Debugging data science (working draft)

Zico Kolter

November 28, 2016

1 Introduction

The vast majority of this course, as is the case for most data science courses, is focused on methodology and algorithms. This is to be expected, as they are things that can be easily defined and taught, and they do form the "tool chest" with which you will attack most data science problems. But the real skill in data science lies much less in your encyclopedic knowledge of these different algorithms (though I believe that a broad understanding at the level of what was presented in this course is definitely critical), and much more about your ability to *solve actual problems* using these techniques. In practice, this means that one skill actually tends to dominate others when it comes to being a successful data scientist, the ability to *debug* data science approaches. I am fond of the following way of summarizing this: what makes someone an expert data scientist is not what they do first when analyzing a data set; it's what they do second, when that first thing doesn't work.

In writing this lecture I hope to distill much of what I've learned about applying data science within the last several years, though I'm also aware that these topics are really only learned by going through the motions yourself, hitting up against a barrier, and seeing if you can use these techniques to break them down. Thus, my hope is that while you may be able to immediately get some valuable information from this lecture, them majority of its utility will come in being able to refer back to it as you hit up against similar problems in the future.

1.1 Traditional debugging

Debugging "traditional" programs if a relatively well-established idea. While there are a lot of formalisms here that I won't discuss, the following illustrates my own approach to debugging: I have some set of test input/output cases that a program is supposed to produce. For instance, if I'm writing a sorting routine, then a test case would be a random array, and the desired output would be the sorted version of that array. I also have a mental model of how my algorithms is supposed to work: each step should produce certain desired properties of the intermediate products of the algorithm, and these should obey certain properties. Again, thinking of the sorting example, if I'm implementing a quick sort algorithm, then a substep will be to select a pivot, and the next step will be to shuffle the array such that everything preceding the pivot is smaller and everything following it is larger. Note that this can all be formalized via pre and post conditions, and maybe some people do this more formally even when developing their own programs, but I know that in my own case, these steps are just more a mental model that I have in my head (more in a sec on what happens if this mental model is wrong).

Given this setup, the debugging process is fairly straightforward. If the program does not produce the desired output (which may include, e.g., crashing) for a given input, I follow an

execution trace of the program (either by stepping through with a debugger or just via print statements), and I check to see where the actual variables in the program are diverging from my mental model. Additionally, if there is something *wrong* with my mental model (a common occurrence, since after all it is usually just based upon my own vague understanding, at least at first), then this same process is useful for isolating very specific flaws that must have been in my own understanding of the problem.

1.2 Data science debugging: an example

Let's compare this now to a typical data science debugging problem. Let's say that I am working for a large industrial manufacturer, and I would like to write a program to determine which of the many machines that I have in a factory is likely to experience a failure in the next 3 months (I've worked on similar projects in the past, though I'm making this case generic to avoid any specificity to a given domain). Clearly, if I were to know this in advance, I could send a repair crew to fix the system *before* it actually experiences the failure, potentially allowing for substantially more uptime than would otherwise be possible. This is sometimes known as a "predictive maintenance" problem, and it is a big business is a lot of industrial control and operations. At my disposal I have a number of sensors that monitor each piece of equipment, and I have stored time series of the historical readings for each of these sensors, for every relevant piece of equipment in my plant; I also have work order history on the machines, and records of when each of them have failed in the past.

So I go ahead and look to take a data science approach to this problem. I can look back in time at all the readings that these machines had in the past, and (for all dates further back than 90 days), I can create a "label" for each machine, for each day, to indicate whether it indeed experienced a failure or not within the subsequent 90 days. This then becomes a simple binary classification problem: input features are sensor readings, either just at the specified time, or also including some vector of past readings as well; and output label indicates whether or not the machine subsequently experienced a failure.

I fire up whatever my favorite classification may be (I'd start with just linear classification, but could also soon move to something like RBF-features, gradient boosting, or a neural net) ... and the resulting classifier performs terribly. Classification performance on a held out test set is very poor, not much better than randomly guessing based upon just the prior probability of any machine experiencing a failure. From an applied standpoint, obviously this classifier is not at all useful.

