Rules of Machine Learning:

Best Practices for ML Engineering

Martin Zinkevich

This document is intended to help those with a basic knowledge of machine learning get the benefit of Google's best practices in machine learning. It presents a style for machine learning, similar to the Google C++ Style Guide and other popular guides to practical programming. If you have taken a class in machine learning, or built or worked on a machine-learned model, then you have the necessary background to read this document.

Martin Zinkevich introduces 10 of his favorite rules of machine learning. Read on to learn all 43 rules!

Terminology

The following terms will come up repeatedly in our discussion of effective machine learning:

- **Instance**: The thing about which you want to make a prediction. For example, the instance might be a web page that you want to classify as either "about cats" or "not about cats".
- **Label**: An answer for a prediction task either the answer produced by a machine learning system, or the right answer supplied in training data. For example, the label for a web page might be "about cats".
- **Feature**: A property of an instance used in a prediction task. For example, a web page might have a feature "contains the word 'cat'".
- **Feature Column**: A set of related features, such as the set of all possible countries in which users might live. An example may have one or more features present in a feature column. "Feature column" is Google-specific terminology. A feature

column is referred to as a "namespace" in the VW system (at Yahoo/Microsoft), or a field (https://www.csie.ntu.edu.tw/%7Ecjlin/libffm/).

- **Example**: An instance (with its features) and a label.
- **Model**: A statistical representation of a prediction task. You train a model on examples then use the model to make predictions.
- **Metric**: A number that you care about. May or may not be directly optimized.
- **Objective**: A metric that your algorithm is trying to optimize.
- **Pipeline**: The infrastructure surrounding a machine learning algorithm. Includes gathering the data from the front end, putting it into training data files, training one or more models, and exporting the models to production.
- **Click-through Rate** The percentage of visitors to a web page who click a link in an ad.

Overview

To make great products:

do machine learning like the great engineer you are, not like the great machine learning expert you aren't.

Most of the problems you will face are, in fact, engineering problems. Even with all the resources of a great machine learning expert, most of the gains come from great features, not great machine learning algorithms. So, the basic approach is:

1. Make sure your pipeline is solid end to end.
2. Start with a reasonable objective.
3. Add common-sense features in a simple way.
4. Make sure that your pipeline stays solid.

This approach will work well for a long period of time. Diverge from this approach only when there are no more simple tricks to get you any farther. Adding complexity slows future releases.

Once you've exhausted the simple tricks, cutting-edge machine learning might indeed be in your future. See the section on Phase III (#ml_phase_iii_slowed_growth_optimization_refinement_and_complex_models) machine learning projects.

This document is arranged as follows:

1. The first part (#before_machine_learning) should help you understand whether the time is right for building a machine learning system.
2. The second part (#ml_phase_i_your_first_pipeline) is about deploying your first pipeline.
3. The third part (#ml_phase_ii_feature_engineering) is about launching and iterating while adding new features to your pipeline, how to evaluate models and training-serving skew.
4. The final part (#ml_phase_iii_slowed_growth_optimization_refinement_and_complex_models) is about what to do when you reach a plateau.
5. Afterwards, there is a list of related work (#related_work) and an appendix (#appendix) with some background on the systems commonly used as examples in this document.

Before Machine Learning

Rule #1: Don't be afraid to launch a product without machine learning.

Machine learning is cool, but it requires data. Theoretically, you can take data from a different problem and then tweak the model for a new product, but this will likely underperform basic heuristics. If you think that machine learning will give you a 100% boost, then a heuristic will get you 50% of the way there.

For instance, if you are ranking apps in an app marketplace, you could use the install rate or number of installs as heuristics. If you are detecting spam, filter out publishers that have sent spam before. Don't be afraid to use human editing either. If you need to rank contacts, rank the most recently used highest (or even rank alphabetically). If machine learning is not absolutely required for your product, don't use it until you have data.

Rule #2: First, design and implement metrics.

Before formalizing what your machine learning system will do, track as much as possible in your current system. Do this for the following reasons:

1. It is easier to gain permission from the system's users earlier on.
2. If you think that something might be a concern in the future, it is better to get historical data now.
3. If you design your system with metric instrumentation in mind, things will go better for you in the future. Specifically, you don't want to find yourself grepping for strings in logs to instrument your metrics!
4. You will notice what things change and what stays the same. For instance, suppose you want to directly optimize one-day active users. However, during your early manipulations of the system, you may notice that dramatic alterations of the user experience don't noticeably change this metric.

Google Plus (#google_plus_overview) team measures expands per read, reshares per read, plusones per read, comments/read, comments per user, reshares per user, etc. which they use in computing the goodness of a post at serving time. **Also, note that an experiment framework, in which you can group users into buckets and aggregate statistics by experiment, is important.** See Rule #12 (#rule_12_don_t_overthink_which_objective_you_choose_to_directly_optimize).

By being more liberal about gathering metrics, you can gain a broader picture of your system. Notice a problem? Add a metric to track it! Excited about some quantitative change on the last release? Add a metric to track it!

Rule #3: Choose machine learning over a complex heuristic.

A simple heuristic can get your product out the door. A complex heuristic is unmaintainable. Once you have data and a basic idea of what you are trying to accomplish, move on to machine learning. As in most software engineering tasks, you will want to be constantly updating your approach, whether it is a heuristic or a machinelearned model, and you will find that the machine-learned model is easier to update and maintain (see Rule #16 (#rule_16_plan_to_launch_and_iterate)).

ML Phase I: Your First Pipeline

Focus on your system infrastructure for your first pipeline. While it is fun to think about all the imaginative machine learning you are going to do, it will be hard to figure out what is happening if you don't first trust your pipeline.

Rule #4: Keep the first model simple and get the infrastructure right.

