Summary of Machine Learning on Coursera

Yan Gobeil

February 15, 2019

This is a summary of the important things that I learned while doing Andrew Ng's course on Machine Learning on Coursera. I focus on the theoretical concepts and on the small tricks instead on the definitions and intuitions. The goal is to have a reference for each algorithm with useful tricks. Pictures would be useful but can be found in the course.

Contents

1 Notation and general idea

Let's first setup the notation that is used for most of the course. The number of training examples is m and they are represented by the variable x. The number of features is n so each training example lives in $\mathbb{R}^n$. The target for each training example is called y and a set $(x^{(i)}, y^{(i)})$ represents the full training example i with its target. It is sometimes useful to use a vectorized notation where y is a column vector that includes all the training examples and X is a matrix of size (m, n) where each row is a training example and each column is a feature. Most formulas in the course can be written using this notation but I will only write the explicit ones for simplicity.

The goal of machine learning is to use the data to train a model that can describe the situation. The model is called a hypothesis and it is a function $h(x)$ of the features that is supposed to predict accurately the targets and generalize nicely to new examples. The hypothesis contains parameters called θ that must be determined in order to get the best function possible in order to make predictions. The parameters are found by asking that a certain cost function $J(\theta)$, that determines how close the hypothesis is to the data, is at its smallest. There are many ways to do this and they will be discussed soon.

Regularization

One important problem that can occur in machine learning is overfitting. This happens when the model predicts very nicely the data that was used to train it but is unable to generalize to new data. This will be discussed in more details later but one way to manage this is to use a regularization technique to make sure that the parameters don't get so big that some of them dominate the others and pick up bad trends in the data. This is done by adding a term of the form $\lambda \sum_{j=1}^{n} (\theta_j)^2$ to the cost function.

2 Gradient descent

The usual way of minimizing the cost function is to use gradient descent. The idea is to start with some random set of parameters that doesn't matter very much and then follow the gradient of the cost function since it is supposed to point in the direction of highest variation. Algorithmically it means that you start with some set of θ_j's and update them simultaneously according to the following rule

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

The parameter α is the learning rate and it determines the size of the steps used when following the gradient. If α is too small it may take a while for gradient descent to converge but if it is too large it is possible to not even get convergence. If the algorithm works correctly the cost should decrease at each step so a way of making sure that everything works is to plot the cost as a function of the number of iterations.

Gradient checking

There will be explicit formulas for the gradient but it is possible to check if gradient descent is doing its job by comparing to the approximate derivatives obtained directly from the cost function. The idea is that we can update each θ_i by the following quantity at each step

$$\frac{\partial J(\theta)}{\partial \theta_i} = \frac{J(\theta_1, ..., \theta_i + \epsilon, ..., \theta_n) - J(\theta_1, ..., \theta_i - \epsilon, ..., \theta_n)}{2\epsilon}$$

Here ϵ is just some random small number.

Stochastic gradient descent

As the cost function is normally a sum over all the training examples it becomes very computationally expensive to do the sum at each step of the algorithm when the data set is very large. This is why people use in this case stochastic gradient descent. The idea is to randomly mix the training example and then update all the parameters by adding the contribution from the cost of only one training example at each step and repeating this until all the examples have been used. This can then be repeated a few times until convergence is obtained. This way of doing things is faster because we don't go over all the examples too many times, but the cost function will not necessarily decrease at each step. In order to evaluate if the algorithm works we wait for some time (something like 1000 steps) and plot the average of the cost for all the steps since the previous plot. This is supposed to make some wavy line that decreases with the steps.

There is a middle ground between the usual method, called batch gradient descent, and the stochastic version. We can sum over a small fraction of the training examples at each step to do mini-batch gradient descent. It is also possible to parallelize the calculations for batch gradient descent by asking different computers to manage the sum for a part of the data and gather everything together once the sums are done to update the parameters.

Feature scaling

To help gradient descent converge it may be useful to make sure that all the features are on the same scale. The usual rule is that we would like $-1 \lessapprox x_i \lessapprox 1$. This can be achieved by normalizing the data to have zero mean and scaling it

$$x_i \to \frac{x_i - \mu_i}{s_i}$$

Here μ is the average and s can be either the range of the feature (max-min) or the standard deviation.