So, what do I do next? From our discussion of machine learning, there appear to be many possible options: we could create more or different (non-linear) features; we would try a different machine learning algorithm; we could gather more data. Or, maybe it doesn't matter what we do, because the problem simple is not possible: there is no signal in our input data that can reliably predict much better than random chance, whether a machine is likely to fail or not within 90 days. Maybe failures are completely spontaneous and happen at a much higher frequency than we can detect (and detecting an imminent failure with seconds left to go may be ultimately pointless anyway, if the mitigation process doesn't allow for any action to be taken in this time period). How can we figure out whether or not this is the case, and if not, how do we take steps to improve the classification performance?

This is the fundamental characteristic of debugging data science that is to different from traditional debugging. We may have input/output pairs (this portion people are reasonably good at setting up), but we don't know if there exists *any* algorithm that can produce the desired output from the input. And we often have no mental model about how the system "should" work, because

after all this is what the machine learning algorithm is "supposed" to figure out. So our old traditional model of debugging, where we step through each line of code and check the current state of our program, is not going to work: it doesn't help to step through the lines of a logistic regression classification algorithm; the *algorithm* itself is probably fine (especially if we're using an existing library), but it's the underlying data itself, or the hyper parameters supplied to the algorithm, that are causing the problem.

1.3 Aside: debugging machine learning vs. debugging data science

Although I'm going to frame this lecture as "debugging data science" there any many elements that are similar to "debugging machine learning." Andrew Ng has a lecture on debugging machine learning algorithms (slides available here: `http://cs229.stanford.edu/materials/ML-advice.pdf`) that shaped a lot of how I think about debugging machine learning algorithms. This is a fantastic lecture, and worth looking over in detail (you can find various videos of the lecture online), but as a lecture primarily focused on the machine learning component, it's a bit narrower (though of course related to, and some portions of this lecture are taken from that source), as it often deals with problems (like whether to use logistic regression or an SVM,) that are not the typical problems you'll encounter in data science applications, where choices like that honestly typically have very little effect. In contrast, the data science debugging I'll discuss here also centers around the problem of creating the machine learning problem in the first case, and checking whether this actually solves the underlying data science problem we really care about.

2 The first step of data science debugging: determine if the problem is impossible

There is not going to be any "magic" here, and the process of debugging data science is frankly the hardest skill you'll need to acquire as you progress as a data scientist, but as you may expect, the process of debugging data science largely involves trying to covert the (seemingly opaque) process of figuring out why or why not a data science pipeline is working, to something more akin to the "traditional" debugging process. This will involve writing diagnostics and tests to evaluate whether or not the process really does line up with our own understanding.

With that in mind, the first step to debugging a data science problem is typically trying to find an answer to the question: is the problem I have set up possible at all? Is there sufficient signal in my input to achieve the desired level of prediction performance? Or am I effectively trying to predict the output of a random number generator?

Here's a basic rule for getting a handle on this question: after defining your input/output data, see if you can solve the prediction problem manually. By this, I mean you yourself should take on the role of the machine learning or other statistical analysis algorithm you're trying to use. Consider an input/output pair, look only at the input data, and see if *you* can predict what the output should be.

This is a very simple step, but students often ignore this task because machine learning algorithms are taught as being somehow "superhuman" in their abilities; it doesn't matter if we ourselves can solve the problem, the thinking goes, because a machine learning algorithm will be infinitely more capable at making decisions based upon all the available data. But this line of thinking simply isn't true in most cases. Yes, there are some cases where machine learning algorithms can make better predictions than human experts (especially when the goal is to produce accurately calibrated probabilities of events occurring, which humans are famously bad at). But

when it comes to a "first pass" of a machine learning algorithm on some data science problem, a human expert will almost always outperform the algorithm. And believe me, even if you're not an expert in your problem domain before you get started, you *will* be an expert in the domain soon enough.

