Practical advices for applying machine Learning

From Coursera machine learning course: Andrew NG

General Steps to debug a machine learning algorithm

- Get more examples --> helps to fix high variance. Not good if you have high bias.
- Use smaller set of features --> fixes high variance (overfitting). Not good if you have high bias
- Try adding additional features --> fixes high bias (because hypothesis is too simple, make hypothesis more specific)
- Add polynomial terms --> fixes high bias problem.
- Decreasing λ --> fixes high bias.
- Increases λ --> fixes high variance.

Evaluating a single model with training-test sets

How do you tell if a hypothesis is overfitting, or we cannot generalize? For a very low dimensional hypothesis function we can plot to see whether it is overfitting. **Don't forget this for this simple and much better way for visualized hypothesis function, and always resort to the train-test split.**

For non-visualized hypothesis function, then we can use the normal train-test-split way. However, if the data set is ordered, then sample randomly for the training set.

To increase the accuracy, we may split the training-test set in many ways and then take the average, called cross validation. However, it is OK not to do cross validation here as we have only one hypothesis function, and we have trained only ONCE. If we want to further choose different models, then we should do cross-validation as we trained more than once there. **The thumb of rule should be: for each training process we should have a data split(see next section).**

Evaluating multiple models with training-validation-test sets

Here the model section refers to the choice of regularization parameter, or the degrees of polynomial function. In other words, the different models are obtained by variating a controlling parameter. If we choose very different models, it should following the same rule described below.
Whenever we choose an optimal target from multiple candidates, we then have a training process. Whenever we have a training process, we need a train-test splitting to check the generalization error. For example, when evaluating the single hypothesis function case (previous

section), we choose a set of parameters for the target hypothesis function to minimize the cost function. Thus we split the data into train and test sets to calculate the generalized error. Now we will do a model selection we need two training processes. Thus we need split the data twice, or we need have three different parts: train, cross-validate, tests.

Step 1:
In data set 1 (training set) we train a model (e.g. a specific hypothesis function with a polynomial order d = 1) and obtain a set of parameters for this model in order to achieve an optimal target (e.g. a minimized cost or maximized likelihood). We then train a similar model with d = 2 on the same training data and obtain another set of parameters for this model. We repeat the similar training process for, e.g., d = 1,2,..10 and thus we have 10 sets of parameters, or 10 different models.

Step 2:
In data set 2 (called cross validation set) we train the 10 different models obtained before. We choose the set of parameter (e.g. d = 5) that can achieve the minimum cost function with data set 2. Note we cannot use the chosen parameter set to calculate the generalized error. This is because we are still do the training on this data set.

Step 3:
In data set 3 (called test set), we use the chose parameters to calculate the finally generalized test error.

The test data should never be used for training purpose. In other words, it cannot be used to determine parameters of any models. It can only be used for testing.

In practice, many people will select the model using the test set and then check the model is OK for generalization using the test error. However, a much better practice is to have separate training and validation sets. Only with a MASSIVE test set, the former practice maybe OK.

Bias-variance in model selection: choosing polynomial degree

Plot $J_{cv}(J_{test})$ and J_{train} as a function of d (the degree of parameter), we find $J_{cv}(J_{test})$ is a bow-shaped curve (not always). This shows either very low d or very high d corresponds to high error, the left side is mainly from bias and right side is from variance.

The $J_{train} \sim d$ curve is monotonically going up, indicating higher order polynomial can simulate the training set with very small error.

Bias-variance in model selection: choosing regularization parameter

Plot $J_{cv}(J_{test})$ and J_{train} as a function of regularization parameter λ, we find $J_{cv}(J_{test})$ is a bow-shaped curve (not always). This shows either very low λ or very high λ corresponds to high error, the left side is mainly from variance and right side is from bias (**Different from that of previous section**).

The $J_{train} \sim d$ curve is monotonically going up (**different from that of previous section**) indicating higher λ will give very big bias.

Learning curves

The previous two sections are about the J_{cv}/J_{train} as function of d and λ respectively. Here the Learning curves are about the functions of J_{cv}/J_{train} with respect to the number of samples m. However, the main purpose of studying learning curve is still bias and variance.

If we have high bias, then the two curves will approach the same horizontal line (with big value) for a moderate number of m. Adding more samples will not help.

If we have high variance, then for moderate m, the two curves will still have big gap. If we add more samples, then it helps.

The schematic curves shown in other notes should be far dirtier in reality.

Machine Learning System Design

Machine learning systems design

You can understand all the algorithms, but if you don't understand how to make them work in a complete system that's no good!

Prioritizing what to work on

The idea of prioritizing what to work on is perhaps the most important skill programmers typically need to develop. It's so easy to have many ideas you want to work on, and as a result do none of them well, because doing one well is harder than doing six superficially.

- So you need to make sure you complete projects.
- Get something "shipped" - even if it doesn't have all the bells and whistles, that final 20% getting it ready is often the toughest.
- If you only release when you're totally happy you rarely get practice doing that final 20%.

