Machine learning strategies

Kristian Wichmann

September 14, 2017

1 Introduction

This note describes a number of strategies, tips and tricks for successfully navigating actual machine learning projects.

1.1 The project development cycle

Basically, the evolution of a ML project over time comes in cycles of three phases:

$$\text{Idea} \rightarrow \text{Code} \rightarrow \text{Experiment} \tag{1.1}$$

In short: Start with an idea, implement it in code. Then get your hands dirty and experiment with running the code in different ways. Eventually, this will hopefully lead to getting a new and hopefully better idea, which will start the cycle over again, until a satisfactory product has been build.

1.2 Orthogonalization

Orthogonalization is essentially knowing what to tune to achieve what effect.

Ideally, when turning one "knob" on you ML problem, it should only affect one aspect of the outcome - each does only one, easily interpretable, thing. That way, you may tune one "knob" at a time to get the desired behaviour from the algorithm.

1.2.1 Chain of assumptions

When building an ML product, there's a chain of assumptions being made:

- We can fit the parameters well to the training set, so that the chosen cost function is low. In other words, we assume that we can get an acceptable performance on the training set. Sometimes this means getting human level performance or better.

- We then hope this model also does well on the dev set. Failure to do so us typically a sign of overfitting.
- And on the test set as well.
- This should lead to good performance out in the real world.

For each of these problems, we have different "knobs" to turn in order to get the desired result. Here's some examples that may help achieve succes for each:

- Performance on the training set:
  - Training a bigger network, or more generally a model with greater capacity.
  - Choosing a better strategy for gradient descent. ADAM for instance.
  - Train your model for more epochs.
- Performance on the dev set:
  - Regularization, like ridge regression or dropout.
  - Getting a bigger training set.
- Performance on the test set:
  - Get a bigger dev set.
- Performance in the real world:
  - Change the dev set to better correspond to real world data.
  - Change the cost function to better fit the problem.

We'll dive into a lot of these in more detail below.

1.2.2 Non-orthogonal example: Early stopping

Some strategies are non-orthogonal. I.e. they affect several of the points in the chain above at the same time. One such example is early stopping, where performance is monitored on the training and dev set simultaneously. Therefore, in this case we're "tuning two knobs" at once.

	Precision	Recall
Classifier A	95%	90%
Classifier B	98%	85%

Table 1: Properties of two classifiers

2 Setting your goal

This is the crucial question for training the model: What are our optimization criteria? Which quantities do we want to optimize, and under which contraints (if any)? This section outlines a number of strategies, and thoughts on when/if it's a good idea to change these goals along the way.

2.1 Single evaluation metric

In this case, there's simply one quantity which gauges the performance of the model. We can then train the model to optimize the quantity. The cost function is an example.

If we have a number of discrete options to choose from, we can simply pick the one with the highest/lowest performance.

Often, there will be several relevant metrics related to a problem. Sometimes we can combine these into one, as the following example shows.

2.1.1 Example: F_1 score

For classification, both precision and recall are relevant metrics to consider. If we have trained two classifiers A and B with the properties when evaluated on the dev set shown in table 2.1.1, it is not immediately clear which one we should pick; there's often a trade-off between the two.

One way to combine the two, is the F_1 score, which is the harmonic mean of the precision (P) and recall (R):

$$F_1 = \frac{2}{\frac{1}{P} + \frac{1}{R}} = \frac{2}{\frac{P+R}{PR}} = \frac{2PR}{P+R} \tag{2.1}$$

We may now calculate the F_1 score of the two classifiers:

$$A : F_1 = \frac{2 \cdot 0.96 \cdot 0.90}{0.96 + 0.90} \approx 92.4\% \quad B : F_1 = \frac{2 \cdot 0.98 \cdot 0.85}{0.98 + 0.85} \approx 91.0\% \tag{2.2}$$

So, with F_1 score as the single evaluation metric, we should pick classifier A.