So, try to solve the classification problem yourself. For example, write a little utility that will show you inputs from your problem, and asks for your predictions about what the output should be. Granted, for everything but problems with very small numbers of features, showing yourself the raw inputs you provide to a machine learning algorithm, as a simple vector of numbers, is likely not going to be that useful. Instead, you want to produce some sort of "visualization" of the input data itself (I put visualization in quote here, because often it's not really a visual illustration, but just a way of organizing the data so that you can quickly identify what may be the most relevant features). If some of your input is a list of human-understandable features, show these in a spreadsheet-like format (feature value and value, if there are too many features use your own knowledge the of task to find what you expect to be maybe the 10-20 most relevant ones); if your input data is a time series, display a plot of the data over time as your "input"; if it's image data, plot the image; if it's free text, display the text itself (or if you alternatively want to know if bag-of-words representations are enough, display the words that occur in the text). And given all this information about each example, see if you can predict what the output should be. If you can, then there is a good chance that your problem is solvable. If you can't make good predictions (with some caveats that I mention below), there is no reason to expect that a machine learning algorithm will be able to do any better. Put simply: the power of machine learning is not it's ability to achieve superhuman performance (though this is sometimes possible), it is its ability to achieve human-expert-level performance at a vastly increased scale and speed.

Naturally, there are cases where you won't be able to immediately predict the correct output on a test set without first understanding your data itself (though there are some domains, like vision or free text situations, where we are already "pre-trained" to do very well on these tasks). So in making these predictions, just like a machine learning algorithm, you should have at your disposal a "training set" upon which you can base your prediction. This training set can look much like the examples you're trying to predict, except they also have the true outputs attached. You are free to refer back to any collection of this training data when making your predictions (again, this is exactly the data that a machine learning algorithm will have available to it), and you probably also will want to look at plots of individual features versus.

At this point, there are two possibilities: either you are able to make reasonable predictions on your data set (what I'll call a "feasible" problem), or you can't (what I'll call an "impossible" problem, with quotes being there for a reason). Most descriptions of how to debug machine learning algorithms focus entirely on the first case, where the problem is in fact feasible, but the single most frequent case I see in practice is the latter, where people are trying to solve a problem using machine learning that even they cannot solve. We'll go through what to do in both these cases below.

2.1 Some additional details

Before we move on to how to handle these these two cases, there are a few items worth mentioning. While I've formulated the problem above as "make the prediction yourself" it's also common to encounter situations where human fallibility can make this a difficult problem, even when it's not that hard for a computer. Two common cases where this emerges are the cases of imbalanced data sets and regression predictions.

In imbalanced data sets, one class label (i.e., one type out output) may be a whole lot more likely than the other. We haven't discussed, in class, how to handle imbalanced data from a

machine learning perspective, because even for reasonably unbalanced problems (say a 10:1 ratio of occurrence), machine learning algorithms often do much better than people assume. However, humans are admittedly very bad at doing this, and if most of the samples in the script you create are all of one type, then chances are you'll start to zone out and miss whatever important features there are. So when you do set up the script to see if you can predict the labels of your problem, you probably want to select equal numbers of examples from each different class. However, be aware that this can create problems to, for instance if "random" examples from your (frequent) negative class don't look anything like the (infrequent) positive class. Returning to our predictive maintenance problem, for instance, it may be that there are signals that occur frequently before a failure, yet they also occur just as often with no failure at all. If these signals themselves are rare, then by randomly sampling negative examples, you may miss all the cases, and conclude that you could use the signal to predict failure, but this would be a mistake. Thus, in selecting samples to manually classify, you want to include positive (e.g., failure) samples, but the negative samples that "most closely" resemble the failure cases.

Regression problems can also be quite difficult to predict manually, simply because we are often quite bad at handling a continuum of possible prediction options (the same problem occurs in . In these cases, it is advisable to try to either discretize your predictions (so instead of trying to predict the exact output, just try to predict if the output is above or below some threshold. Alternatively, it can also be useful to consider a "ranking" setting, where instead our goal is to consider two cases at once, and try to predict which one has the higher output. Even when we can't predict absolute quantities, this ranking task can often be easier, and if it's possible, it still provides strong evidence that the prediction task can indeed be solved. Indeed, it's often the case that if you can solve this ranking problem given some set of inputs, then the machine learning algorithm will be able to solve the underlying regression problem using the same inputs.

3 What to do with an "impossible" problem

So you have now gone through the exercise described above, built a script that shows your data inputs in a human-interpretable form, referred back to the training set when making predictions. You have manually tried to classify some reasonable number of examples (maybe a hundred or so) ... and you find that you are entirely unable to make good predictions (or at least, your performance is nowhere near as good as you had hoped, to the level that would make a useful tool).