In summary, to prioritize is just first to focus on something, complete it, and then improve. But not start a lot of things without finishing a single one.

Designing the feature

Designing a machine learning system is very different from project to project. Here we use an example of designing spam classification system. See other write-up for feature selection and extraction.

In practice it is more common to pick the most frequently n words, where n is 10 000 to 50 000 from a training set to form a vector. Then map each email in the training set into this vector. If a word is in it, the vector element takes a value of 1. See details of cs229 notes 2.

What's the best use of your time to improve system accuracy?

- Natural inclination is to collect lots of data. Honey pot anti-spam projects try and get fake email addresses into spammers' hands, collect loads of spam. This doesn't always help though.
- Develop sophisticated features based on email routing information (contained in email header) Spammers often try and obscure origins of email. Send through unusual routes
- Develop sophisticated features for message body analysis. Discount == discounts? DEAL == deal?
- Develop sophisticated algorithm to detect misspelling. Spammers use misspelled word to get around detection systems.
- Often a research group randomly focus on one option. May not be the most fruitful way to spend your time. If you brainstorm a set of options this is really good. Very tempting to just try something

Error analysis

- If you're building a machine learning system often good to start by building a simple algorithm which you can implement quickly. Spend at most 24 hours developing an initially bootstrapped algorithm. Implement and test on **cross validation data**.
- Plot learning curves to decide if more data, features etc. will help algorithmic optimization.
  (1) Hard to tell in advance what is important.
  (2) Learning curves really help with this.
  (3) Way of avoiding premature optimization. We should let evidence guide decision making regarding development trajectory. Optimization or performance tuning should often be performed **at the end of the development stage**. Otherwise premature optimization or performance considerations may affect the design of a general project. In other words, the early stage of optimization on branches may affect the general optimization of the whole project.
- Error analysis examples on spamming classification
  (1) Built a spam classifier with 500 examples **in cross validation set**. Usually in the early stage, error rate is high. Manually examine the samples that your algorithm made errors on. See if you can work out why. For systematic patterns - help design new features to avoid these shortcomings.
  (2) Manually look at 100 and categorize them depending on features. e.g. type of email.
  (3) Looking at those email (a) May find most common type of spam emails are pharmacy emails, phishing emails. See which type is most common - focus your work on those ones. (b) What features would have helped classify them correctly. e.g. deliberate misspelling. Unusual email routing, Unusual punctuation. May find some "spammer technique" is causing a lot of your misses. Guide a way around it.
  (4) If you have implemented new features on, for example, spelling correction, sender host feature, email header feature, email text parser features, javascript parser, feartures from embedded images.., then you can analyze your accuracy by removing these features one by one. This is sometimes also called ablative analysis.

- Error analysis examples on image recognition
  For a pipeline of image -> pre-process (e.g. background removal) -> Face detection (eye segmentation, Nose segmentation, mouse segmentation) -> logistic regression -> output.
  (1) First, we may individually study the performance of each algorithm. For example, we can do a perfect background removal manually, and then compare with our background removal algorithm. If the perfect background removal does not have a big boost in accuracy, then we may need first focus on other issues.
  (2) Second we can analyze whether each step is useful in accounting for the accuracy. If it is really not that important, then we may focus on other issues. This step is also called ablative analysis.
- Importance of numerical evaluation (1) It's really good to have some performance calculation which gives a single real number to tell you how well its doing.

(2) Example:
(a) Say we are deciding if we should treat a set of similar words as the same word.
(b) This is done by stemming in NLP (e.g. "Porter stemmer" looks at the etymological stem of a word).
(c) This may make your algorithm better or worse. Also worth consider weighting error (false positive vs. false negative) e.g. is a false positive really bad, or is it worth have a few of one to improve performance a lot.
(d)Can use numerical evaluation to compare the changes.

(3) A single real number may be hard/complicated to compute (**see F1 score later**). But makes it much easier to evaluate how changes impact your algorithm.

- Error analysis should always be done on cross validation set

Data for machine learning

Turns out **under certain conditions** getting more data is a very effective way to improve performance. When is this true and when is it not?

- If we can correctly assume that features x have enough information to predict y accurately, then more data will probably help. A useful test to determine if this is true can be, **"given x, can a human expert predict y?"**
- So lets say we use a learning algorithm with many parameters such as logistic regression or linear regression with many features, or neural networks with many hidden features.
  (1) These are powerful learning algorithms with many parameters which can fit complex functions Such algorithms are low bias algorithms
  Little systemic bias in their description - flexible
  (2) Use a small training set Training error should be small (3) Use a very large training set If the training set error is close to the test

set error
Unlikely to over fit with our complex algorithms
So the test set error should also be small

- Another way to think about this is we want our algorithm to have low bias and low variance Low bias --> use complex algorithm
  Low variance --> use large training set