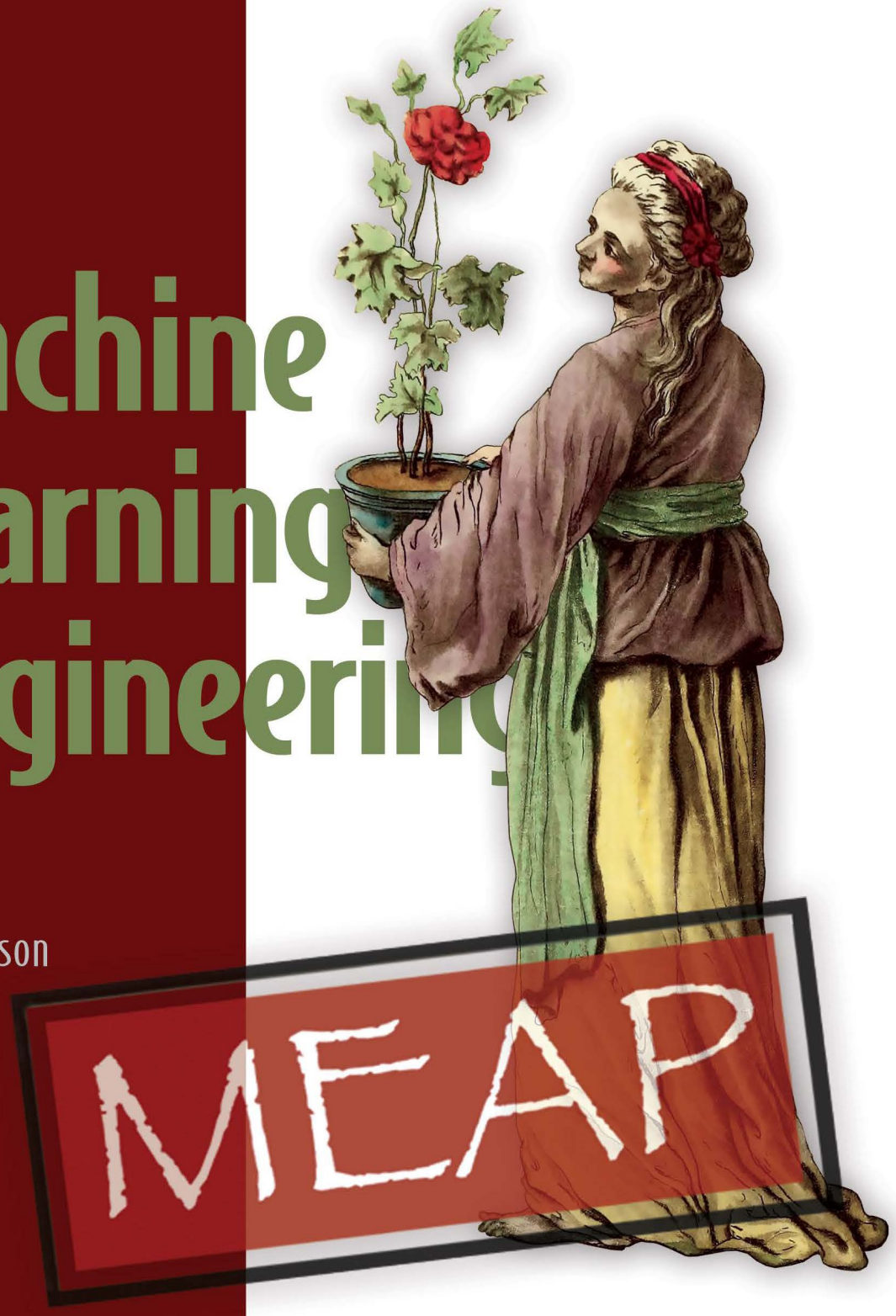
Machine Learning Engineering

Ben T. Wilson



MANNING

MEAP Edition
Manning Early Access Program
Machine Learning Engineering

Version 2

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for taking the time to take a look at *Machine Learning Engineering*. I sincerely hope that all of the lessons that I've learned the hard way throughout my career can be of a benefit to you!
Like any ML project that I've ever worked on, I've always made the solution better by listening very carefully to what other people have to say and learning from their thoughts and opinions. This MEAP is no different. I'm excited to read what you have to say about what you're seeing in this book and working to make it better through your feedback.

To get the most out of most of the lessons and topics in this book it will help to have at least a moderate knowledge of the *applications of* ML (algorithmic theory will be discussed in places, but there are no requirements to understand the nuances of these libraries) and have the ability to build an ML solution in either Python or Scala.

Throughout the book we'll be covering the enormous elephant in the room of nearly all companies who are working towards getting benefits out of Data Science and ML work: why so many projects fail. Whether it be failure-to-launch scenarios, overly complex solutions to problems that could be solved in simpler manners, fragile code, expensive solutions, or poorly trained implementations that provide really poor results, the end result in many projects across so many industries are failing to live up to the promises of what is expected in predictive modeling. As we go through scenarios and code implementation demonstrations, we'll be covering not only the 'how' of building resilient ML solutions, but the 'why' as well.

The goal of this book is not to show how to implement certain solutions through the application of specific algorithms, explain the theory behind how the algorithms work, nor is it focused on one particular sub-genre of ML. It instead is a focus on the broad topic of successful ML work, how to apply Agile fundamentals to ML, and demonstrably production-ready code bases that will, if adhered to, ensure that you have a maintainable and robust solution for projects.

With the dozens of ML theory and model-centric books that I've read and collected throughout my career, I really wish that I had a copy of this book when I was getting started in this field many years ago. The crippling failures that I've endured as I've learned these paradigms have been both painful and formative, and to be able to provide them for the next generation of ML practitioners out there is truly a gift for me.

Thank you once again for taking the time to look at this work, provide feedback in liveBook discussion forum (pointing out things that I might have missed or that you'd like me to cover

will be most welcome!), and joining in on (hopefully pleasant) conversations about what you don't and do like about the work.

Kindest Regards,
Ben Wilson

brief contents

1

What is a Machine Learning Engineer?

This chapter covers

- **Defining Machine Learning Engineering and why it is important to increase the chances of successful ML project work**
- **Explaining why ML Engineering and the processes, tools, and paradigms surrounding it can minimize the chances of ML project abandonment**
- **Discussing the six primary tenets of ML Engineering and how failing to follow them causes project failure**

Machine learning (ML) is exciting. To the layperson, it brings with it the promise of seemingly magical abilities of soothsaying; uncovering mysterious and miraculous answers to difficult problems. ML makes money for companies, it autonomously tackles overwhelmingly large tasks, and relieves people from the burden of monotonous work involved in analyzing data to draw conclusions from. To state the obvious, though, it's challenging. From thousands of algorithms, a diverse skill set ranging from Data Engineering (DE) skills to advanced statistical analysis and visualization, the work required of a professional practitioner of ML is truly intimidating and filled with complexity.

ML Engineering is the concept of applying a *system* around this staggering level of complexity. It is a set of standards, tools, processes, and methodology that aims to minimize the chances of abandoned, misguided, or irrelevant work being done in an effort to solve a business problem or need. It, in essence, is the roadmap to creating ML-based systems that can not only be deployed to production, but can be maintained and updated for years in the future, allowing businesses to reap the rewards in efficiency, profitability, and accuracy that ML in general has proven to provide (when done correctly).

This book is, at its essence, a roadmap to guide you through this system, as shown in figure 1.1 below. It gives a proven set of processes about the planning phase of project work, navigating the difficult and confusing translation of business needs into the language of ML work. From that, it covers a standard methodology of experimentation work, focusing on the tools and coding standards for creating an MVP that will be comprehensive and maintainable. Finally, it will cover the various tools, techniques, and nuances involved in crafting production-grade maintainable code that is both extensible and easy to troubleshoot.