What do you do now?

What you should absolutely *not* do is try to throw more machine learning algorithms at the problem. For all the reasons stated above, throwing more horsepower at an problem is very unlikely to give "step change" improvements, moving an impossible problem to a feasible one.

The most tempting answer, in fact, is perhaps "give up"; or, if you've been assigned the problem as part of your job, take the script you wrote and have your boss/advisor (whoever assigned you the problem) try to make the predictions themselves, until they are also convinced that the problem is impossible. And there are some cases where I would argue that everyone would be a lot better off if you are able to identify impossible problems quickly, so as not to waste any more efforts on them. But assuming you don't want to completely abandon the task, then there are a few other options available to you. The aim of all these options is to change the problem so that a formerly "impossible" problem becomes "feasible". And there two main ways to go about this, which involve, not surprisingly, either changing the inputs to you problem (i.e., gather more features), or changing the outputs to your problem (i.e., changing the problem definition).

3.1 Gathering more, different, features (i.e., changing the inputs).

This is of course the holy grail of any data science problem. Maybe the problem is impossible as you currently have it set up, but maybe there is *just one more* feature or data source you missed, that will suddenly render the problem completely solvable. The fact that this is always possible in theory is what requires that we describe these problems as "impossible" (with quotes) instead of outright impossible (no quotes). And a lot of people will hold out hope for very long that this may be the case (see the number of papers that claim to predict the stock market based e.g. on Twitter feeds). The problem in most cases it that going about actually adding features is an ill-defined task, and you'll end up wasting a lot of time looking into new data sources that *may* have some hypothetical correlation with the output, but which in practice probably won't help very much. To mitigate this as much as possible, I'll offer two pieces of advice, but these are no guarantee of success, just general advice to avoid much wasted time.

1. Spot check your new data sources. Before spending time integrating a new feature into your pipelines and visualization (and certainly before spending time putting it into your machine learning algorithm), do some quick "spot checks" to see if the examples you were previously predicting incorrectly can be obviously corrected by including this one or more new features. If there doesn't seem to be any obvious correlation, or no more than what you're already capturing in your predictions, don't bother. Another interesting (though a bit more involved) test is to see if you can predict this new feature *given your existing data*; if this is the case, you clearly already have all the information contained in the new feature, so it will not help to add it to your data.

2. Consult with real domain experts when possible. If people are already attempting to solve this problem manually, find out what data sources they are using to do so. Make sure that each of these data sources is available to your own manual classification script, in a way that makes it easy to visualize what are the salient characteristics that the domain experts actually rely upon.

Ultimately, you'll have to make the call, after consulting available data sources and experts, about whether you are satisfied that the problem as you have framed it so far really cannot be solved (by you, and thus very likely not by a machine learning algorithm).

3.2 Changing the problem (i.e., changing the outputs).

The alternative to changing the inputs, of course, is to change the outputs of the problem. And while often times it's not obvious how to do this while preserving the character of the underlying problem (if we change the problem too much, a solution ceases to a be a solution to the underlying question we actually care about). But often time is it *is* possible to change the problem or the goal of the analysis more than you may expect, while still delivering a useful piece of analysis. There are two primary ways I've seen this done successfully.

1. Change task from "predicting the future" to "predicting as well as a human expert". The first strategy is to try to force the problem to be a "feasible" one by the criteria that we defined above. Instead of trying to "predict the future", which is the goal of many data science problems and one that may well be impossible due to the inherent uncertainty of the real world (at least at the level that we are every able to model it at), try to instead predict as well as a human expert *given only the data available to the algorithm.* In other words, have a team of true experts walk through your script that attempts to predict the outputs, look at all the features they are using, and try to predict *these* labels instead of the actual future events (because, for the human experts, we

already have an existence proof that the problem is feasible, at least to a certain degree). Although it's possible that this will no longer get at the problem you really care about, if you're working in a domain where there has been any amount of past manual work, chances are that there is some useful knowledge that these experts have, which would be valuable to encode in an automated system.

