i} represent the label of ith example.

And for every 1 \le i \le m,x_0^i = 0.

**Hypothesis:**

    Vector version:

{h_\theta } = X*\theta 

    Element version:

{h_\theta }\left( {{x^i}} \right) = \mathop \sum \limits_{j = 0}^n x_j^i{\theta _j}

**Cost function:**

**Gradient descent:**

    Vector version:

{\rm{\theta }} = {\rm{\theta }} - {\rm{\alpha }}\frac{1}{m}{X^T}\left( {{h_\theta } - {\rm{y}}} \right)

    Element version:

**Computing parameters analytically:**

To minimize the cost function J, we can also compute the parameter vector {\rm{\theta }} by mathematical analysis. Consequently there is a normal equation to compute the optimal {\rm{\theta }}:

{\rm{\theta }} = {\left( {{X^T}X} \right)^{ - 1}}{X^T}{\rm{y}}

The following table compares Gradient descent and Normal equation:

|  |  |
| --- | --- |
| Gradient descent | Normal equation |
| Need to choose {\rm{\alpha }} | No need to choose {\rm{\alpha }} |
| Need many iterations | No need to iterate |
| Work well even when n is large | Need to compute {\left( {{X^T}X} \right)^{ - 1}}(time complexity: {\rm{O}}\left( {{n^3}} \right)),slow when n is large, such as 10000 |

**Polynomial regression:**

For every example, by multiplying its features with each other to make new features, such as {x_1}{x_2},{x_1}{x_1} and so on. Then append these new features to the original ones and we get polynomial regression.

**Note:**

    (1) Make sure features are on a similar scale. You can make  - 1\le x \le 1 by feature scaling:

{x_i} = \frac{{{x_i} - mean}}{{max - min}}

    (2) Learning rate {\rm{\alpha }}: too small slow convergence, too large J may not decrease on every iteration; may not converge.

**2. Logistic Regression**

**Classification problem:**

Every example {x^i} has a label {y_i} whose value is either 0 or 1. The goal of classification is to label examples without labels.

**Hypothesis:**

Sigmoid function:

g\left( z \right) = \frac{1}{{1 + {e^{ - z}}}}

    Hypothesis:

{h_\theta }\left( {{x^i}} \right) = g\left( {{x^i}\theta } \right) = \frac{1}{{1 + {e^{{x^i}\theta }}}}

    Notice that value of {h_\theta }\left( {{x^i}} \right) represents the probability of ith example being labeled as 1.

**Cost function:**

**Gradient descent:**

    Vector version:

{\rm{\theta }} = {\rm{\theta }} - {\rm{\alpha }}{X^T}\left( {{h_\theta } - {\rm{y}}} \right)

    Element version:

{\theta _j} = {\theta _j} - \alpha \mathop \sum \limits_{i = 1}^m \left( {{h_\theta }\left( {{x^i}} \right) - {y_i}} \right)x_j^i

**Multiclass classification:**

One vs all: pick each class as the positive class and others the negative one, compute {h_\theta }, then select the highest-{h_\theta }-class as the label.

**3. Neural Network**

Neural network is a structure with some layers, which includes at least an input and output layer, and besides there may be some mid-layers.

Every layer is a vector. Supposing an example with n features, the input layer is a vector with n+1 elements (including {x_0} = 1), and if there are totally k classes, then the output layer is a vector with k elements each of whom represent the probability for one class. The mid-layers can have as many elements as you want, which depends on the structure of your neural networks.

Between every two neighboring layers, there is a set of parameters which determines how the next layer comes from the last layer, and this process is called forward propagation. To minimize the cost function, we use back propagation to update those parameters.

**Forward propagation:**

    Element version:

a_i^c = g\left( {\mathop \sum \limits_{j = 0}^n \theta _j^ia_j^l} \right), *i=1~m*

    Vector version:

{a^c} = g\left( {\theta {a^l}} \right)

    Note:

(1) You should add an element a_0^c = 1.

(2) Letter 'a' is the layer vector and the superscript 'c' refers to 'current', and 'l' refers to 'last'. N is the number of elements in the last layer, while m is that in the current layer both excluding {a_0} which always equals to 1. Therefore {\rm{\theta }} is a matrix with m rows and n+1 columns. Notice that there is an independent matrix {\rm{\theta }} between every two neighboring layer vector. The function g() is the sigmoid function.

**Cost function:**

L: number of layers

K: number of classes

{s_l}: number of units in layer l

**Back propagation:**

    Def: \delta _j^l: "error" of node j in layer l.