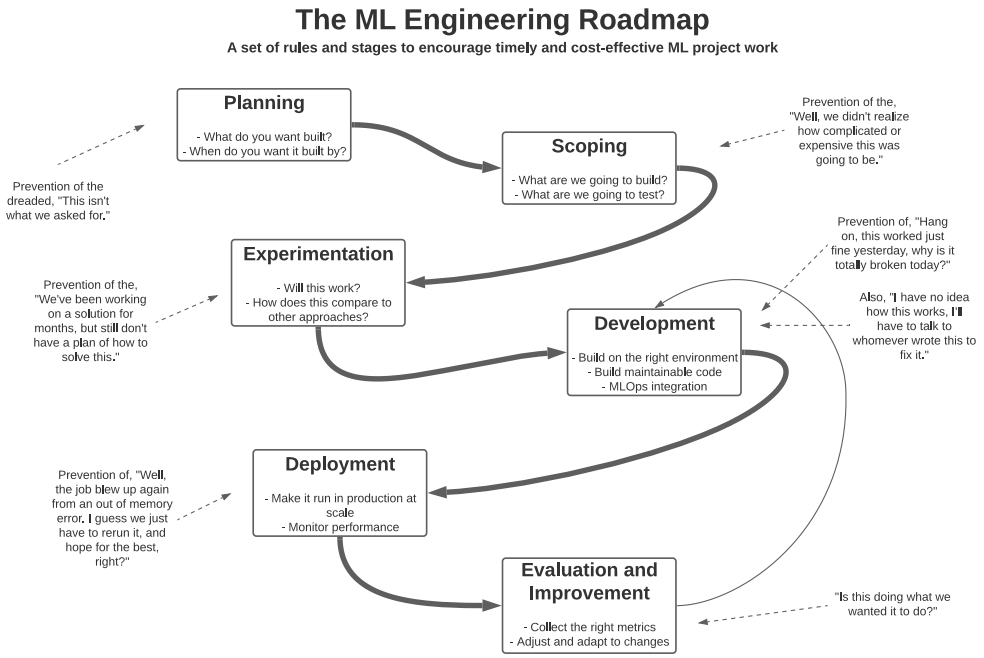


Figure 1.1 The ML Engineering Roadmap, showing the proven stages of work involved in creating successful ML solutions. While some projects may require additional steps (particularly if working with additional Engineering teams), these are the fundamental steps that should be involved in any ML-backed project.

ML Engineering is not exclusively about the path shown in figure 1.1. It is also the methodology within each of these stages that can make or break a project. It is the way in which a Data Science team talks to the business about the problem, the manner in which research is done, the details of experimentation, the way that code is written, and the multitude of tools and technology that are employed while traveling along the roadmap path that can greatly reduce the worst outcome that a project might have – abandonment.

The end goal of ML work is, after all, about solving a problem. By embracing the concepts of ML Engineering and following the road of effective project work, the end goal of getting a useful modeling solution can be shorter, far cheaper, and have a much higher probability of succeeding than if you just 'wing it' and hope for the best.

1.1 Why ML Engineering?

To put it most simply, ML is hard. It's even harder to do correctly in the sense of serving relevant predictions, at scale, with reliable frequency. With so many specialties existing in the field (NLP, forecasting, Deep Learning, traditional linear and tree-based modeling, et al), an enormous focus on active research, and so many algorithms that have been built to solve specific problems, it's remarkably hard to learn even slightly more than an insignificant fraction of all there is to learn. Coupling that complexity with the fact that one can develop a model on everything from a RapsberryPi to an enormous NVIDIA GPU cluster, the very platform complexities that are out there is an entirely new set of information that no one person could have enough time in their life to learn.

There are also additional realms of competency that a Data Scientist is expected to be familiar with. From mid-level Data Engineering skills (you have to get your data for your Data Science from somewhere, right?), software development skills, project management skills, visualization skills, and presentation skills, the list grows ever longer and the volumes of experience that need to be gained become rather daunting. It's not much of a surprise, considering all of this, as to why 'just figuring it out' in reference to all of the required skills to create production-grade ML solutions is untenable.

The aim of ML Engineering is not to iterate through the lists of skills just mentioned and require that a DS master each of them. Instead, it's a collection of certain aspects of those skills, carefully crafted to be relevant to Data Scientists, all with the goal of increasing the chances of *getting an ML project into production* and to make sure that it's not a solution that needs constant maintenance and intervention to keep running.

An ML Engineer, after all, doesn't need to be able to create applications and software frameworks for generic algorithmic use cases. They're also not likely to be writing their own large-scale streaming ingestion ETL pipelines. They similarly don't need to be able to create detailed and animated front-end visualizations in JavaScript.

An ML Engineer needs to know ***just enough*** software development skills to be able to write modular code and to implement unit tests. The don't need to know about the intricacies of non-blocking asynchronous messaging brokering. They need just enough Data Engineering skills to build (and schedule the ETL for) feature datasets for their models, but not how to construct a PB-scale streaming ingestion framework. They need just enough visualization skills to create plots and charts that communicate clearly what their research and models are doing, but not how to develop dynamic web apps that have complex UX components. They also need just enough project management experience to know how to properly define, scope, and control a project to solve a problem, but need not go through a PMP certification.

If you've come here hoping for discussions on the finer points of Data Science, algorithms, or to get implementation ideas for a specific use case, this book isn't going to focus on any of that in incredible detail. Those books have already been (and continue to be) written in great detail.

We're here instead to talk about the giant elephant in the room when it comes to ML. We're going to be talking about why, with so many companies going all-in on ML, hiring massive teams of highly compensated Data Scientists, devoting massive amounts of financial and temporal resources to projects, these projects end up failing at an incredibly high rate. We'll be covering the 6 major parts of project failure, as showing in figure 1.1 below,

discussing process around how to identify the reasons why these each cause so many projects to fail, be abandoned, or take far longer than they should to reach production. In each section throughout the first part of this book we will be discussing the solutions to each of these common failures and covering the processes that can make the chances of these derailing your projects very low.

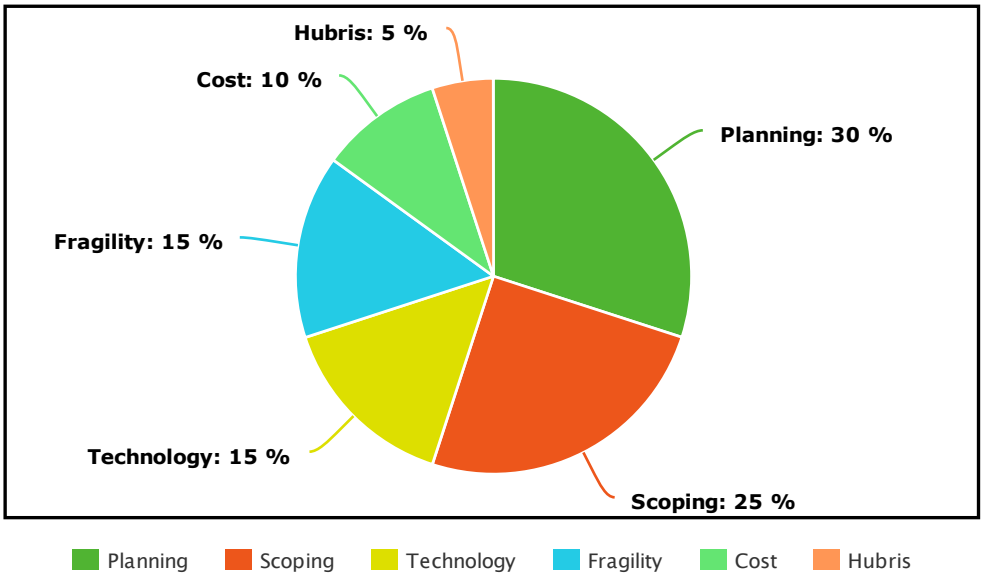


Figure 1.2 Primary reasons for ML project failures

Figure 1.2 shows some rough estimates of why projects typically fail (and the rates of these failures in any given industry, from my experience, are truly staggering). Generally, this is due to a DS team that is either inexperienced with solving a large-scale production-grade model to solve a particular need or has simply failed to understand what the desired outcome from the business is. By focusing on each of these areas in a conscientious and deliberate manner, each of these risks can be entirely mitigated.