You may find that this subtly changes the characteristic of the problem you're considering. For example, in the predictive maintenance problem, the task may change from "predicting future failure" to "predicting what a maintenance crew (with infinite time and patience to analyze every single machine and signal) would choose to work on". The latter may not have the "magic" feel of the former, but it is very likely still a task that has substantial practical value.

2. Move from "prediction" to "characterizing uncertainty". This may sound like a cop-out, but often times there is a great deal of value in moving from the concept of "prediction" as a black box that will somehow give you high accuracy on whatever task you want, to the idea of prediction as way of generating a distribution over possible outcomes. As mentioned above, there are many real-world tasks (I would say virtually all of them, in fact) where perfect or even "high sounding" accuracy is simply not possible: there is inherent uncertainty in the problem that makes accuracy beyond a certain threshold literally impossible given the inputs you are using (there is still always the caveat that we could find some hidden magical feature that allows for perfect prediction, but this usually won't happen in practice). But often it can be extremely valuable to characterize this conditional uncertainty over the outputs given the inputs as accurately as possible. This is indeed what the majority of statistics is about, to some degree: they have understood for a long time that simple "prediction" is not always the goal here, and topics like "analysis of variance" or "explained variance" get at this precise quantity: how much uncertainty is there in our prediction compared to the raw output proportions in the data (i.e., the "predictions" you would make if you had no input features at all).

Characterizing uncertainty is much easier in the setting of simpler linear models, and more complex machine learning models like neural networks will frequently create "probabilities" that don't accurately reflect the true underlying model, so you often want to resort to simpler models here, but the end result can still be quite valuable. For example, in our predictive maintenance problem, if 10

By following these two elements, you should be able to transform problems from the "impossible" setting to the "feasible" one. If you can't do this (i.e., you have no additional features to add, and changing the problem to match with expert opinions doesn't produce a useful result), then it's not a bad idea to think about moving on to a different problem entirely.

4 What do to with a "feasible" problem

Good news You have found (or really, created) a problem where you're able yourself to manually figure out what the right prediction "should" be, at least to a reasonable degree, and now you want to use machine learning to match or improve upon your performance. This is now a return to the domain of "traditional" machine learning debugging, and there are a lot of avenues.

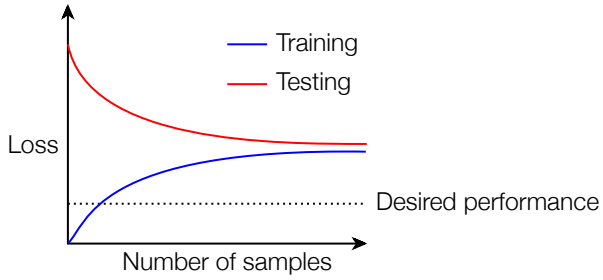
As you went through the process of manually classifying examples, you probably started to come up with a lot of "rules of thumb" about how to predict your outputs. These rules of thumb fill exactly the same role as the "mental model" I spoke about before in the context of traditional programming. You will start to develop an (admittedly simplistic) picture of what features the algorithm "should" be looking at to make its decisions. Now, all of a sudden, when you do apply a

machine learning algorithm, you can look at what information the algorithm looks at (by looking at the weights of a linear classifier, for instance, or the thresholds and rules of a decision tree), and seeing if the algorithm is matching what you expect. If not, then this provides a way forward for developing new features, or for adjusting the problem description itself, in order to ensure that you *are* able to accurately predict what is happening.

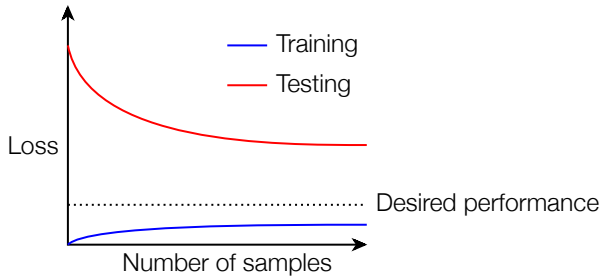
So let's discuss the failure mode that can occur here: you are able to predict the problems reasonably well through (possibly very time-consuming) manual inspection, but when you run a machine learning on the problem set, it doesn't even meet your own performance. What do you do now? The key here is that, just like before, instead of just throwing different algorithms, more data, more features, different hyperparameters, run the algorithm for longer, etc it helps to have concrete diagnostics that you can run which inform you about the best course of action.