The first model provides the biggest boost to your product, so it doesn't need to be fancy. But you will run into many more infrastructure issues than you expect. Before anyone can use your fancy new machine learning system, you have to determine:

- How to get examples to your learning algorithm.
- A first cut as to what "good" and "bad" mean to your system.

- How to integrate your model into your application. You can either apply the model live, or precompute the model on examples offline and store the results in a table. For example, you might want to preclassify web pages and store the results in a table, but you might want to classify chat messages live.

Choosing simple features makes it easier to ensure that:

- The features reach your learning algorithm correctly.
- The model learns reasonable weights.
- The features reach your model in the server correctly.

Once you have a system that does these three things reliably, you have done most of the work. Your simple model provides you with baseline metrics and a baseline behavior that you can use to test more complex models. Some teams aim for a "neutral" first launch: a first launch that explicitly deprioritizes machine learning gains, to avoid getting distracted.

Rule #5: Test the infrastructure independently from the machine learning.

Make sure that the infrastructure is testable, and that the learning parts of the system are encapsulated so that you can test everything around it. Specifically:

1. Test getting data into the algorithm. Check that feature columns that should be populated are populated. Where privacy permits, manually inspect the input to your training algorithm. If possible, check statistics in your pipeline in comparison to statistics for the same data processed elsewhere.
2. Test getting models out of the training algorithm. Make sure that the model in your training environment gives the same score as the model in your serving environment (see Rule #37 (#rule_37_measure_training_serving_skew)).

Machine learning has an element of unpredictability, so make sure that you have tests for the code for creating examples in training and serving, and that you can load and use a fixed model during serving. Also, it is important to understand your data: see Practical Advice for Analysis of Large, Complex Data Sets (http://www.unofficialgoogledatascience.com/2016/10/practical-advice-for-analysis-of-large.html).

Rule #6: Be careful about dropped data when copying pipelines.

Often we create a pipeline by copying an existing pipeline (i.e., cargo cult programming (https://wikipedia.org/wiki/Cargo_cult_programming)), and the old pipeline drops data that we need for the new pipeline. For example, the pipeline for Google Plus (#google_plus_overview) What's Hot drops older posts (because it is trying to rank fresh posts). This pipeline was copied to use for Google Plus (#google_plus_overview) Stream, where older posts are still meaningful, but the pipeline was still dropping old posts. Another common pattern is to only log data that was seen by the user. Thus, this data is useless if we want to model why a particular post was not seen by the user, because all the negative examples have been dropped. A similar issue occurred in Play. While working on Play Apps Home, a new pipeline was created that also contained examples from the landing page for Play Games without any feature to disambiguate where each example came from.

Rule #7: Turn heuristics into features, or handle them externally.

Usually the problems that machine learning is trying to solve are not completely new. There is an existing system for ranking, or classifying, or whatever problem you are trying to solve. This means that there are a bunch of rules and heuristics. **These same heuristics can give you a lift when tweaked with machine learning.** Your heuristics should be mined for whatever information they have, for two reasons. First, the transition to a machine learned system will be smoother. Second, usually those rules contain a lot of the intuition about the system you don't want to throw away. There are four ways you can use an existing heuristic:

- Preprocess using the heuristic. If the feature is incredibly awesome, then this is an option. For example, if, in a spam filter, the sender has already been blacklisted, don't try to relearn what "blacklisted" means. Block the message. This approach makes the most sense in binary classification tasks.
- Create a feature. Directly creating a feature from the heuristic is great. For example, if you use a heuristic to compute a relevance score for a query result, you can include the score as the value of a feature. Later on you may want to use machine learning techniques to massage the value (for example, converting the value into one of a finite set of discrete values, or combining it with other features) but start by using the raw value produced by the heuristic.
- Mine the raw inputs of the heuristic. If there is a heuristic for apps that combines the number of installs, the number of characters in the text, and the day of the week, then consider pulling these pieces apart, and feeding these inputs into the learning separately. Some techniques that apply to ensembles apply here (see Rule #40 (#rule_40_keep_ensembles_simple)).
- Modify the label. This is an option when you feel that the heuristic captures information not currently contained in the label. For example, if you are trying to maximize the number of downloads, but you also want quality content, then maybe the solution is to multiply the label by the average number of stars the app received. There is a lot of leeway here. See "Your First Objective" (#your_first_objective).

Do be mindful of the added complexity when using heuristics in an ML system. Using old heuristics in your new machine learning algorithm can help to create a smooth transition, but think about whether there is a simpler way to accomplish the same effect.

Monitoring

In general, practice good alerting hygiene, such as making alerts actionable and having a dashboard page.

Rule #8: Know the freshness requirements of your system.

How much does performance degrade if you have a model that is a day old? A week old? A quarter old? This information can help you to understand the priorities of your monitoring. If you lose significant product quality if the model is not updated for a day, it makes sense to have an engineer watching it continuously. Most ad serving systems have new advertisements to handle every day, and must update daily. For instance, if the ML model for Google Play Search (#google_play_overview) is not updated, it can have a negative impact in under a month. Some models for What's Hot in Google Plus (#google_plus_overview) have no post identifier in their model so they can export these models infrequently. Other models that have post identifiers are updated much more frequently. Also notice that freshness can change over time, especially when feature columns are added or removed from your model.

Rule #9: Detect problems before exporting models.

Many machine learning systems have a stage where you export the model to serving. If there is an issue with an exported model, it is a user-facing issue.

Do sanity checks right before you export the model. Specifically, make sure that the model's performance is reasonable on held out data. Or, if you have lingering concerns with the data, don't export a model. Many teams continuously deploying models check the area under the ROC curve (https://wikipedia.org/wiki/Receiver_operating_characteristic) (or AUC) before exporting. **Issues about models that haven't been exported require an email alert, but issues on a user-facing model may require a page.** So better to wait and be sure before impacting users.