At the end of the day, most projects fail. Figure 1.3 below shows the key areas of how these failures occur, illustrating the small percentage of projects that make it through successfully. By focusing on building up the knowledge, skills, and utilization of processes and tooling, each of these 6 pitfalls can be avoided entirely.

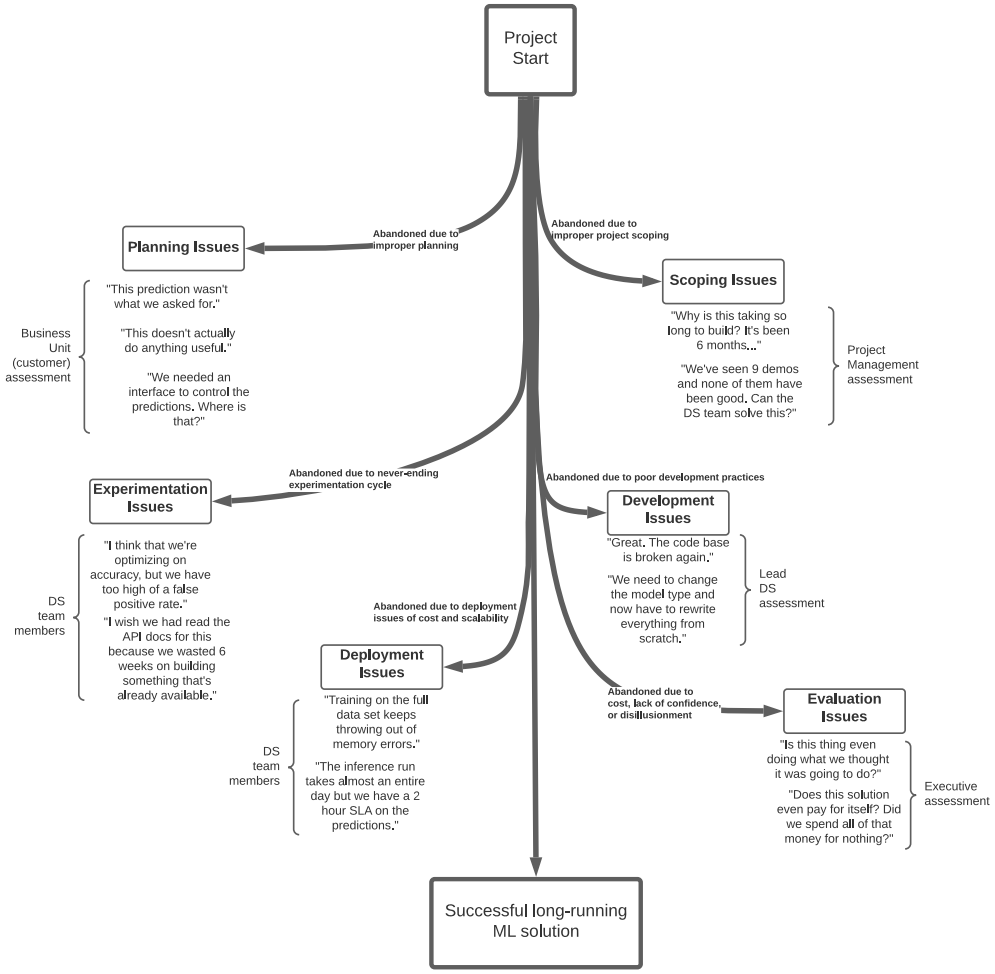


Figure 1.3 The branching paths of failure for the vast majority of ML projects. Nearly all ML solutions that don't adhere to these 6 core areas of focus will be abandoned either before production, or shortly after running in production.

The framework that is ML Engineering is exactly dedicated to address each of these primary failure modes shown above in figure 1.3. Eliminating these chances of failure is at the heart of this methodology. It is done by providing the processes to make better decisions, ease communication with internal customers, eliminate rework during the experimentation and development phases, create code bases that can be easily maintained, and to bring a best-practices focused approach to any project work that is heavily influenced by DS work. Just as software engineers decades ago refined their processes from large-scale waterfall implementations to a more flexible and productive agile process, ML Engineering

seeks to define a new set of practices and tools that will optimize the wholly unique realm of software development for Data Scientists.

1.2 The core components of ML Engineering

Now that we have a general idea of 'what' ML Engineering is, we can focus in a bit to the key elements that make up those incredibly broad categories from section 1.1. Each of these topics will be a focus of entire chapter-length in-depth discussions later in this book, but for now we're going to look at them in a holistic sense by way of potentially painfully familiar scenarios to elucidate why they're so important.

1.2.1 Planning

By far the largest cause of project failures, failing to plan out a project thoroughly is one of the most demoralizing ways for a project to be cancelled. Imagine for a moment that you're the first-hired DS for a company. On your first week, an executive from marketing approaches you, explaining (in their terms) a serious business issue that they are having. They need to figure out an efficient means of communicating to customers through email to let them know of upcoming sales that they might be interested in. With very little additional detail provided to you, the executive merely says, "I want to see the click and open rates go up."

If this was the only information supplied and repeated queries to members of the marketing team simply state the same end-goal of 'increasing the clicking and opening rate', there seems to be a limitless number of avenues to pursue. Left to your own devices, do you:

- Focus on content recommendation and craft custom emails for each user?
- Provide predictions with an NLP-backed system that will craft relevant subject lines for each user?
- Attempt to predict a list of products most relevant to the customer base to put on sale each day?

With so many options of varying complexity and approaches, with very little guidance, the possibility of creating a solution that is aligned with the expectations of the executive is highly unlikely.

If a proper planning discussion were had that goes into the correct amount of detail, avoiding the complexity of the ML side of things, the true expectation might be revealed, letting you know that the only thing that they are expecting is a prediction for each user for when they would be most likely open to reading emails. They simply want to know when someone is most likely to not be at work, commuting, or sleeping so that they can send batches of emails throughout the day to different cohorts of customers.

The sad reality is that many ML projects start off in this way. There is frequently very little communication with regards to project initiation and the general expectation is that 'the DS team will figure it out'. However, without the proper guidance on *what* needs to be built, *how* it needs to function, and *what* the end goal of the predictions is, the project is almost doomed to failure.

After all, what would have happened if an entire content recommendation system were built for that use case, with months of development and effort wasted when a very simple analytics query based on IP geolocation was what was really needed? The project would not only be cancelled, but there would likely be many questions from on-high as to why this system was built and why the cost of development was so high.

If we were to look at a very simplified planning discussion, at an initial phase of discussion, as shown in figure 1.4 below, we can see how just a few careful questions and clear answers can give the one thing that every Data Scientist should be looking for in this situation (of being the first DS at a company working on the first problem): a quick win.