3 Linear regression

The first machine learning algorithm is linear regression, which uses a linear hypothesis to predict the value of some variable. The hypothesis for this one is

$$h(x) = \theta_0 x_0 + \theta_1 x_1 + ... + \theta_n x_n = \theta^T x$$

Here we introduce $x_0 = 1$ for a cleaner notation. Each feature appears linearly so it is a very simple hypothesis. The cost function that is used is

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})^2$$

This is a convex function so gradient descent will always converge to the global minimum. Including the regularization term, the correct updating rule for gradient descent is

$$\theta_0 = \theta_0 - \frac{\alpha}{m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})$$

$$\theta_j = \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \frac{\alpha}{m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)}) x_j^{(i)} \qquad (j \geq 1)$$

Note that θ_0 is not regularized.

For this simple potential it turns out that the minimum can be found directly by asking where the gradient vanishes. The result in matrix form is called the normal equation

$$\theta = (X^T X + \lambda I)^{-1} X^T y$$

This gives a direct answer but is slower than gradient descent when the dataset is large (around 10^4) since we have to invert very big matrices. This can also fail if the inverse doesn't exist because some features are linearly dependent. Without regularization this also doesn't work if $m < n$.

It is actually possible to make more complicated hypothesis using polynomial regression simply by using new features built out of the old ones. An example would be to consider a case where there is only one feature but we build two new ones out of it and use

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

4 Logistic regression

The next machine learning algorithm is used to classify data points into categories. The examples that are positive are labeled by 1 and the negative ones are 0. The hypothesis is now

$$h(x) = g(\theta^T x)$$

with the sigmoid function

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

The hypothesis is interpreted as the probability that the data point is positive and the classification is achieved by using a threshold $h(x) \geq p$, which is usually 0.5. The equation $h(x) = p$ gives a formula for the boundary that separates the positive and negative predictions. Depending if polynomial features are used this boundary can take many shapes.

The cost function that is used for logistic regression is

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

The expression for gradient descent is actually the same as for linear regression but simply with a different hypothesis function.

Classification into multiple classes can be achieved by using the one vs all method. This means that we run the algorithm multiple times and each time the positive examples are the ones from a different class and all the other classes are negative. This will give a probability for each class and the highest one is the prediction.

5 Neural networks

For systems that are very complex with a lot of data it may be hard for simple algorithms to work correctly. Neural networks have been developed to manage big data and to model problems with non linearities, such as image and language processing. The class discusses how to use neural networks for classification but they are useful for regression as well. A neural network is made out of layers of neurons that are connected to each other. The first layer is composed of the input features x_i and it simply contains the data into n neurons. There is also an output layer that gives us the expected result. For usual classification there is one neuron with the probability that the example is positive but for multi class classification there are the same number of neurons as the number of classes K. Each neuron gives the probability that an example is each class. The rest of the network is composed of some number of hidden layers (total of L including input and output) that each contain some number s_j of neurons. The values of the neurons in layer j are $a_0^{(j)}, ..., a_{s_j}^{(j)}$ and we always add a bias neuron $a_0^{(j)} = 1$ for hidden layers and $x_0 = 1$ for the input layer. The most important part of the network is the parameters that relate the different layers. This is achieved by matrices $\Theta^{(j)}$ that

relate layer j to layer $j+1$. The elements of these matrices are the parameters that the algorithm tunes in order to make accurate predictions.

There are two parts in the training of a neural network. The first one is forward propagation, which is the way the information is propagated through the network from the input layer to the output layer. Even if the notation is a little heavy this step is pretty simple. The main calculation that is done is

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

$$a^{(j+1)} = g(z^{(j+1)}) \,.$$

Since this is discussed for classification the function $g(z)$ is the usual sigmoid function.This procedure starts from the input layer $a^{(1)} = x$ and continues layer by layer until the output layer.

After the information has been propagated to the output the model must know if it has made a good prediction. This is achieved by using the cost function

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\Big(a_k^{(L)}\Big) + (1-y_k^{(i)}) \log\Big(1-a_k^{(L)}\Big)\right] + \frac{\lambda}{2m}\sum_{\ell=1}^{L-1}\sum_{i=1}^{s_\ell}\sum_{j=1}^{s_{\ell+1}} (\Theta_{ij}^{(\ell)})^2 \,.$$

This is basically the same as for regularized logistic regression but with a more complicated hypothesis and many more parameters. The weights are then updated according to the usual procedure of gradient descent in order to get a better answer. However since the network is so complicated there is a special way of doing gradient descent that is called back propagation. The idea is to compute, starting from the output, the error on each neuron. The output error is simply

$$\delta^{(L)} = a^{(L)} - y$$

and the others are calculated with

$$\delta^{(\ell)} = (\Theta^{(\ell)})^T \delta^{(\ell+1)} .* g'(z^{(\ell)}) \,.$$