Rule #10: Watch for silent failures.

This is a problem that occurs more for machine learning systems than for other kinds of systems. Suppose that a particular table that is being joined is no longer being updated. The machine learning system will adjust, and behavior will continue to be reasonably good, decaying gradually. Sometimes you find tables that are months out of date, and a simple refresh improves

performance more than any other launch that quarter! The coverage of a feature may change due to implementation changes: for example a feature column could be populated in 90% of the examples, and suddenly drop to 60% of the examples. Play once had a table that was stale for 6 months, and refreshing the table alone gave a boost of 2% in install rate. If you track statistics of the data, as well as manually inspect the data on occassion, you can reduce these kinds of failures.

Rule #11: Give feature columns owners and documentation.

If the system is large, and there are many feature columns, know who created or is maintaining each feature column. If you find that the person who understands a feature column is leaving, make sure that someone has the information. Although many feature columns have descriptive names, it's good to have a more detailed description of what the feature is, where it came from, and how it is expected to help.

Your First Objective

You have many metrics, or measurements about the system that you care about, but your machine learning algorithm will often require a single **objective, a number that your algorithm is "trying" to optimize.** I distinguish here between objectives and metrics: a metric is any number that your system reports, which may or may not be important. See also Rule #2 (#rule_2_first_design_and_implement_metrics).

Rule #12: Don't overthink which objective you choose to directly optimize.

You want to make money, make your users happy, and make the world a better place. There are tons of metrics that you care about, and you should measure them all (see Rule #2 (#rule_2_first_design_and_implement_metrics)). However, early in the machine learning process, you will notice them all going up, even those that you do not directly optimize. For instance, suppose you care about number of clicks and time spent on the site. If you optimize for number of clicks, you are likely to see the time spent increase.

So, keep it simple and don't think too hard about balancing different metrics when you can still easily increase all the metrics. Don't take this rule too far though: do not confuse your objective with the ultimate health of the system (see Rule #39 (#rule_39_launch_decisions_are_a_proxy_for_long_term_product_goals)). And, **if you find yourself increasing the directly optimized metric, but deciding not to launch, some objective revision may be required.**

Rule #13: Choose a simple, observable and attributable metric for your first objective.

Often you don't know what the true objective is. You think you do but then as you stare at the data and side-by-side analysis of your old system and new ML system, you realize you want to tweak the objective. Further, different team members often can't agree on the true objective. **The ML objective should be something that is easy to measure and is a proxy for the "true" objective.** In fact, there is often no "true" objective (see Rule#39 (#rule_39_launch_decisions_are_a_proxy_for_long_term_product_goals)). So train on the simple ML objective, and consider having a "policy layer" on top that allows you to add additional logic (hopefully very simple logic) to do the final ranking.

The easiest thing to model is a user behavior that is directly observed and attributable to an action of the system:

- Was this ranked link clicked?
- Was this ranked object downloaded?
- Was this ranked object forwarded/replied to/emailed?
- Was this ranked object rated?
- Was this shown object marked as spam/pornography/offensive?

Avoid modeling indirect effects at first:

- Did the user visit the next day?
- How long did the user visit the site?
- What were the daily active users?

Indirect effects make great metrics, and can be used during A/B testing and during launch decisions.

Finally, don't try to get the machine learning to figure out:

- Is the user happy using the product?
- Is the user satisfied with the experience?
- Is the product improving the user's overall wellbeing?
- How will this affect the company's overall health?

These are all important, but also incredibly hard to measure. Instead, use proxies: if the user is happy, they will stay on the site longer. If the user is satisfied, they will visit again tomorrow. Insofar as well-being and company health is concerned, human judgement is required to connect any machine learned objective to the nature of the product you are selling and your business plan.

Rule #14: Starting with an interpretable model makes debugging easier.

Linear regression, logistic regression, and Poisson regression are directly motivated by a probabilistic model. Each prediction is interpretable as a probability or an expected value. This makes them easier to debug than models that use objectives (zero-one loss, various hinge losses, and so on) that try to directly optimize classification accuracy or ranking performance. For example, if probabilities in training deviate from probabilities predicted in side-by-sides or by inspecting the production system, this deviation could reveal a problem.

For example, in linear, logistic, or Poisson regression, **there are subsets of the data where the average predicted expectation equals the average label (1- moment calibrated, or just calibrated)**. This is true assuming that you have no regularization and that your algorithm has converged, and it is approximately true in general. If you have a feature which is either 1 or 0 for each example, then the set of 3 examples where that feature is 1 is calibrated. Also, if you have a feature that is 1 for every example, then the set of all examples is calibrated.

With simple models, it is easier to deal with feedback loops (see Rule #36 (#rule_36_avoid_feedback_loops_with_positional_features)). Often, we use these probabilistic predictions to make a decision: e.g. rank posts in decreasing expected value (i.e. probability of click/download/etc.). **However, remember when it comes time to choose which model to use, the decision matters more than the likelihood of the data given the model (see Rule #27 (#rule_27_try_to_quantify_observed_undesirable_behavior)).**

Rule #15: Separate Spam Filtering and Quality Ranking in a Policy Layer.

Quality ranking is a fine art, but spam filtering is a war. The signals that you use to determine high quality posts will become obvious to those who use your system, and they will tweak their posts to have these properties. Thus, your quality ranking should focus on ranking content that is posted in good faith. You should not discount the quality ranking learner for ranking spam highly. **Similarly, "racy" content should be handled separately from Quality Ranking.** Spam filtering is a different story. You have to expect that the features that you need to generate will be constantly changing. Often, there will be obvious rules that you put into the system (if a post has more than three spam votes, don't retrieve it, et cetera). Any learned model will have to be updated daily, if not faster. The reputation of the creator of the content will play a great role.