1. Understand bias/variance tradeoffs of classifier. If the performance of your machine learning classifier is not sufficient, it helps to understand whether the problem is one of bias (a model which cannot capture the underlying decision rule), or variance (a model which is too powerful, and thus likely to overfit to the training data). The key way to differentiate between these two scenarios is to look at both the training loss *and* the testing loss. If training loss is low and testing loss is unacceptably high, then your problem is one of high variance; if training loss and testing loss are both high, then your problem is one of high variance.

A slightly more nuanced view of this same picture is to consider a plot of training and testing performance versus the number of examples included for training (you can use only a subset of your training data to train "simpler" models). If the figure looks like this:



then your problem is one of high bias; both the training and test losses are unacceptably high, and have seemed to both plateau in terms of the number of samples. If, on the other hand, your figure looks like this:



then your problem is one of high variance; performance is reasonable on the training set, but not on the test set, and is improving (somewhat) with added data.

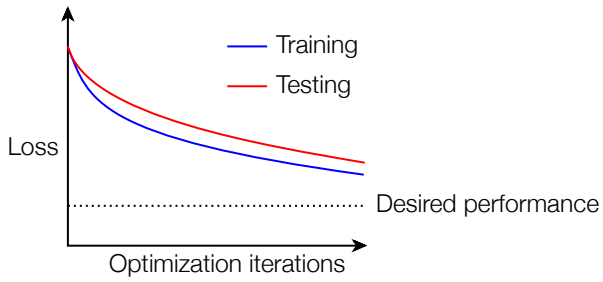
What do you do in each situation? For problems of high bias, the key issue is that the machine learning algorithm is not able to perform well *even on the training set*. Because you already know from your own analysis that the problem is "solvable" in some sense (i.e., you could write

start to think of some rules of thumb that should guide good classification), and because the machine learning algorithms typically do a pretty good job of searching for the function within their hypothesis class that achieves minimum error on the training set (more later on when this may not be the case, but it is usually a reasonable assjmption), then it is likely the case that whatever "rules of thumb" you came up with on to do the classification are not being encoded in the features available to the classifier. You should consider additional additional features that *do* encode these rules of thumb *directly as features*, and make sure the classifier is able to use these to determine its classification.

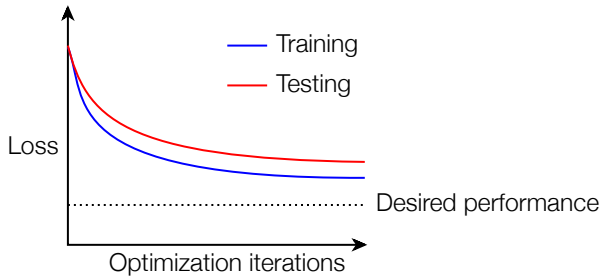
For problems of high variance, we need to take a different set of approaches. In this case performance on the training set looks good, but not on the test set. In this case it may be beneficial, for example, to simply collect more training data, because in the limit of infinite training data, training and testing losses should be identical. This can be a great solution (indeed, much recent success in machine learning has been driven by the availability of data as much as by new algorithms), but be aware that this can be a time-consuming process, and a good heurstic to keep in mind is that you will typically need *orders of magnitude* more training data to overcome significant variance problems by adding data alone. Furthermore, because the capacity (i.e., the space of allowable hypotheses) for many of these complex algorithms like gradient boosting or deep network is extremely large, it can often seem as though no matter how much more data you feed into the system, training loss will always substantially outperform testing loss.

An alternative approach for high variance, which I would recommend you try first, is to actually *reduce* the complexity of your model until you reach a point where the higher bias in your concern. Select only a subset of the features to feed to the algorithm, again focusing on those that can most directly encode your rules of them, or scrap the "raw" data features entirely and *just* build a classifier with your rule of thumb predictions as features, then incrementally add some seemingly-important raw features until you start to see some improvement over your original baseline.