    For the output layer:

\delta _j^L = a_j^L - {y_j}

    For layer l = 2 ~ L-1 :

{\delta ^l} = {\left( {{\theta ^l}} \right)^T}{\delta ^{l + 1}}.*g'\left( {{z^l}} \right)

g'\left( {{z^l}} \right) = {a^l}.*\left( {1 - {a^l}} \right)

    Gradient descent:

\frac{{\partial J\left( \theta \right)}}{{\partial \theta _{ij}^l}} = a_j^l\delta _i^{l + 1}\frac{{}}{{}}

**Note:**

    (1) {\rm{\theta }} should be initialized randomly

    (2) Before applying back propagation, you should check whether it is right

**4. Support Vector Machine**

**Support vector machine:**

    Cost function:

*Function*cos{t_1}\left( z \right)*has the property that when z is large (larger than 1)*cos{t_1}\left( z \right)*is 0, while the function*cos{t_0}\left( z \right)*has the property that when z is small (smaller than -1)*cos{t_0}\left( z \right)*is 0.*

    Goal: minimize J\left( \theta \right)

    Property: SVM can find a boundary with a large margin.

**Kernel:**

    Intro: Given x, compute new feature depending on proximity to landmarks.

    Gaussian kernel:

Supposing we have k landmarks: {l^i}*(i=1~k)* , then for each example x, we can compute a new example f with k features:

{f^i} = \exp \left( { - \frac{{x - {l^i}^2}}{{2{\sigma ^2}}}} \right) ,  
*for each i in 1~k*

    Other kernels: polynomial kernel, string kernel, chi-square kernel, histogram intersection kernel, …

**Note:**

1. The parameter C:

Large C: lower bias, high variance

Small C: higher bias, low variance

2. Logistic regression vs. SVMs vs. Neural network

*n=number of features, m=number of training examples*

(1) If n (relative to m) is large: use logistic regression of SVM without a kernel

(2) If n is small, m is intermediate: use SVM with Gaussian kernel

(3) If n is small, m is large: create/add more features, then use logistic regression or SVM without a kernel

(4) Neural network likely to work well for most of these settings, but may be slower to train.

**5. K-Means**

(1) Choose K points as the centroid: {\mu _1},{\mu _2}, \ldots .,{\mu _K}

(2) For each example {x^i}, *i = 1~m*

Update {c^i}: index(from 1 to K) of cluster centroid closest to {x^i}

(3) For k = 1 to K

{\mu _k}: update to average of (mean) of points assigned to cluster k.

Do until {\mu _k} remains fixed, or come to the maximum steps.

**6. Principal components analysis**

**Compress m dimensions to k dimensions:**

Sigma = \mathop \sum \limits_{i = 1}^m \left( {{x^i}} \right){\left( {{x^i}} \right)^T}

*[U,S,V]=SVD(Sigma)*

{U_{reduce}} = U\left[ {:,1:K} \right]

z = {U'_{reduce}}*x

*Where U is an n by n matrix, and*{U_{reduce}}*is an n by k matrix, and z is the reduced representation.*

**Reconstruction from compressed representation:**

{x_{approx}} = {U_{reduce}}*z

**Choosing K:**

Compute the rate of variance retained:

\frac{{\mathop \sum \nolimits_{i = 1}^k {S_{ii}}}}{{\mathop \sum \nolimits_{i = 1}^n {S_{ii}}}}

Choose K according to the rate above.

**7. Anomaly Detection**

**Anomaly detection:**

    Gaussian distribution:

    Steps:

        (1) Choose features {x_i} that you think might be indicative of anomalous examples

        (2) Fit parameters {\mu _1}, \ldots ,{\mu _n},\sigma _1^2, \ldots ,\sigma _n^2

{\mu _j} = \frac{1}{m}\mathop \sum \limits_{i = 1}^m x_j^i

\sigma _j^2 = \frac{1}{m}\mathop \sum \limits_{i = 1}^m {\left( {x_j^i - {\mu _j}} \right)^2}

        (3) Given new example x, compute p(x):

{\rm{p}}\left( {\rm{x}} \right) = \mathop \prod \limits_{j = 1}^n p\left( {{x_j};{\mu _j},\sigma _j^2} \right)

        Anomaly if p(x) < ϵ.