At some level, the output of these two systems will have to be integrated. Keep in mind, filtering spam in search results should probably be more aggressive than filtering spam in email messages. This is true assuming that you have no regularization and that your algorithm has converged. It is approximately true in general. Also, it is a standard practice to remove spam from the training data for the quality classifier.

ML Phase II: Feature Engineering

In the first phase of the lifecycle of a machine learning system, the important issues are to get the training data into the learning system, get any metrics of interest instrumented, and create a serving infrastructure. **After you have a working end to end system with unit and system tests instrumented, Phase II begins.**

In the second phase, there is a lot of low-hanging fruit. There are a variety of obvious features that could be pulled into the system. Thus, the second phase of machine learning involves pulling in as many features as possible and combining them in intuitive ways. During this phase, all of the metrics should still be rising. There will be lots of launches, and it is a great time to pull in lots of engineers that can join up all the data that you need to create a truly awesome learning system.

Rule #16: Plan to launch and iterate.

Don't expect that the model you are working on now will be the last one that you will launch, or even that you will ever stop launching models. Thus consider whether the complexity you are adding with this launch will slow down future launches. Many teams have launched a model per quarter or more for years. There are three basic reasons to launch new models:

- You are coming up with new features.
- You are tuning regularization and combining old features in new ways.
- You are tuning the objective.

Regardless, giving a model a bit of love can be good: looking over the data feeding into the example can help find new signals as well as old, broken ones. So, as you build your model, think about how easy it is to add or remove or recombine features. Think about how easy it is to create a fresh copy of the pipeline and verify its correctness. Think about whether it is possible to have two or three copies running in parallel. Finally, don't worry about whether feature 16 of 35 makes it into this version of the pipeline. You'll get it next quarter.

Rule #17: Start with directly observed and reported features as opposed to learned features.

This might be a controversial point, but it avoids a lot of pitfalls. First of all, let's describe what a learned feature is. A learned feature is a feature generated either by an external system (such as an unsupervised clustering system) or by the learner itself (e.g. via a factored model or deep learning). Both of these can be useful, but they can have a lot of issues, so they should not be in the first model.

If you use an external system to create a feature, remember that the external system has its own objective. The external system's objective may be only weakly correlated with your current objective. If you grab a snapshot of the external system, then it can become out of date. If you update the features from the external system, then the meanings may change. If you use an external system to provide a feature, be aware that this approach requires a great deal of care.

The primary issue with factored models and deep models is that they are nonconvex. Thus, there is no guarantee that an optimal solution can be approximated or found, and the local minima found on each iteration can be different. This variation makes it hard to judge whether the impact of a change to your system is meaningful or random. By creating a model without deep features, you can get an excellent baseline performance. After this baseline is achieved, you can try more esoteric approaches.

Rule #18: Explore with features of content that generalize across contexts.

Often a machine learning system is a small part of a much bigger picture. For example, if you imagine a post that might be used in What's Hot, many people will plus-one, reshare, or comment on a post before it is ever shown in What's Hot. If you provide those statistics to the learner, it can promote new posts that it has no data for in the context it is optimizing. YouTube (#youtube_overview) Watch Next could use number of watches, or co- watches (counts of how many times one video was

watched after another was watched) from YouTube (#youtube_overview) search. You can also use explicit user ratings. Finally, if you have a user action that you are using as a label, seeing that action on the document in a different context can be a great feature. All of these features allow you to bring new content into the context. Note that this is not about personalization: figure out if someone likes the content in this context first, then figure out who likes it more or less.

Rule #19: Use very specific features when you can.

With tons of data, it is simpler to learn millions of simple features than a few complex features. Identifiers of documents being retrieved and canonicalized queries do not provide much generalization, but align your ranking with your labels on head queries. Thus, don't be afraid of groups of features where each feature applies to a very small fraction of your data, but overall coverage is above 90%. You can use regularization to eliminate the features that apply to too few examples.

Rule #20: Combine and modify existing features to create new features in human-understandable ways.

There are a variety of ways to combine and modify features. Machine learning systems such as TensorFlow allow you to pre-process your data through transformations (https://www.tensorflow.org/tutorials/linear#feature-columns-and-transformations). The two most standard approaches are "discretizations" and "crosses".

Discretization consists of taking a continuous feature and creating many discrete features from it. Consider a continuous feature such as age. You can create a feature which is 1 when age is less than 18, another feature which is 1 when age is between 18 and 35, et cetera. Don't overthink the boundaries of these histograms: basic quantiles will give you most of the impact.

Crosses combine two or more feature columns. A feature column, in TensorFlow's terminology, is a set of homogenous features, (e.g. {male, female}, {US, Canada, Mexico}, et cetera). A cross is a new feature column with features in, for example, {male, female} × {US,Canada, Mexico}. This new feature column will contain the feature (male, Canada). If you are using TensorFlow and you tell TensorFlow to create this cross for you, this (male, Canada) feature will be present in examples representing male Canadians. Note that it takes massive amounts of data to learn models with crosses of three, four, or more base feature columns.

Crosses that produce very large feature columns may overfit. For instance, imagine that you are doing some sort of search, and you have a feature column with words in the query, and you have a feature column with words in the document. You can combine these with a cross, but you will end up with a lot of features (see Rule #21 (#rule_21_the_number_of_feature_weights_you_can_learn_in_a_linear_model_is_roughly_proportional_to_the_amount_of_data_you_have)).

When working with text there are two alternatives. The most draconian is a dot product. A dot product in its simplest form simply counts the number of words in common between the query and the document. This feature can then be discretized. Another approach is an intersection: thus, we will have a feature which is present if and only if the word "pony" is in both the document and the query, and another feature which is present if and only if the word "the" is in both the document and the query.