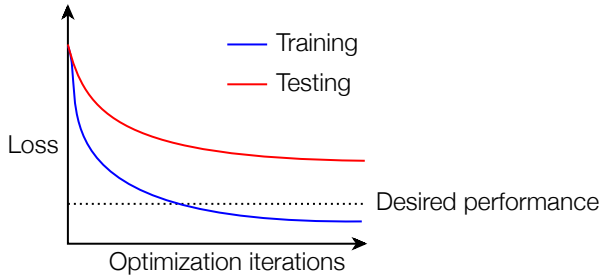
2. Understand the optimization behavior of the algorithm. This is rarely the problem in most data science tasks (off-the-shelf algorithms are fairly "good" at achieveing a reasonable optimum), but one thing to keep in mind is the possibility that your machine learning algorithm is not converging to the best loss it can actually achieve, and you may need to run it longer (more iterations of gradient descent, more boosting iterations, etc). To assess this, you should plot the training and testing loss of your algorithm versus the number of optimization iterations (these are usually accessible interally for most machine learning libraries). If you get performance that looks like this



then by all means keep running more iterations of the machine learning optimization routine. But more than likely, your figure will look like this

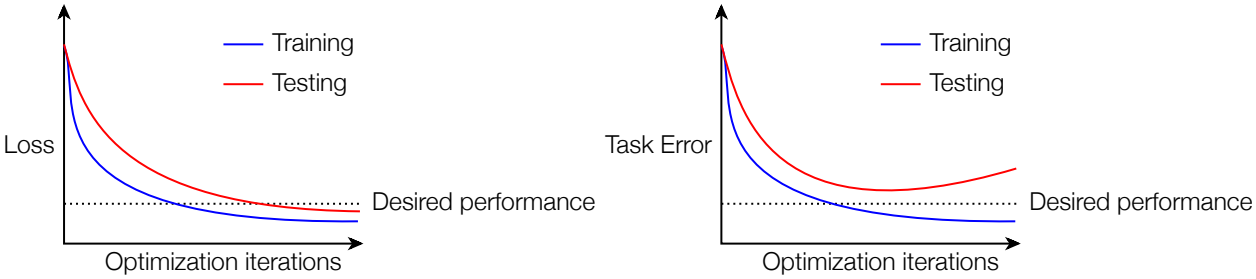


(the high bias case), or this



(the high variance case). In these situations, running additional optimization iterations is not going to help.

3. Understand loss of the algorithm versus the true "error" you care about. Remember that virtually every machine learning algorithm can be viewed as trying to minimize a loss function, which is typically *not* the actual loss function you ultimately are about from an applied point of view. To see if this may be causing some of the difficulties, you plot one of the some figures above, once for the loss that the algorithm is actually optimizing, and once for the error that you casre about. In the case that the two figures diverge, e.g.



then you likely need to rethink the loss that you are optimizing in your machine learning algorithm. This doesn't usually mean switching from (say) logistic loss to hinge loss, but it more frequently means that you need to weight the positive or negative examples differently, or attach example-specific weights based upon some characteristics of each data point. This is fairly common, for instance, when you are dealing with imbalanced data sets, where the task-specific error of one type of misclassification is substantially more than the other; in these cases, you should tune your loss function to also capture these rough costs. Most machine learning libraries allow you to specify different weights for different types of misclassications or for different examples.

4. Go get a Ph.D. I'm being facetious here, but there is one remaining possibility. There is a very remote chance that, even though you have discovered a prediction task where you can figure

out the correct predictions yourself, and you simply cannot find a machine learning algorithm that can solve it. If this is still the case, after exhaustively building all the above diagnostics, then there is a possibility that you happen to have stumbled upon an "AI Hard" problem, a problem involving some characteristic of intelligence that humans are very good at, yet which we still don't have good machine learning algorithms for. Many "core" vision, speech, and natural language processing tasks are of this nature, despite the impressive progress that that has been made using deep learning techniques in recent years.

In this case, you probably *weren't* able to come up with good succinct rules of thumb that can explain your own prediction methodology, but instead they relied on more intuitive measure like what you saw, heard, or understood about the example as you originally visualized it in your prediction script. These are, of course, the most "interesting" problems in some sense, but as I have emphasized throughout this course, they are a very small fraction of what you will actually encounter in practice.