	Accuracy	Time/ms
Classifier A	90%	80
Classifier B	92%	95
Classifier C	95%	1500

Table 2: Three image classifiers

2.2 Satificing and optimizing metrics

Often, it is not possible to include all the relevant metrics into one. We can get around this problem by deciding on a metric that we want to optimize, and for the others decide on a performance that is good enough. The latter as known as satificing metrics.

2.2.1 Example: Image classification

We have trained three image classifiers, and their accuracies on the dev set as well as run time is shown in table 2.2.1. We want to optimize accuracy with run time as a saticficing metric, so that is should be below 100 ms.

The satisficing condition disqualifies classifier C, even though it has the highest accuracy. Of the two that are left, classifier B is picked, as it has the highest accuracy.

2.3 Distributions of dev vs. test sets

In a sense, out goal is set by choosing the metric/loss function along with the dev set. Together, these can be thought of as a dart board target. Through the idea $\rightarrow$ code $\rightarrow$ experiment cycle, an ML project team can get progressively closer to hitting the bullseye of this dart board.

But when the time comes to checking the performance at the test set, we might be in trouble if we have not chosen it from the same distribution as the dev set! If these differ, we have actually ”moved the target” for the test set, and hence the performance will suffer.

2.3.1 Example: Geographical differences

Let’s say our data are divided into geographical regions: North America, South America, Europe, Asia, Africa, Australia. Now, imagine we choose our dev and test sets as follows:

- Dev set: North America, South America, Europe
- Test set: Asia, Africa, Australia

Here, we would be in trouble, since these represents to "different targets" (two different distributions): Good performance on the dev set, may not correspond to good performance on the test set!

Instead, all of the data should be shuffled, and randomly distributed between the dev and test sets. Then the two sets come from the same distribution.

2.3.2 Size of train, dev and test sets

Earlier, it might have been adviced to have a split in size between the three sets be 60% training set, 20% dev set, and 20% test set (or figures around these lines).

However, in the big data era, we often have a so much data, that there's no reason to make the dev and test percentages so large.

The test set should be large enough, that it ensures confidence in the performance of the model. The dev set should be of comparable size. Exact measures depend on the problem, but if, for instance we have a million data points, 1% is 10000 samples! Often, this will be enough for confidence in the model. So, a reasonable split for such modern data sets is often closer to 98% training set, 1% dev set, and 1% test set.

Sometimes, you may even forego having a test set, and simply work with a dev set (which confusingly, is then sometimes referred to as the test set). This is generally not recommendable, but may work nonetheless.

2.4 When to change metrics and/or dev/test sets

Sometimes, we find that the target we have set is simply not appropriate for what we wish to achieve. This section offers some insight into detecting such a problem, and how to deal with it.

2.4.1 Example: Cat image classifier

Let's say, you're building a cat app, and have made two binary classifiers for distringuishing between cat and non-cat pictures. You have chosen the - appearantly perfectly reasonable - optimization metric to be classification accuracy.

Now, on the dev set classifier A has an accuracy of 3%, while classifier B has 5%. So initially classifier A is chosen. However, it turns out, that classifier A lets some pornographic images through! This is of course unacceptable for a cat app.

We are now in a situation, where the "target" set (i.e. the metric and dev set) prefers classifier A, while you/the app users prefer classifier B. This is a sign that it's time to move the "target"!

2.4.2 Changing the metric

In the example above, we might choose to penalize the mis-classified pornographic images much heavier than other errors. Let's see how. The original metric to be minimized is:

$$\frac{1}{m}\sum_{i=1}^{m}\delta_i \tag{2.3}$$

Here m is the number of samples in the dev set, and δ_i is either 0 or 1, depending on whether or not the sample is mis-classified or not.