Anomaly detection vs. supervised learning:

|  |  |
| --- | --- |
| Anomaly detection | Supervised learning |
| Very small number of positive examples, large number of negative examples. | Large number of positive and negative examples. |
| Many different "types" of anomalies. Hard to learn the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far. | 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. |

**Multivariate gaussian distribution**

    Parameter fitting:

{\rm{\mu }} = \frac{1}{m}\mathop \sum \limits_{i = 1}^m {x^i},

**8. Recommender System**

**Supposing:**

{r^{\left( {i,j} \right)}} = 1*, if user j has rated movie i (0 otherwise)*

{y^{\left( {i,j} \right)}} = *rating by user j on movie i (if defined)*

{\theta ^j} = *parameter vector for user j*

{x^i} = *feature vector for movie i*

{\left( {{\theta ^j}} \right)^T}\left( {{x^i}} \right)*:the predicted rating for user j on movie i*

{m^j} = *no.of movies rated by user j*

{n_u}*:no.of user*

{n_m}*:no.of movies*

n*:no.of movie features*

**Optimization algorithm:**

**Gradient descent update:**

,for k = 0

,for k ≠0

**Collaborative filtering:**

    Given {x^i}, \ldots ,{x^{{n_m}}}, estimate {\theta ^1}, \ldots ,{\theta ^{{n_u}}}:

    Given {\theta ^1}, \ldots ,{\theta ^{{n_u}}}, estimate {x^i}, \ldots ,{x^{{n_m}}}:

**Mean normalization:**

In case that some users rated none of movies, for every rating value, we can minus the mean rating of that movie before training. And when predicting ratings, add the mean rating of that movie.

**9. Evaluation of algorithm**

1. Split the dataset into three portions: train set, validate set and test set, in a proportion 3:1:1.

2. When the number of examples m increase, the cost {J_{test}} increases, while {J_{val}} decrease. When m is very large, if {J_{test}} is about equal to {J_{val}} the algorithm may suffer from large bias, while if there is a gap between {J_{test}} and {J_{val}} the algorithm may suffer from large variance.

3. To solve the problem of large bias, you may decrease {\rm{\lambda }} in regularization, while increase it for the problem of large variance.

4. To evaluate the performance of a classification algorithm, we can use the value: precision, recall and F1.

    Precision:

\frac{{TruePositive}}{{TruePositive + FakePositive}}

    Recall:

\frac{{TruePositive}}{{TruePositive + FakeNegtive}}

    F1:

\frac{{2*Recall*Precision}}{{Recall + Precision}}

**10. Overfitting and Regularization**

When the hypothesis function is too complex or there are too many features while the number of training examples is not large enough, you may get an overfitting problem. In that case, J\left( \theta \right) of the training set may be very low, while that of the validate set and test set can be high. A good method to solve the problem is regularization which adds the squared term of parameters to the cost function.

**Regularized linear regression:**

    Cost function:

    Gradient descent:

 ,j ≠0

**Regularized logistic regression:**

    Cost function:

    Gradient descent:

 ,j ≠0

**11. Advanced Topics**

**Stochastic gradient descent:**

1. Randomly shuffle (reorder) training examples

2. Repeat {

    For i:=1,…,m{

{\theta _j} = {\theta _j} - \alpha \left( {{h_\theta }\left( {{x^i}} \right) - {y^i}} \right)x_j^i

            (for every j=0,…,n)

    }

}

To test whether the algorithm is converging, you can plot the mean cost of one example in a set of iterators, and if it's decreasing, then the algorithm is converging, otherwise, {\rm{\alpha }} may be too large.

**Online learning:**

Online learning examples always have a dynamic data, which means that the data is like a flood and there are always new data coming. In that case, we can update the parameters as new data coming:

Repeat forever {

    Get new (x, y)

Update {\rm{\theta }} using (x, y): {\theta _j} = {\theta _j} - \alpha \left( {{h_\theta }\left( x \right) - {y^i}} \right){x_j}

}

**Artificial data synthesis:**

Create data from scratch: this means that you create data using your creation and innovation, such as using different front to create examples in character recognition.

Synthesizing data by introducing distortions: Notice that distortion introduced should be representation of the type of noise/distortions in the test set, while adding purely random/meaningless noise does not help.

**Some advanced algorithm to minimize**\left( \theta \right)**:**

Conjugate descent, BFGS, L-BFGS. When using these algorithms you need to have the function to compute J\left( \theta \right) and \frac{{\partial {J_{\left( {{\theta _ \ldots }} \right)}}}}{{\partial {\theta _j}}}