The symbol $.*$ means multiply component by component. This is mainly the chain rule for derivatives of the potential so that

$$\frac{\partial}{\partial \Theta_{ij}^{(\ell)}} J(\Theta) = a_j^{(\ell)} \delta_i^{(\ell+1)} \,.$$

The full algorithm to train a neural network is then

1. Initialize $\Delta_{ij}^{(\ell)} = 0$ for all i, j, ℓ
2. For $i = 1, ..., m$
   (a) Initialize the input layer $a^{(1)} = x^{(i)}$
   (b) Do forward propagation to get all the $a^{(\ell)}$'s
   (c) Compute all the $\delta^{(\ell)}$'s
   (d) Do back propagation with the update $\Delta_{ij}^{(\ell)} += a_j^{(\ell)} \delta_i^{(\ell+1)}$
   (e) Compute the gradient (including regularization for the terms that don't involve the bias units $j = 0$) with $D_{ij}^{(\ell)} += \frac{1}{m}\Delta_{ij}^{(\ell)} + \lambda \Theta_{ij}^{(\ell)}$
3. Repeat until convergence

The last few tips that are important when dealing with neural nets are that the matrices for the parameters and the gradient must be unrolled into vectors to make the calculations simpler. The weights must also be randomly initialized at the beginning in some range $(-\epsilon, \epsilon)$ to avoid being stuck in a trivial minimum. Finally since the gradient is computed in a very complicated way it is useful to use gradient checking at the beginning to confirm that the algorithm works correctly.

6 Train/validation/test

When training a machine learning algorithm on some data it is expected that the error of the algorithm on the same data will be small. Nothing however guarantees that the model will do well on new data so we need to check that. The most common way is to split the available data into two or three sets before even starting the fitting. Most of the data should go into the training set that will be used to train the model. The rest goes either on a test set that is used as a check since the model has never seen the data or it is split again into test and validation sets. The validation data is important when there are parameters to change in the model. For example we would like to be able to fix the regularization constant λ or the degree of a polynomial in polynomial regression. The model is trained on the training data for each value of the parameter, the error on the validation set allows us to decide the best value and the test set serves as a check for the accuracy of the final model. Usually the error on the training data will decrease as the complexity of the hypothesis grows. The test error will however have the shape of a parabola since a simple model can't describe data that is too complex and models that are too complex are prone to overfitting, as explained later. This leaves a sweet spot for the parameters.

One of the most important concept in machine learning is the bias-variance tradeoff. A model has high bias when it is underfitting. This can be seen from the fact that the training set error and the test set error are both high. A way of avoiding this is to decrease the regularization parameter or add more features to the model to make it more flexible. Having more data is not helpful when a model is underfitting. A model has high variance when it has a low error on the training set but high error on the test set. The most important way of avoiding this is regularization but it is also possible to get rid of some features or use more data.

A nice way to diagnose over or under fitting is the learning curve, which plots the training and test errors as a function of the size of the training set. The training error will always increase as we add data because it is harder to fit exactly with all the points. The test error will always decrease as we add data because the model can pick up more patterns and generalize better. The key difference between high bias and high variance is that getting more data doesn't help when we underfit so the test error curve will become flat very fast. The overfitting case will however show that adding more data is useful and that the test error can be improved by a lot.

7 Error metrics

It is important to know how to evaluate if a model does well. For regression the cost function is actually a simple way of judging but for classification it is more subtle. Accuracy (number of correct predictions over total examples) is a good one but it can be misleading in the case where the data is skewed, meaning that there are only a few positive examples. In this case we need more advanced metrics. A few terms are needed to understand the metrics. True positives (TP) are positive examples that were classified as positive. False positives (FP) are negative examples that were classified as positive. False negatives (FN) are positive examples that were classified as negative. True negative (TN) are negative examples that were classified as negative. The first useful metric is

$$precision(P) = \frac{TP}{TP + FP}$$

This counts the proportion of all the examples that were classified as positives that indeed were positive. The next metric is

$$recall(R) = \frac{TP}{TP + FN}$$

This counts the proportion of all the actual positive examples that were classified as positive.

If we want to insist on one of the metrics, changing the threshold that determines which examples are positive can be useful. Increasing the threshold leads to less positive classification, so precision normally increases and recall decreases. Lowering the threshold makes the classification of an example as positive more frequent, so it normally decreases the precision but lowers the recall.