Rule #21: The number of feature weights you can learn in a linear model is roughly proportional to the amount of data you have.

There are fascinating statistical learning theory results concerning the appropriate level of complexity for a model, but this rule is basically all you need to know. I have had conversations in which people were doubtful that anything can be learned from one thousand examples, or that you would ever need more than one million examples, because they get stuck in a certain method of learning. The key is to scale your learning to the size of your data:

1. If you are working on a search ranking system, and there are millions of different words in the documents and the query and you have 1000 labeled examples, then you should use a dot product between document and query features, TF-IDF

(https://wikipedia.org/wiki/Tf%E2%80%93idf), and a half-dozen other highly human-engineered features. 1000 examples, a dozen features.

2. If you have a million examples, then intersect the document and query feature columns, using regularization and possibly feature selection. This will give you millions of features, but with regularization you will have fewer. Ten million examples, maybe a hundred thousand features.
3. If you have billions or hundreds of billions of examples, you can cross the feature columns with document and query tokens, using feature selection and regularization. You will have a billion examples, and 10 million features. Statistical learning theory rarely gives tight bounds, but gives great guidance for a starting point.

In the end, use Rule #28 (#rule_28_be_aware_that_identical_short_term_behavior_does_not_imply_identical_long_term_behavior) to decide what features to use.

Rule #22: Clean up features you are no longer using.

Unused features create technical debt. If you find that you are not using a feature, and that combining it with other features is not working, then drop it out of your infrastructure. You want to keep your infrastructure clean so that the most promising features can be tried as fast as possible. If necessary, someone can always add back your feature.

Keep coverage in mind when considering what features to add or keep. How many examples are covered by the feature? For example, if you have some personalization features, but only 8% of your users have any personalization features, it is not going to be very effective.

At the same time, some features may punch above their weight. For example, if you have a feature which covers only 1% of the data, but 90% of the examples that have the feature are positive, then it will be a great feature to add.

Human Analysis of the System

Before going on to the third phase of machine learning, it is important to focus on something that is not taught in any machine learning class: how to look at an existing model, and improve it. This is more of an art than a science, and yet there are several antipatterns that it helps to avoid.

Rule #23: You are not a typical end user.

This is perhaps the easiest way for a team to get bogged down. While there are a lot of benefits to fishfooding (using a prototype within your team) and dogfooding (using a prototype within your company), employees should look at whether the performance is correct. While a change which is obviously bad should not be used, anything that looks reasonably near production should be tested further, either by paying laypeople to answer questions on a crowdsourcing platform, or through a live experiment on real users.

There are two reasons for this. The first is that you are too close to the code. You may be looking for a particular aspect of the posts, or you are simply too emotionally involved (e.g. confirmation bias). The second is that your time is too valuable. Consider the cost of nine engineers sitting in a one hour meeting, and think of how many contracted human labels that buys on a crowdsourcing platform.

If you really want to have user feedback, **use user experience methodologies**. Create user personas (one description is in Bill Buxton's Sketching User Experiences (https://play.google.com/store/books/details/Bill_Buxton_Sketching_User_Experiences_Getting_the?id=2vfPxocmLh0C)) early in a process and do usability testing (one description is in Steve Krug's Don't Make Me Think (https://play.google.com/store/books/details/Steve_Krug_Don_t_Make_Me_Think_Revisited?id=QlduAgAAQBAJ)) later. User personas involve creating a hypothetical user. For instance, if your team is all male, it might help to design a 35-year-old female user persona (complete with user features), and look at the results it generates rather than 10 results for 25-to-40 year old males.

Bringing in actual people to watch their reaction to your site (locally or remotely) in usability testing can also get you a fresh perspective.

Rule #24: Measure the delta between models.

One of the easiest and sometimes most useful measurements you can make before any users have looked at your new model is to calculate just how different the new results are from production. For instance, if you have a ranking problem, run both models on a sample of queries through the entire system, and look at the size of the symmetric difference of the results (weighted by ranking position). If the difference is very small, then you can tell without running an experiment that there will be little change. If the difference is very large, then you want to make sure that the change is good. Looking over queries where the symmetric difference is high can help you to understand qualitatively what the change was like. Make sure, however, that the system is stable. Make sure that a model when compared with itself has a low (ideally zero) symmetric difference.

Rule #25: When choosing models, utilitarian performance trumps predictive power.

Your model may try to predict click-through rate. However, in the end, the key question is what you do with that prediction. If you are using it to rank documents, then the quality of the final ranking matters more than the prediction itself. If you predict the probability that a document is spam and then have a cutoff on what is blocked, then the precision of what is allowed through matters more. Most of the time, these two things should be in agreement: when they do not agree, it will likely be on a small gain. Thus, if there is some change that improves log loss but degrades the performance of the system, look for another feature. When this starts happening more often, it is time to revisit the objective of your model.

Rule #26: Look for patterns in the measured errors, and create new features.

Suppose that you see a training example that the model got "wrong". In a classification task, this error could be a false positive or a false negative. In a ranking task, the error could be a pair where a positive was ranked lower than a negative. The most important point is that this is an example that the machine learning system knows it got wrong and would like to fix if given the opportunity. If you give the model a feature that allows it to fix the error, the model will try to use it.

On the other hand, if you try to create a feature based upon examples the system doesn't see as mistakes, the feature will be ignored. For instance, suppose that in Play Apps Search, someone searches for "free games". Suppose one of the top results is a less relevant gag app. So you create a feature for "gag apps". However, if you are maximizing number of installs, and people install a gag app when they search for free games, the "gag apps" feature won't have the effect you want.