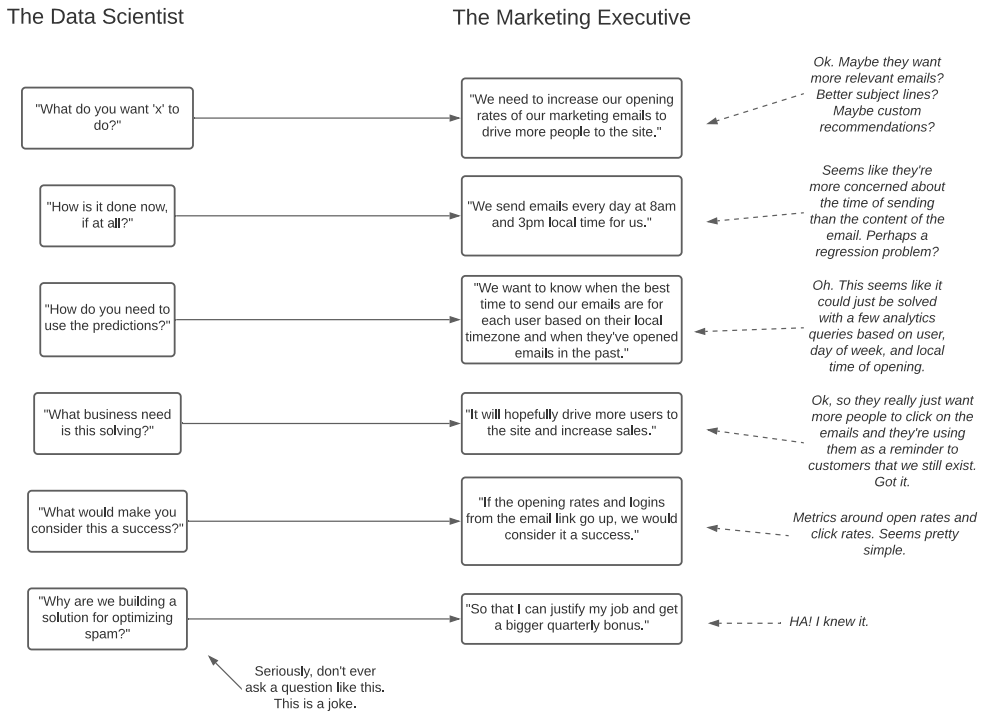


Figure 1.4 A simplified planning discussion to get to the root of what an internal customer (in this case, the marketing executive who wants high open rates on their emails) actually wants for a solution. In this discussion, in only a few quick questions, the actual problem is defined, an appropriate solution is discovered, and the complexity of the project can be accurately scoped without running the risk of building something that is not at all what the marketing team needs.

As figure 1.4 showed at the right (the DS internal monologue), the problem at hand here is not at all in the list of original assumptions that were made. There is no talk of content of the emails, relevancy to the subject line or the items in the email. It's a simple analytical query to figure out which time zone customers are in and to analyze historic opening in local

times for each customer. By taking a few minutes to plan and understand the use case fully, weeks (if not months) of wasted effort, time, and money were saved.

We will be covering the processes of planning, having project expectation discussions with internal business customers, and general communications about ML work with a non-technical audience at length and in much greater depth throughout Chapter 2.

1.2.2 Scoping & Research

The focus of scoping and research needs to answer the two biggest questions that internal customers (the business) have about the project.

FIRST QUESTION Is this going to solve my problem?

SECOND QUESTION How long is this going to take?

Let's take a look at another potentially quite familiar scenario to discuss polar opposite ways that this stage of ML project development can go awry. For this example, there are two separate DS teams at a company, each being pitted against one another for developing a solution to an escalating incidence of fraud being conducted with the company's billing system. Their process and results of their work on researching and scoping is detailed in figure 1.5 below.

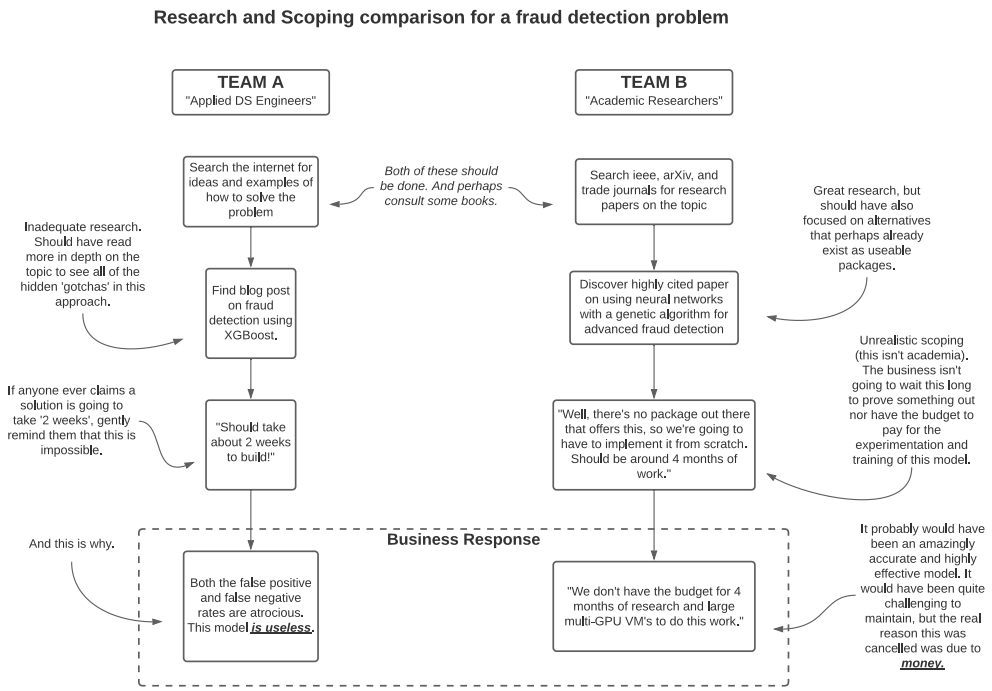


Figure 1.5 The two polar opposites in scoping and research. Team A shows a level of shortcut taking in the research and scoping of the project to the point that the project will be cancelled or reworked due to lack of understanding of the complexity of the subject. Team B shows a bit too much research with the goal of creating a novel algorithm to solve the problem. True best practice scoping and research resides in the middle, where the appropriate amount of knowledge is gained, but with the understanding that the implemented solution needs to be simple enough to solve the problem and cheap enough to be palatable to a business.

For team 'A', comprised of mostly junior Data Scientists, all of whom entered the workforce without an extensive period in academia. Their actions, upon getting the details of the project and what is expected of them, is to immediately go to blog posts. They search the internet for 'detecting payment fraud' and 'fraud algorithms', finding hundreds of results from consultancy companies, a few extremely high-level blog posts from similar junior Data Scientists who have likely never put a model into production, and some open source data examples with very rudimentary examples.

Team 'B', on the other hand, is filled with a group of PhD researchers. Their first actions are to dig into published papers on the topic of fraud modeling. Spending days reading through journals and papers, they are now armed with a large collection of theory encompassing some of the most cutting-edge research being done on detecting fraudulent activity.

If we were to ask either of these teams what the level of effort is to produce a solution, we would get wildly divergent answers. Team 'A' would likely state that it would take about 2

weeks to build their XGBoost binary classification model (they already have the code, after all, from the blog post that they found).

Team 'B' would tell a vastly different tale. They would estimate that it would take several months to implement, train, and evaluate the novel deep learning structure that that they found in a highly regarded white paper whose proven accuracy for the research was significantly better than any perforce implemented algorithm for this use case.

The problem here with scoping and research is that these two polar opposites would both have their projects fail for two completely different reasons. Team 'A' would have a project failure due to the fact that the solution to the problem is significantly more complex than the example shown in the blog post (the class imbalance issue alone is too challenging of a topic to effectively document in the short space of a blog post)

Project scoping for ML is incredibly challenging. Even for the most seasoned of ML veterans, making a conjecture about how long a project will take, which approach is going to be most successful, and the amount of resources that will need to be involved is a futile and frustrating exercise. The risk associated with making erroneous claims is fairly high, but there are means of structuring proper scoping and solution research that can help minimize the chances of being wildly off on estimation.