Now, we introduce a weighting factor w_i, depending on whether or not the image is pornographic in character:

$$\frac{1}{m}\sum_{i=1}^{m}w_i\delta_i \tag{2.4}$$

Here, we set w_i equal to 1 for most images, and a larger number, like 10 or maybe even 100, for pornographic ones. Finally, we may wish to change the normalization constant, so we get a rating between 0 and 1:

$$\frac{1}{\sum_{i=1}^{m}w_i}\sum_{i=1}^{m}w_i\delta_i \tag{2.5}$$

Note the orthogonalization in this approach: First, we worry about changing the metric to something more appropriate. Then we go through development cycles to improve performance with regards to the new metric.

2.4.3 Another example: More cat images

Consider the same cat image classifier example as above. However, in this case, the problem is, that the images we've used for the dev/test sets are images collected on the internet. These are often taken by professional photographers, and so tend to be of much higher quality/clarity than ones taken by amateaurs, who are the typical users of the cat app.

So when deployed, the classifier actually does much worse than it did on the dev/test sets, since the real app images comes from a different distribution.

This would be a good time to change the dev/test sets (and maybe the metrics) to better fit the actual problem.

3 Comparison to human-level performance

3.1 Bayes error rate

The Bayes error rate (or Bayes optimal error) for a machine learning problem, is the theoretical lower bound of how well an algorithm is able to perform. This rate is usually not zero, as the quality of the data may not be sufficient to make a perfectly performant algorithm.

3.2 Avoidable bias

We are interested in estimating the Bayes error rate, since any algoritm we make will have an error rate higher than the Bayes error rate. The difference between Bayes error rate and actual rate shows how much room we have for improvement. This is known as the avoidable bias.

3.3 Human level performance as a proxy for Bayes error rate

Human level performance can mean many things. If we want to diagnose a patient based on x-ray images, the average human will not do too well. A trained doctor will do much better. And a team of trained doctors even better.

What all these have in common is, that since they are algorithms, their error rate must me at least as large as the Bayes error rate of the problem! Therefore, if we know the human error rate (or the best one if we have several, as in the example above), we have a limit on the Bayes error rate, and therefore also on the avoiadable bias.

3.4 Beyond human level performance

There is however no guarantee that human level actually is the Bayes error rate, though for some tasks it will be close. But often, it is possible to develop an algoritm that will have a lower error rate than the best human performance. However, in this region, it is trickier to guide the development. The reason is, that as long as the algoritm performs worse than humans, we have the following advantages when developing:

- We can get more data labelled by human with relative ease.
- We can gain insights from manual error analysis: "Why did a person get this right?"

- We can do a better analysis of bias and variance (see below).

3.5 What to focus on? Bias or variance?

Let's consider a task, for which our algorithm has an error rate of 8% on the training set and 10% on the dev set. The question is now, if we should focus our energy on correcting bias (lowering error on training set) or variance (lowering error on dev set).

If we know what the human level performance is, we can use it as a proxy for the Bayes error rate. This will be helpful in guiding this decision.

If the human error level is 1%, then our algorithm, isn't really doing a terribly good job on the training set: We know that the avoidable bias is at least 7%, while the correponding variance (the different between training and dev set errors) is 2%. Here, it would make sense to work on increasing the capacity of the algorithm. I.e. work on reducing bias.

On the other hand, if the human error level var 7.5%, then chances are, that our algorithm is actually doing pretty well (unless we have reasons to suspect the Bayes error rate is drastically lower than human performance). Here, working to reduce variance, for instance by regularization or gathering more data, would make sense.

3.5.1 Summary

As long as human error is a reasonable proxy for the Bayes error, it makes sense to make a decision based on comparing the following:

- The avoidable bias: The difference between (the best) human-level error and training set error.
- Variance: The difference between training set error and dev set error.

3.6 Beyond human-level performance

Here, making an informed decision as in the previous section is much harder: We can no longer use human-level error as a proxy for Bayes error, and hence, we are much more in the dark.

That does not mean that further improvements are not possible, but there's fewer clues to what might be an effective strategy.

4 Case study: Bird recognition

5 Error analysis

6 Mismatched training/dev/test sets

7 Learning from multiple tasks

8 End-to-end deep learning

9 Case study: Image recognition for driverless car