Once you have examples that the model got wrong, look for trends that are outside your current feature set. For instance, if the system seems to be demoting longer posts, then add post length. Don't be too specific about the features you add. If you are going to add post length, don't try to guess what long means, just add a dozen features and the let model figure out what to do with them (see Rule #21 (#rule_21_the_number_of_feature_weights_you_can_learn_in_a_linear_model_is_roughly_proportional_to_the_amount_of_data_you_have)). That is the easiest way to get what you want.

Rule #27: Try to quantify observed undesirable behavior.

Some members of your team will start to be frustrated with properties of the system they don't like which aren't captured by the existing loss function. At this point, they should do whatever it takes to turn their gripes into solid numbers. For example, if they think that too many "gag apps" are being shown in Play Search, they could have human raters identify gag apps. (You can feasibly use humanlabelled data in this case because a relatively small fraction of the queries account for a large fraction of the traffic.) If your issues are measurable, then you can start using them as features, objectives, or metrics. The general rule is "**measure first, optimize second**".

Rule #28: Be aware that identical short-term behavior does not imply identical long-term behavior.

Imagine that you have a new system that looks at every doc_id and exact_query, and then calculates the probability of click for every doc for every query. You find that its behavior is nearly identical to your current system in both side by sides and A/B testing, so given its simplicity, you launch it. However, you notice that no new apps are being shown. Why? Well, since your system only shows a doc based on its own history with that query, there is no way to learn that a new doc should be shown.

The only way to understand how such a system would work long-term is to have it train only on data acquired when the model was live. This is very difficult.

Training-Serving Skew

Training-serving skew is a difference between performance during training and performance during serving. This skew can be caused by:

- A discrepancy between how you handle data in the training and serving pipelines.
- A change in the data between when you train and when you serve.
- A feedback loop between your model and your algorithm.

We have observed production machine learning systems at Google with training- serving skew that negatively impacts performance. The best solution is to explicitly monitor it so that system and data changes don't introduce skew unnoticed.

Rule #29: The best way to make sure that you train like you serve is to save the set of features used at serving time, and then pipe those features to a log to use them at training time.

Even if you can't do this for every example, do it for a small fraction, such that you can verify the consistency between serving and training (see Rule #37 (#rule_37_measure_training_serving_skew)). Teams that have made this measurement at Google were sometimes surprised by the results. YouTube (#youtube_overview) home page switched to logging features at serving time with significant quality improvements and a reduction in code complexity, and many teams are switching their infrastructure as we speak.

Rule #30: Importance-weight sampled data, don't arbitrarily drop it!

When you have too much data, there is a temptation to take files 1-12, and ignore files 13-99. This is a mistake. Although data that was never shown to the user can be dropped, importance weighting is best for the rest. Importance weighting means that if you decide that you are going to sample example X with a 30% probability, then give it a weight of 10/3. **With importance weighting, all of the calibration properties discussed in Rule #14** (#rule_14_starting_with_an_interpretable_model_makes_debugging_easier) **still hold.**

Rule #31: Beware that if you join data from a table at training and serving time, the data in the table may change.

Say you join doc ids with a table containing features for those docs (such as number of comments or clicks). Between training and serving time, features in the table may be changed. Your model's prediction for the same document may then differ between training and serving. The easiest way to avoid this sort of problem is to log features at serving time (see Rule #32 (#rule_32_re_use_code_between_your_training_pipeline_and_your_serving_pipeline_whenever_possible)). If the table is changing only slowly, you can also snapshot the table hourly or daily to get reasonably close data. Note that this still doesn't completely resolve the issue.

Rule #32: Re-use code between your training pipeline and your serving pipeline whenever possible.

Batch processing is different than online processing. In online processing, you must handle each request as it arrives (e.g. you must do a separate lookup for each query), whereas in batch processing, you can combine tasks (e.g. making a join). At

serving time, you are doing online processing, whereas training is a batch processing task. However, there are some things that you can do to re-use code. For example, you can create an object that is particular to your system where the result of any queries or joins can be stored in a very human readable way, and errors can be tested easily. Then, once you have gathered all the information, during serving or training, you run a common method to bridge between the human-readable object that is specific to your system, and whatever format the machine learning system expects. **This eliminates a source of training-serving skew**. As a corollary, try not to use two different programming languages between training and serving. That decision will make it nearly impossible for you to share code.

Rule #33: If you produce a model based on the data until January 5th, test the model on the data from January 6th and after.

In general, measure performance of a model on the data gathered after the data you trained the model on, as this better reflects what your system will do in production. If you produce a model based on the data until January 5th, test the model on the data from January 6th. You will expect that the performance will not be as good on the new data, but it shouldn't be radically worse. Since there might be daily effects, you might not predict the average click rate or conversion rate, but the area under the curve, which represents the likelihood of giving the positive example a score higher than a negative example, should be reasonably close.

Rule #34: In binary classification for filtering (such as spam detection or determining interesting emails), make small short-term sacrifices in performance for very clean data.

In a filtering task, examples which are marked as negative are not shown to the user. Suppose you have a filter that blocks 75% of the negative examples at serving. You might be tempted to draw additional training data from the instances shown to users. For example, if a user marks an email as spam that your filter let through, you might want to learn from that.

But this approach introduces sampling bias. You can gather cleaner data if instead during serving you label 1% of all traffic as "held out", and send all held out examples to the user. Now your filter is blocking at least 74% of the negative examples. These held out examples can become your training data.

Note that if your filter is blocking 95% of the negative examples or more, this approach becomes less viable. Even so, if you wish to measure serving performance, you can make an even tinier sample (say 0.1% or 0.001%). Ten thousand examples is enough to estimate performance quite accurately.

Rule #35: Beware of the inherent skew in ranking problems.