Most companies have a mix of the types of people in the hyperbolic scenario above. There are academics whose sole goal is to further the advancement of knowledge and research into algorithms, paving the way for future discoveries from within industry. There are also Applications of ML Engineers who just want to use ML as a tool to solve a business problem. It's very important to embrace and balance both aspects of these philosophies toward ML work, strike a compromise during the research and scoping phase of a project, and know that the middle ground here is the best path to trod upon to ensure that a project actually makes it to production.

1.2.3 Experimentation

In the experimentation phase, the largest causes of project failure is either due to the experimentation taking too long (testing too many things or spending too long fine-tuning an approach) or in an under-developed prototype that is so abysmally bad that the business decides to move on to something else.

Let's use a similar example from section 1.2.2 to illustrate how these two approaches might play out at a company that is looking to build an image classifier for detecting products on retail store shelves. The experimentation paths that the two groups take (showing the extreme polar opposites of experimentation) are shown in figure 1.6 below.

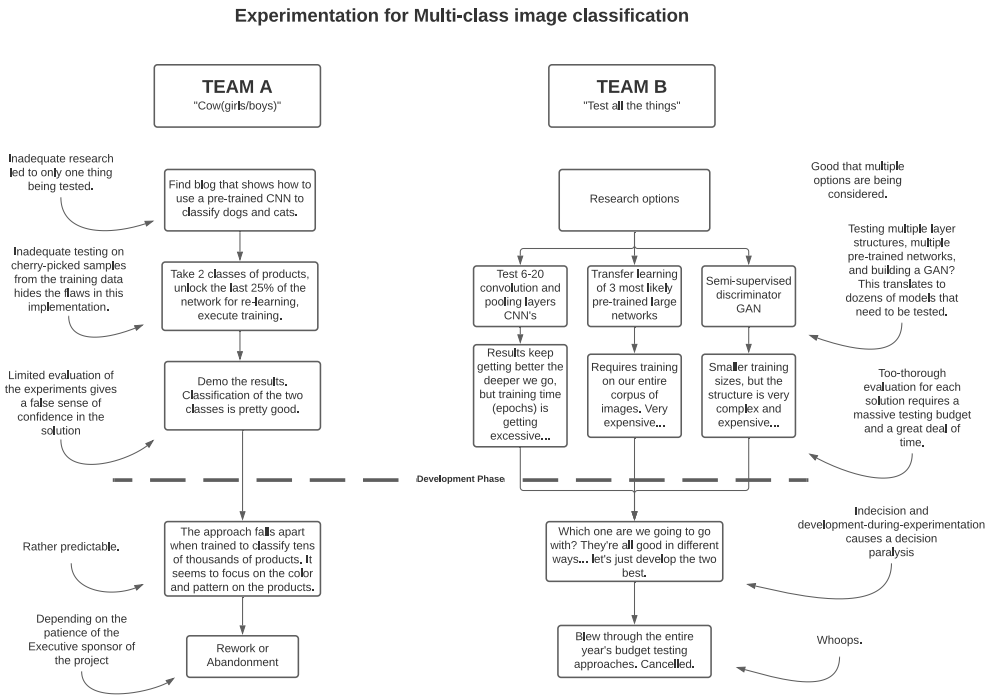


Figure 1.6 Juxtaposition of two extremes in experimentation processes. From a rushed and poorly designed experimentation process (Team A) to an overly complex, lengthy, and ludicrously expensive approach (Team B). Neither of these approaches would result in a successful project. The path to successful experimentation lies between these two, but in a highly structured way that focuses on answering the business problem properly in the shortest (and cheapest) manner.

Team 'A' in figure 1.6 is an exceedingly hyperbolic caricature of an exceptionally inexperienced DS team, performing only the most cursory of research. Using the single example blog post that they found regarding image classification tasks, they copy the example code, use the exact pre-trained TensorFlow-Keras model that the blog used, retrained the model on only a few hundred images of two of the many thousands of products from their corpus of images, and demonstrated a fairly solid result in classification for the holdout images from these two classes.

Due to the fact that they didn't do thorough research, they were unable to understand the limitations that were in the model that was chosen. With their rushed approach to creating a demo to show how good they could classify their own images, they chose a too-simplistic test of only two classes. With cherry-picked results and a woefully inadequate evaluation of the approach, this project would likely fail early in the development process (if someone on their leadership team was checking in on their progress), or late into the final delivery phases before production scheduling (when the business unit internal customer could see just how badly the approach was performing). Either way, using this rushed and

lazy approach to testing of approaches will nearly always end in a project that is either abandoned or cancelled.

Team 'B' in figure 1.6, on the other hand, is the polar opposite of team 'A'. They're an example of the 'pure researchers'; people who, even though they currently work for a company, still behave as though they are conducting research in a University Their approach to solving this problem is to spend weeks searching for an devouring cutting edge papers, reading journals, and getting a solid understanding of the nuances of both the theory and construction around various convolutional neural network (CNN) approaches that might work best for this project. They've settled on 3 broad approaches, each consisting of several tests that need to run and be evaluated against the entire collection of their training image dataset.

It isn't the depth that failed them in this case. The research was appropriate, but the problem that they got themselves into was that they were trying too many things. Varying the structure and depth of a custom-built CNN requires dozens (if not hundreds) of iterations to 'get right' for the use case that they're trying to solve. This is work that should be scoped into the development stage of the project, when they have no other distractions other than developing this single approach. Instead of doing an abbreviated adjudication of the custom CNN, they decided to test out transfer learning of 3 large pre-trained CNN's, as well as building a Generative Adversarial Network (GAN) to get semi-supervised learning to work on the extremely large corpus of classes that are needed to be classified.

Team B quite simply took on too much work for an experimentation phase. What they're left with at the point that they need to show demonstrations of their approaches is nothing more than decision paralysis and a truly staggering cloud services GPU VM bill. With no real conclusion on the best approach and such a large amount of money already spent on the project, the chances that the entire project will be scrapped is incredibly high.

While not the leading cause of project failure, an effective experimentation phase can, if done too incorrectly, stall or cancel an otherwise great project. There are patterns of experimentation that have proven to work remarkably well for ML project work, though, the details of which lie somewhere between the paths that these two teams took. We will be covering this series of patterns and ways in which they can be adapted to any ML-based project at length in Chapters 2, 3, and 4.

1.2.4 Development

While not precisely a major factor for getting a project cancelled directly, having a poor development practice for ML projects can manifest itself in a multitude of ways that can completely kill a project. It's usually not as directly visible as some of the other leading causes but having a fragile and poorly designed code base and poor development practices can actually make a project harder to work on, easier to break in production, and far harder to improve as time goes on.

For instance, let's look at a rather simple and frequent modification situation that comes up during the development of a modeling solution: changes to the feature engineering. In figure 1.7 below, we see two Data Scientists trying to make the exact same set of changes in two different development paradigms. The left region is a monolithic code base, all of the logic for the entire job is written in a single notebook through scripted variable declarations

and functions. On the right side, is a modularized code base written in an Integrated Development Environment (IDE). While both changes are identical in their nature (Julie is adding a few fields to the feature vector and updating encodings for these new fields, while Joe is updating the scaler used on the feature vector), the amount of effort and time spent getting these changes working in concert with one another is dramatically different.