In the astonishing event that you don't want to abandon your current career path to pursue an academic life, you can alternatively try to use many of the same techniques that we used previously to redefine your problem to be one where machine learning has a real possibility. Construct "simple" features from your intuitive-but-hard-to-describe data (for instance, replace free text with its bag of words representation), and see if you can frame the problem in a manner that can be solved with this new representation. as one that

5 The possibility of unreasonably good performance

Because I know you won't actually wait to do any machine learning until you develop a nice visualization interface and script for classifying your data, there is another remaining possibility. There is some chance that machine learning will perform great on your task right off the bat. Unless you know the problem you have set up is extremely simple (i.e., one where you already know that you could write a couple of rules of thumb that would also likely get very good performance), you should be *skeptical* about any such result. Do *not* take the result for granted, assume that machine learning "solves" your problem, and pass off the project or method to someone else. Instead, still go through the exercise of manually classifying the data just like before.

If you go through this exercise and find that you *cannot* perform well on this task, despite the fact that the machine learning algorithm does, you should now be *extremely skeptical*. You need to figure out why this is the case: what sort of predictions is the machine learning algorithm making that you cannot figure out yourself? If your machine learning algorithm really does work, then by inspecting the weights or more important features of the system (or for instance, running a decision tree to get a more explainable set of rules directly from your classifier), you should be able to understand its internal logic to some small degree. This may be slightly less true of domains that require deep learning, as these models are famously opaque, but here there should be some other way of understanding your data visually or otherwise that still makes the task apparent to you, or at least to the point where you could imagine the task being apparent to an expert in the field.

I bring this all up because in these settings there is, sadly, a very good chance that you've set up your problem incorrectly in a manner that lets the algorithm "cheat" somehow. Sometimes this is done by a simple bug: you may include a feature in the data that is essentially a virtually deterministic one-to-one mapping to the desired prediction. Sometimes this is allowed and you have effectively "solved" your problem by looking at this one features, but more often than not, you included the feature by mistake, and it's not something that would actually be available to

the system if you deploy your algorithm "in the wild". Or, somewhat equivalently, it doesn't allow you to draw any interesting conclusions if you're just isolating this obvious relationship. In a less obvious setting, I have seen many cases like the predictive maintenance examples, where companies will happily "update" historical sensor information to account for the fact that there was a failure in the future (something along the lines of "we know in retrospect that we couldn't trust these signals, since the machine was about to fail, so we set them all to zero"). The data you are looking at now may *not* be what it actually looked like before the failure, and so any machine learning algorithm that uses this "updated" data is effectively using an unallowable signal from the future.

If this is the case, then by going through the exercise of classifying examples manually, with the trained machine learning model in hand, you'll likely be able to spot some "odd" uses of features that don't make intuitive sense to you (or more importantly, which don't make intuitive sense to domain experts). Now, it is certainly possibly that you have stumbled upon a previously-unknown signal with high predictive power. But you will want to analyze each of these in detail, verify that these signals are available in real-time, and (ideally) actually test the system in a deployed environment, all before you declare "success".

6 Conclusion

This document has been an attempt to synthesize some of the techniques I have found to be most useful in debugging my own data science projects over the course of my career. The key findings are summarized in the following chart, which I find to be a useful summary of "what to do" when you encounter a challenge that you can't overcome in a data science problem.

	ML predicts well	**ML predicts poorly**
You predict well	Congratuations, (though still do some analysis to see what might be the cause of this good performance)	"Feasible" problem, debug the machine learning algorithm
You predict poorly	Be extremely skeptical, analyze the results to see what the algorithm is doing	"Impossible" problem, debug the problem until is it feasible

Another faculty member at CMU tells me a story about a paper on "practical systems debugging" he assigns in his course. He says that whenever he gives this paper to anyone who has worked in the field for several years, they find it incredible, talk about how they wasted this many years of their life on this issues raised in this section, this many years on issue issues raised in that section, etc; how different their career would have been had they only had a chance to read this paper early on in their career. When he assigns it to students in his course, however, the reaction is typically "this is all obvious."

The only way to really learn the debugging process of data science is by doing it, and going through a short paper is not any substitute for hands-on experience. But hopefully these notes give you a little bit of guidance as to what directions you could try when you are completely stuck and have no idea what your next step should be. The rest will come with time.