When you switch your ranking algorithm radically enough that different results show up, you have effectively changed the data that your algorithm is going to see in the future. This kind of skew will show up, and you should design your model around it. There are multiple different approaches. These approaches are all ways to favor data that your model has already seen.

1. Have higher regularization on features that cover more queries as opposed to those features that are on for only one query. This way, the model will favor features that are specific to one or a few queries over features that generalize to all queries. This approach can help prevent very popular results from leaking into irrelevant queries. Note that this is opposite the more conventional advice of having more regularization on feature columns with more unique values.
2. Only allow features to have positive weights. Thus, any good feature will be better than a feature that is "unknown".
3. Don't have document-only features. This is an extreme version of #1. For example, even if a given app is a popular download regardless of what the query was, you don't want to show it everywhere. Not having document-only features keeps that simple. The reason you don't want to show a specific popular app everywhere has to do with the importance of making all the desired apps reachable. For instance, if someone searches for "bird watching app", they might download "angry birds", but that certainly wasn't their intent. Showing such an app might improve download rate, but leave the user's needs ultimately unsatisfied.

Rule #36: Avoid feedback loops with positional features.

The position of content dramatically affects how likely the user is to interact with it. If you put an app in the first position it will be clicked more often, and you will be convinced it is more likely to be clicked. One way to deal with this is to add positional features, i.e. features about the position of the content in the page. You train your model with positional features, and it learns to weight, for example, the feature "1stposition" heavily. Your model thus gives less weight to other factors for examples with "1stposition=true". Then at serving you don't give any instances the positional feature, or you give them all the same default feature, because you are scoring candidates before you have decided the order in which to display them.

Note that it is important to keep any positional features somewhat separate from the rest of the model because of this asymmetry between training and testing. Having the model be the sum of a function of the positional features and a function of the rest of the features is ideal. For example, don't cross the positional features with any document feature.

Rule #37: Measure Training/Serving Skew.

There are several things that can cause skew in the most general sense. Moreover, you can divide it into several parts:

- The difference between the performance on the training data and the holdout data. In general, this will always exist, and it is not always bad.
- The difference between the performance on the holdout data and the "nextday" data. Again, this will always exist. You should tune your regularization to maximize the next-day performance. However, large drops in performance between holdout and next-day data may indicate that some features are time-sensitive and possibly degrading model performance.
- The difference between the performance on the "next-day" data and the live data. If you apply a model to an example in the training data and the same example at serving, it should give you exactly the same result (see Rule #5 (https://developers.google.com/machine-learning/guides/rules-of-ml/rule_5_test_the_infrastructure_independently_from_the_machine_learning)). Thus, a discrepancy here probably indicates an engineering error.

ML Phase III: Slowed Growth, Optimization Refinement, and Complex Models

There will be certain indications that the second phase is reaching a close. First of all, your monthly gains will start to diminish. You will start to have tradeoffs between metrics: you will see some rise and others fall in some experiments. This is where it gets interesting. Since the gains are harder to achieve, the machine learning has to get more sophisticated. A caveat: this section has more blue-sky rules than earlier sections. We have seen many teams go through the happy times of Phase I and Phase II machine learning. Once Phase III has been reached, teams have to find their own path.

Rule #38: Don't waste time on new features if unaligned objectives have become the issue.

As your measurements plateau, your team will start to look at issues that are outside the scope of the objectives of your current machine learning system. As stated before, if the product goals are not covered by the existing algorithmic objective, you need to change either your objective or your product goals. For instance, you may optimize clicks, plus-ones, or downloads, but make launch decisions based in part on human raters.

Rule #39: Launch decisions are a proxy for long-term product goals.

Alice has an idea about reducing the logistic loss of predicting installs. She adds a feature. The logistic loss drops. When she does a live experiment, she sees the install rate increase. However, when she goes to a launch review meeting, someone points out that the number of daily active users drops by 5%. The team decides not to launch the model. Alice is disappointed, but now realizes that launch decisions depend on multiple criteria, only some of which can be directly optimized using ML.

The truth is that the real world is not dungeons and dragons: there are no "hit points" identifying the health of your product. The team has to use the statistics it gathers to try to effectively predict how good the system will be in the future. They need to care about engagement, 1 day active users (DAU), 30 DAU, revenue, and advertiser's return on investment. These metrics that are measureable in A/B tests in themselves are only a proxy for more longterm goals: satisfying users, increasing users, satisfying partners, and profit, which even then you could consider proxies for having a useful, high quality product and a thriving company five years from now.

The only easy launch decisions are when all metrics get better (or at least do not get worse). If the team has a choice between a sophisticated machine learning algorithm, and a simple heuristic, if the simple heuristic does a better job on all these metrics, it should choose the heuristic. Moreover, there is no explicit ranking of all possible metric values. Specifically, consider the following two scenarios:

Experiment	Daily Active Users	Revenue/Day
A	1 million	$4 million
B	2 million	$2 million

If the current system is A, then the team would be unlikely to switch to B. If the current system is B, then the team would be unlikely to switch to A. This seems in conflict with rational behavior; however, predictions of changing metrics may or may not pan out, and thus there is a large risk involved with either change. Each metric covers some risk with which the team is concerned.

Moreover, no metric covers the team's ultimate concern, "where is my product going to be five years from now"?

Individuals, on the other hand, tend to favor one objective that they can directly optimize. Most machine learning tools favor such an environment. An engineer banging out new features can get a steady stream of launches in such an environment. There is a type of machine learning, multi-objective learning, which starts to address this problem. For instance, one can formulate a constraint satisfaction problem that has lower bounds on each metric, and optimizes some linear combination of metrics. However, even then, not all metrics are easily framed as machine learning objectives: if a document is clicked on or an app is installed, it is because that the content was shown. But it is far harder to figure out why a user visits your site. How to predict the future success of a site as a whole is AI-complete (https://wikipedia.org/wiki/AI-complete): as hard as computer vision or natural language processing.