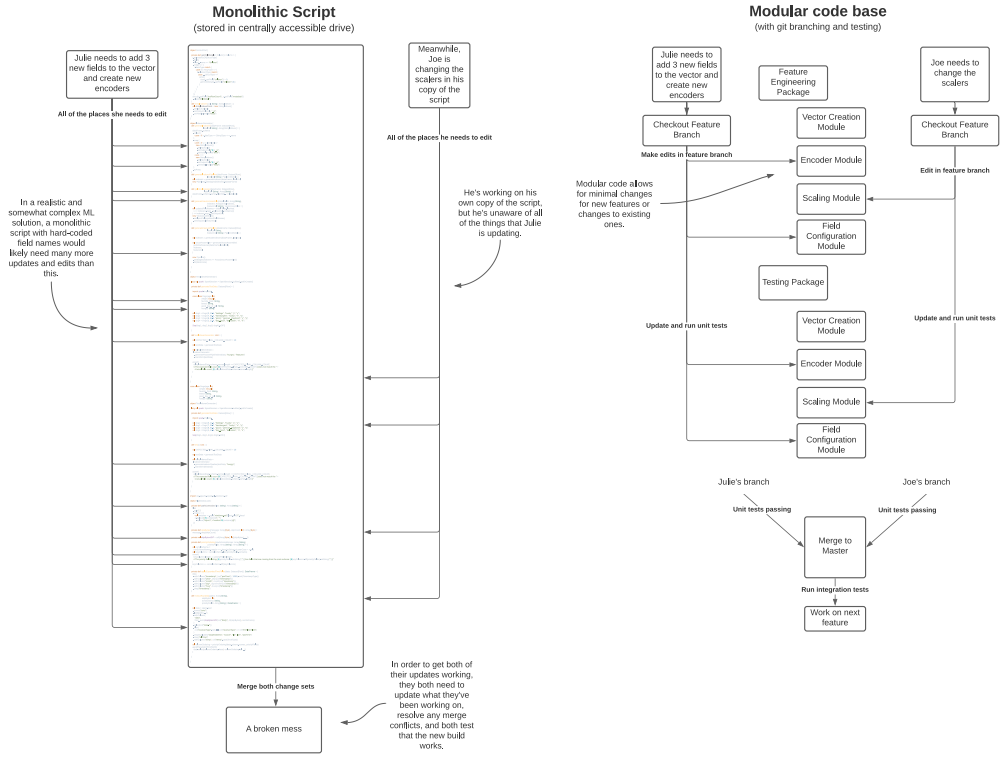


Figure 1.7 The juxtaposition of notebook-centric monolithic code bases (left) vs. modular code bases (right). In both sections, there are two data scientists working on two separate improvements to their project. Julie needs to add a few fields to the feature vector to improve predictive performance, while Joe needs to update the scaling being done to the feature vector. In the monolithic approach, not only does the script need to change in dozens of places (which is error prone) but merging of their two improvements is going to be extremely tricky and will require both of them to update their implementations to make it work. In the modular world, with proper unit testing, code architecture, and a branching strategy, the ease of making these updates is dramatically improved for both of them.

Julie, in the left side monolithic approach is likely going to have a lot of searching and scrolling to do, finding each individual location where the feature vector is defined and adding her new fields to collections. Her encoding work will need to be correct and carried throughout the script in the correct places as well. It's a daunting amount of work for any

sufficiently complex ML code base (where the number of lines of code for feature engineering and modeling combined can reach to the thousands if developed in a scripting paradigm) and is prone to frustrating errors in the form of omissions, typos, and other transcription errors.

Joe, meanwhile, in the left-hand side, has far fewer edits to do, but is still subject to the act of searching through the long code base and relying on editing the hard-coded values correctly.

The real problem with the monolithic approach comes when they try to incorporate each of their changes into a single copy of the script. As they both have mutual dependencies on one another's work, they will both have to update their code and select one of their copies to serve as a 'master' for the project, copying in the changes from the other's work. It will be a long and arduous process, wasting precious development time and likely requiring a great deal of debugging to get correct.

The right hand side however, is a different story. With a fully modularized code base registered in git, the both of them can each check out a feature branch from master, make their small edits to the modules that are part of their features, write some new tests (if needed), run their tests, and submit a pull request. Once their work is complete, due to the configuration-based code and the fact that the methods in each of the modules classes can act upon the data for their project through leveraging the job configuration, each of their feature branches will not impact one another's and should just work as designed. They can cut a release branch of both of their changes in a single build, run a full integration test, and safely merge to master, confident in the fact that their work is correct.

Writing code in this manner (modular design, written in an IDE) is a large departure for many Data Scientists. We've learned in interactive notebooks, and many of us still use notebooks quite frequently for prototyping ideas, experimentation, and for analysis of our work (myself included). However, by adopting this alternate way of writing ML code (porting prototype scripts and functions into object-oriented or functional programming paradigms), projects can support many users developing new features for them simultaneously, as well as ensuring that each new idea and bit of functionality is tested fully to eliminate difficult to track down bugs. The overhead in time and effort associated with creating an ML code framework based in these long-ago proven paradigms of software development will be thoroughly worth it once even the 2nd change to the code base is needed to be done.

We will be covering all of these topics regarding development paradigms, methodology, and tools in depth from Chapter 3 onwards throughout the book.

1.2.5 Deployment

Perhaps the most confusing and complex part of ML project work comes at the point long after a powerfully accurate model is built. The path between the model creation and serving of the predictions to a point that they can be used is nearly as difficult and its possible implementations as varied as there are models to serve prediction needs.

Let's take a company that provides analysis services to the fast food industry as an example for this section. They've been fairly successful in serving predictions for inventory management at region-level groupings for years, running large batch predictions for the per-day demands of expected customer counts at a weekly level, submitting their forecasts as bulk extracts each week.

A new business segment has opened up, requiring a new approach to inventory forecasting at a per-store level, with a requirement that the predictions respond in near-real-time throughout the day. Realizing that they need to not only build a completely different ensemble of models to solve this use case, the DS team focuses most of their time and energy on the ML portion of the project. They didn't realize that the serving component of this solution would need to rely on not only a REST API to serve the data to individual store owners through an application, but that they would have to be frequently updating the per-store forecasts fairly frequently throughout the day.

After coming up with an architecture that supports the business need (months after the start of the project, well after the modeling portion of the project had been finished), they proceed to build it with the assistance of some Java software engineers. It wasn't until after the first week of going live that the business realized that the implementation's in cloud computing costs are more than an order of magnitude higher than the revenue they are getting for the service. The new architecture is shown in figure 1.8 below, on the right portion of the figure. The left shows what they had previously had experience with.

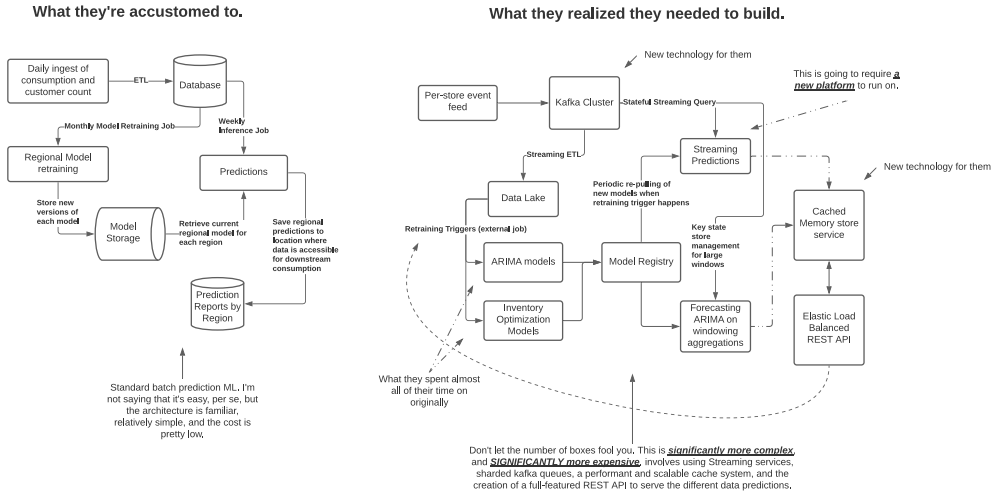


Figure 1.8 The comparison of what the business had been used to (on the left) and what they eventually realized they needed to build (on the right) to solve this new ML request. The sheer amount of new technologies that need to be added into their tech stack, the new platforms that the team needs to learn, and the cost of running this service at the scale that they need created such an enormous unknown increase in scope to the project that it had to be completely changed in order to get it working at-budget. Not thinking of the details of deployment and serving can completely derail projects and if it's not evaluated prior to the start of the project, it's typically found out very late in the development cycle.