A nice metric that combines both precision and recall in a clear and significant way if the F score, defined by

$$F = 2\frac{P\,R}{P+R}$$

8 Anomaly detection

The goal of this algorithm is to predict if some points in the data are anomalously far from the rest. The main idea is to consider all relevant features to be independent and model each of them with a Gaussian probability distribution

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

The total probability is then

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

and if $p(x) < \epsilon$ for some threshold we assume that the event is anomalous. The training that is done is simply computing the mean and the variance of each feature

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$$

The parameter ϵ can be decided in advance or be deduced from the cross-validation set.

If some actual anomalies are known it is best to keep them for the cross-validation and the test sets to make the most efficient use of them. In the training just assume that nothing is anomalous. The trick here is not the algorithm but more the features that are used. It is important to plot the distribution of each feature to check that they look somehow Gaussian. If it is not the case it can be possible to apply some transformation to the feature to make it look more Gaussian. If some known anomalies turn out to not look anomalous with respect to some features it is necessary to create new features from these to make sure that anomalies are visible. Finally if there are a lot of known anomalous examples it may be more efficient to simply use supervised learning algorithms for classification.

If there seems to be some correlation between the features it may be possible to add it by hand to the hypothesis by combining some of the Gaussians together. There is however a more systematic way for taking care of correlations. Consider the vector of means and the covariance matrix

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

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

The probability is then given by a multivariate Gaussian

$$p(x) = \frac{1}{(2\pi)^{\frac{n}{2}}\sqrt{|\Sigma|}}\exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

This is definitely more precise than the assumption of independence but it is more expensive to implement and it doesn't even work when $m < n$ because the matrix is not invertible.

9 K-means clustering

An example of how unsupervised learning is used is to find clusters of similar data points in a set of data. The algorithm starts by randomly initializing the centroids $\mu_k \in \mathbb{R}^n$ for the K clusters. The next step is to go over all the points and give them the label $c^{(i)}$ that corresponds to the centroid that is the closest to it. The last step is to move the clusters to a new place determined by the average of all the points that were assigned to it. This simple algorithm is repeated until nothing changes.

Since this algorithm can give different results for different random initializations we need a good way to know which result is the best. For that we use the cost function that depends on the position of the centroids and the points attached to them

$$J(c^{(i)}, \mu_i) = \frac{1}{m} \sum_{i=1}^{m} |x^{(i)} - \mu_{c^{(i)}}|^2$$

We then run the algorithm for different initializations and pick the result that minimizes the cost.

If one wants to choose the number of clusters that is the most significant there exists the elbow method. If we run the algorithm for different values of K and plot the total final cost there is sometimes an elbow that appears on the plot and this is the values of K that should be used. Unfortunately this is not perfect and most of the time there will not be a clear elbow.

10 Principal component analysis

Another example of unsupervised learning is PCA, which is used to reduce the dimensionality of the data. This projects the training examples $x \in \mathbb{R}^n$ to some lower dimensional space $z \in \mathbb{R}^k$. It is useful mostly to compress data and to help with visualization. It is essential to do feature scaling before using this algorithm. The main ingredient to calculate is the covariance matrix

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

We then find the eigenvectors of this matrix and the corresponding eigenvalues. We pick the eigenvectors that correspond to the k largest eigenvalues and make the reduction matrix out of them

$$U = (u^{(1)} ... u^{(k)})$$

These are the directions along which the data shows the most variation. The projection of the data points is achieved by $z = U^T x$. We can project back to the higher dimensional space and obtain an approximation to the original data with $x_{approx} = Uz$. It is possible to know approximately how much information is lost in this compression by computing the error

$$1 - \frac{\sum_{i=1}^{k} eigenvalue_i}{\sum_{i=1}^{n} eigenvalue_i}$$

The eigenvalues need to be organized in descending order to use this formula. We can directly compute this quantity for different values of k and decide how many directions we want to keep.

PCA can be useful to speed up the convergence of machine learning algorithms since it contains pretty much the same information in a smaller number of features. It is however important to try the algorithms without PCA before to possibly avoid all the trouble of doing it. If PCA is used it is important to use it separately on the training and the test data. It is also not a good idea to reduce the number of features with PCA to avoid overfitting. Always use regularization instead.

PCA can be useful to visualize clusters or correlations in data but the new features don't have a clear meaning so it is hard to interpret anything from it.

11 Other algorithms

The class covers three other algorithms that we don't discuss because they are too hard to summarize without going into too many details. The first one is recommender systems. The class discusses how to implement content based and collaborative filtering algorithms. The second one is support vector machines. These are classification algorithms that perform very well but are more mathematical and need some visualization to understand. There is also a discussion of photo OCR where they explain how to use the sliding window method for text detection. They also discuss how to create new data from old one in the context of text recognition and how to implement machine learning pipelines and do ceiling analysis for errors in pipelines.