Rule #40: Keep ensembles simple.

Unified models that take in raw features and directly rank content are the easiest models to debug and understand. However, an ensemble of models (a "model" which combines the scores of other models) can work better. **To keep things simple, each model should either be an ensemble only taking the input of other models, or a base model taking many features, but not both.** If you have models on top of other models that are trained separately, then combining them can result in bad behavior.

Use a simple model for ensembling that takes only the output of your "base" models as inputs. You also want to enforce properties on these ensemble models. For example, an increase in the score produced by a base model should not decrease the score of the ensemble. Also, it is best if the incoming models are semantically interpretable (for example, calibrated) so that changes of the underlying models do not confuse the ensemble model. Also, **enforce that an increase in the predicted probability of an underlying classifier does not decrease the predicted probability of the ensemble**.

Rule #41: When performance plateaus, look for qualitatively new sources of information to add rather than refining existing signals.

You've added some demographic information about the user. You've added some information about the words in the document. You have gone through template exploration, and tuned the regularization. You haven't seen a launch with more

than a 1% improvement in your key metrics in a few quarters. Now what?

It is time to start building the infrastructure for radically different features, such as the history of documents that this user has accessed in the last day, week, or year, or data from a different property. Use wikidata (https://wikipedia.org/wiki/Wikidata) entities or something internal to your company (such as Google's knowledge graph (https://wikipedia.org/wiki/Knowledge_Graph)). Use deep learning. Start to adjust your expectations on how much return you expect on investment, and expand your efforts accordingly. As in any engineering project, you have to weigh the benefit of adding new features against the cost of increased complexity.

Rule #42: Don't expect diversity, personalization, or relevance to be as correlated with popularity as you think they are.

Diversity in a set of content can mean many things, with the diversity of the source of the content being one of the most common. Personalization implies each user gets their own results. Relevance implies that the results for a particular query are more appropriate for that query than any other. Thus all three of these properties are defined as being different from the ordinary.

The problem is that the ordinary tends to be hard to beat.

Note that if your system is measuring clicks, time spent, watches, +1s, reshares, et cetera, you are measuring the **popularity** of the content. Teams sometimes try to learn a personal model with diversity. To personalize, they add features that would allow the system to personalize (some features representing the user's interest) or diversify (features indicating if this document has any features in common with other documents returned, such as author or content), and find that those features get less weight (or sometimes a different sign) than they expect.

This doesn't mean that diversity, personalization, or relevance aren't valuable. As pointed out in the previous rule, you can do postprocessing to increase diversity or relevance. If you see longer term objectives increase, then you can declare that diversity/relevance is valuable, aside from popularity. You can then either continue to use your postprocessing, or directly modify the objective based upon diversity or relevance.

Rule #43: Your friends tend to be the same across different products. Your interests tend not to be.

Teams at Google have gotten a lot of traction from taking a model predicting the closeness of a connection in one product, and having it work well on another. Your friends are who they are. On the other hand, I have watched several teams struggle with personalization features across product divides. Yes, it seems like it should work. For now, it doesn't seem like it does. What has sometimes worked is using raw data from one property to predict behavior on another. Also, keep in mind that even knowing that a user has a history on another property can help. For instance, the presence of user activity on two products may be indicative in and of itself.

Related Work

There are many documents on machine learning at Google as well as externally.

- Machine Learning Crash Course (https://developers.google.com/machine-learning/crash-course/): an introduction to applied machine learning.
- Machine Learning: A Probabilistic Approach (https://www.cs.ubc.ca/%7Emurphyk/MLbook/) by Kevin Murphy for an understanding of the field of machine learning.
- Practical Advice for the Analysis of Large, Complex Data Sets (http://www.unofficialgoogledatascience.com/2016/10/practical-advice-for-analysis-of-large.html): a data science approach to thinking about data sets.
- Deep Learning (http://www.iro.umontreal.ca/%7Ebengioy/dlbook/) by Ian Goodfellow et al for learning nonlinear models.

- Google paper on technical debt (http://research.google.com/pubs/pub43146.html), which has a lot of general advice.
- Tensorflow Documentation (https://www.tensorflow.org/).

Acknowledgements

Thanks to David Westbrook, Peter Brandt, Samuel Ieong, Chenyu Zhao, Li Wei, Michalis Potamias, Evan Rosen, Barry Rosenberg, Christine Robson, James Pine, Tal Shaked, Tushar Chandra, Mustafa Ispir, Jeremiah Harmsen, Konstantinos Katsiapis, Glen Anderson, Dan Duckworth, Shishir Birmiwal, Gal Elidan, Su Lin Wu, Jaihui Liu, Fernando Pereira, and Hrishikesh Aradhye for many corrections, suggestions, and helpful examples for this document. Also, thanks to Kristen Lefevre, Suddha Basu, and Chris Berg who helped with an earlier version. Any errors, omissions, or (gasp!) unpopular opinions are my own.

Appendix

There are a variety of references to Google products in this document. To provide more context, I give a short description of the most common examples below.

YouTube Overview

YouTube is a streaming video service. Both YouTube Watch Next and YouTube Home Page teams use ML models to rank video recommendations. Watch Next recommends videos to watch after the currently playing one, while Home Page recommends videos to users browsing the home page.

Google Play Overview

Google Play has many models solving a variety of problems. Play Search, Play Home Page Personalized Recommendations, and 'Users Also Installed' apps all use machine learning.

Google Plus Overview

Google Plus uses machine learning in a variety of situations: ranking posts in the "stream" of posts being seen by the user, ranking "What's Hot" posts (posts that are very popular now), ranking people you know, et cetera.

Last updated October 24, 2018.