It doesn't take long for the project to get cancelled and a complete redesign of the architecture and modeling approach to be commissioned to keep the costs down.

This is a story that plays out time and again at companies implementing ML to solve new and interesting problems. Without focusing on the deployment and serving, the chances of project success will be rather limited due not to the fact that it can't be developed, but rather that the engineering of the solution could cost far more money than the project brings in.

Thinking of deployment and serving with a skeptical eye focused on how much it's going to cost to run, maintain, and monitor is a great habit to have that will help to inform not only the development of the solution, but the feasibility of the general idea that the project is intended to implement. After all, there's no sadder death to an ML project than the one that forces a great solution to be turned off because it simply costs too much to run.

Figure 1.9 below shows some of (not all, by any stretch of the imagination) the elements to think about with regards to serving of prediction results. Notice the focus on relative expense for each section. Having this information analyzed very early in a project can set the expectation for how expensive the project is going to be so that the palatability of the cost of running it can be measured before the cost of development is incurred by the company.

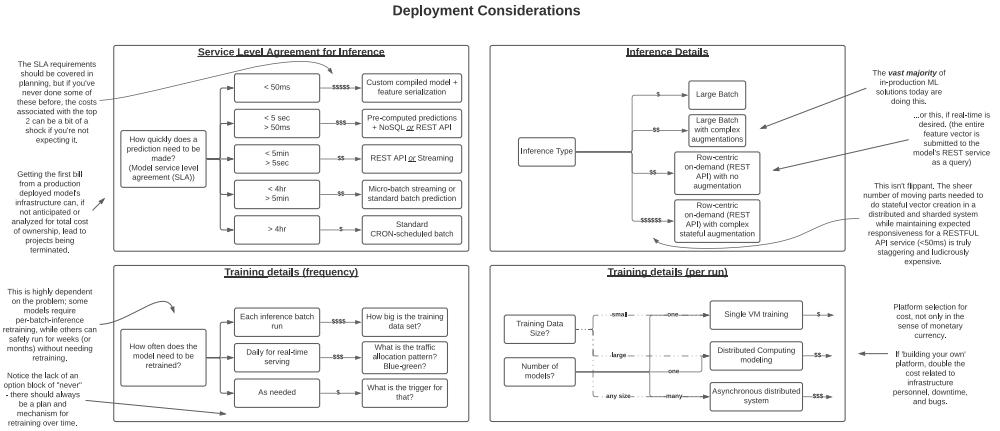


Figure 1.9 Deployment, serving, and training considerations for a project that are focused on the costs associated with different types of project work. There are many more aspects to consider, that we will be touching on throughout this book, but these 4 elements are the most critical aspects which, when overlooked or ignored, can frequently kill a project right after it gets turned on in production.

We will be covering each of these sections from figure 1.9 (and many others that affect the cost of ML), the considerations for serving, platform selection, build vs buy, and data volume costs associated with modeling throughout this book. It's not the most exciting part of ML Engineering, nor is it the most frequently thought about (until it's too late, usually), but it can be the fastest way to get a project cancelled, and as such, should be considered quite seriously.

1.2.6 Evaluation

The absolutely worst way of getting an ML project cancelled or abandoned is one of budgetary reasons. Typically, if the project has gotten into production to begin with, the upfront costs associated with developing the solution were accepted and understood by the leadership at the company. The budgetary assassination of a project that evaluation of a model deals with is entirely different though.

Imagine a company that has spent the past 6 months working tirelessly on a new initiative to increase sales through the use of predictive modeling. They've followed through best practices throughout the project's development, making sure that they're building exactly what the business is asking for, focusing their development efforts on maintainable and extensible code, and have pushed the solution to production. The model has been performing wonderfully over the past 3 months. Each time the team has done post-hoc analysis of the predictions to the state of reality afterwards, the predictions turn out to be eerily close.

Figure 1.10 then rears its ugly head with a simple question from one of the Executives at the company who is concerned about the cost of running this ML solution.

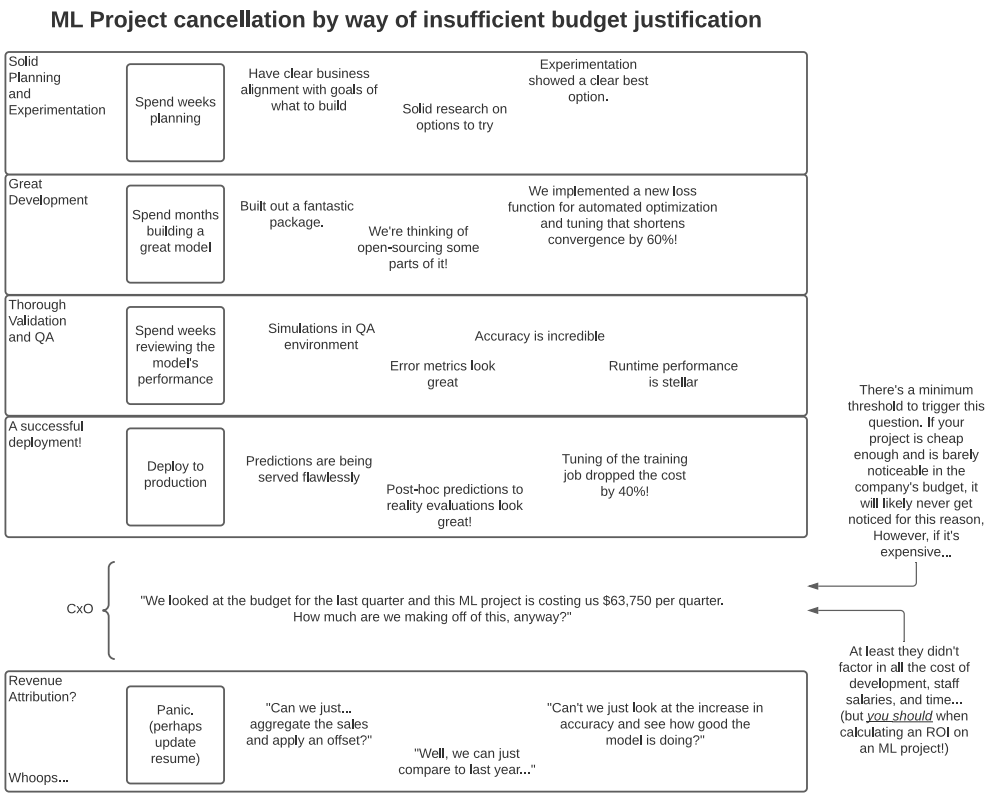


Figure 1.10 That point that every ML professional never wants to have happen if they aren't prepared for it. "Prove to me that your work is useful" can be very stressful if one is not prepared to answer that question, as it requires a specific approach to releasing a predictive capability to production, ensuring that an experiment of 'the world without my model' vs 'the world with my model' is conducted and accurately measured. Not having this data puts the project up for culling at the next available cost-cutting session done by C-suite personnel.

The one thing that the team forgot about in creating a great ML project is in thinking of how to tie their predictions to some aspect of the business that can justify its existence, as shown in figure 1.10. The model that they've been working on and that is currently running in production was designed to increase revenue, but when scrutinized for the cost of using it, the team realized that they hadn't thought of an attribution analytics methodology to prove the worth of the solution. Can they simply add up the sales and attribute it all to the model? No, that wouldn't be even remotely correct. Could they look at the comparison of sales vs last year? That wouldn't be correct to do either, as there are far too many latent factors impacting the sales.

The only thing that they can do to give attribution to their model is to perform AB testing and use sound statistical models to arrive at a revenue lift (with estimation errors) calculation to show how much additional sales are due to their model. However, the ship has

already sailed as it's been fully deployed for all customers. They lost their chance at justifying the continued existence of the model. While it might not be shut off immediately, it certainly will be on the chopping block if the company needs to reduce its budgetary spend.

It's always a good idea to think ahead and plan for this case. Whether it's happened to you yet or not, I can assure you that at some point it most certainly will. It is far easier to defend your work if you have the ammunition at the ready of validated and statistically significant tests showing the justification for the model's continued existence.

We will be covering approaches of building AB testing systems, statistical tests for attribution, and bandit algorithms in Part 3 of this book.

1.3 The goals of ML Engineering

In the most elemental sense, the primary goal of any DS is to solve a difficult problem through the use of statistics, algorithms, and predictive modeling that is either too onerous, monotonous, error-prone, or complex for a human to do. It's not to build the fanciest model, to create the most impressive research paper about their approach to a solution, or to search out the most exciting new tech to force into their project work.

The first and foremost goal of applying Software Engineering fundamentals (DevOps) to the world of ML is the very reason why DevOps was created. It's to increase the chances of having efficient project work being done and making it easier to manage incredibly complex code bases. It's to attempt to eliminate the chances of projects getting cancelled, code being abandoned, and solutions failing to see the light of day.

We're all here in this profession to solve problems. We have a great many tools to play with, a lot of options to consider, and an overwhelmingly complex array of knowledge that we need to attain and maintain in order to be effective at what we do. It's incredibly easy to get lost in this overwhelming avalanche of complexity and detail, only for our project work to suffer a cancellation (budget, time, or complexity), abandonment (unmaintainable and / or fragile code), or re-prioritization (poor objectives, unstable predictions, or lack of business need). These are all preventable through the judicious application of a core set of rules.

By focusing on the core aspects of project work that have been highlighted in section 1.2 and are covered in greater detail throughout this book, you can get to the true desired state of ML work: seeing your models run in production and have them solve a true business problem. The goal in the use of these methodologies, technologies, and design patterns is to help to focus your time and energy on solving the problems that you were hired to solve so that you can move onto solving more of them, making your company and yourself more successful with all of the benefits that predictive modeling has to offer. It's to reduce those high failure and abandonment rates so that your future work can be something other than focusing on apologies, rework, and constant maintenance of ill-conceived solutions.

You can do this.

There is an entire industry out there that is designed to convince you that you can't. That you need to hire them to do all of this complex work for you. They make a great deal of money doing this.

But trust me, you can learn these core concepts, can build a team that follows a methodology to approach ML work that can dramatically increase the success rate of project work. It may be complex and rather confusing at first,

but following the guidelines and using the right tooling to help manage the complexity can help any team develop quite sophisticated ML solutions that won't require massive budgets or consume all of the free time that a DS team has in 'keeping the lights on' for poorly implemented solutions.
You got this.

Before working into the finer details of each of these methodologies and approaches for ML Engineering work, see the outline detailed below in figure 1.x. This is effectively a process flow plan for production ML work. Your own work may look either shockingly similar to this, or significantly less complex than what is shown. The intent of showcasing this here is to introduce the concept of topics that we will be covering throughout the subsequent chapters.

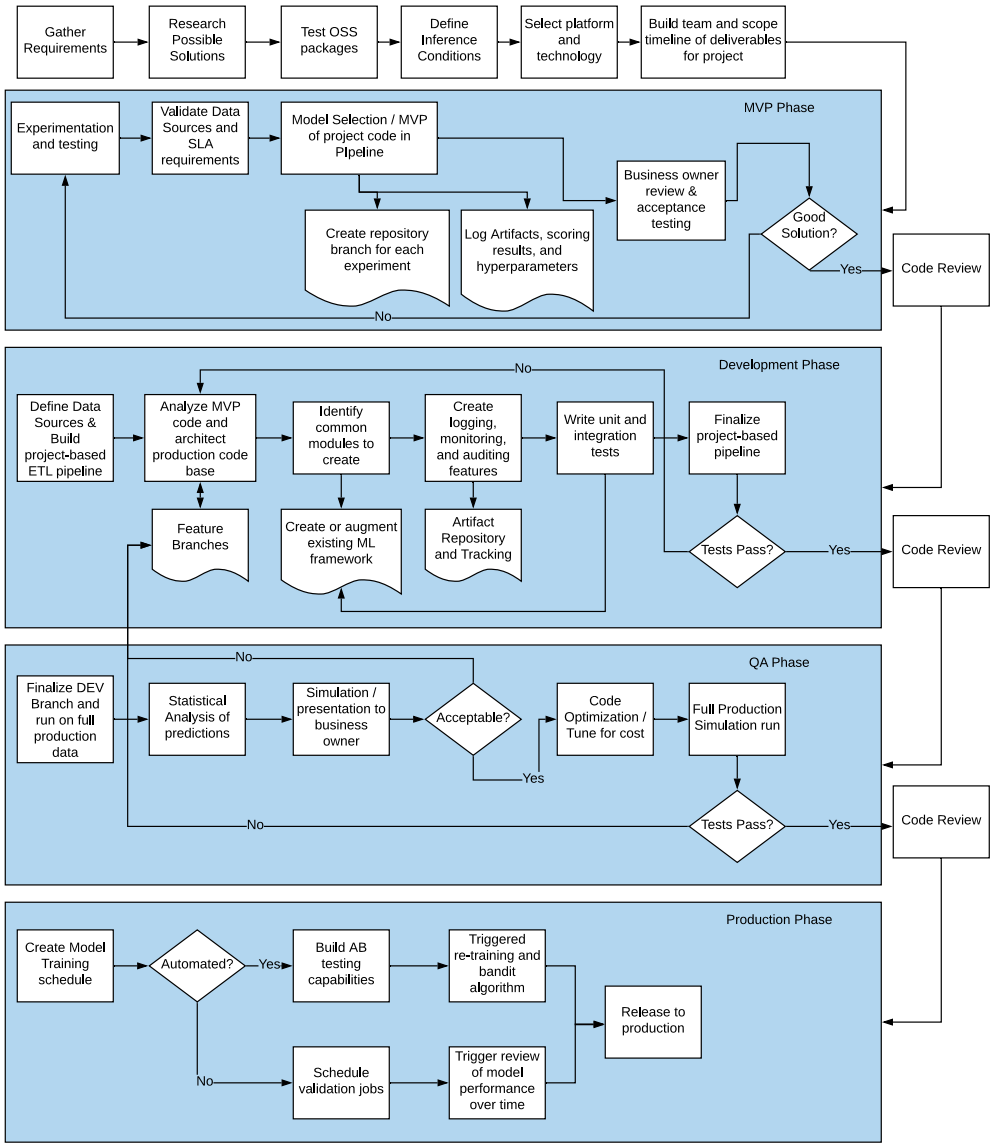


Figure 1.11 The ML Engineering methodology component map. Also, a visual table of contents for this book.

From the top section in figure 1.11 (the planning and scoping phases, covered in chapter 2), to the MVP phase (spanning chapters 3 through 5), all the way to production deployment phases, we will be moving through different aspects of this flowing path through a number of

different scenarios and applications of ML, illuminating the benefits of following this paradigm.

As mentioned before, this is the proven way to ensure that your ML project work will actually meet that base goal that we all strive for: to make something useful that isn't a nightmare to maintain; to use the art of data, the science of mathematical algorithms, and our limitless creativity to solve problems.

1.4 Summary

- The leading causes of the high rates of ML project work failure in industry are due to a failure to follow (or having an ignorance of) the 6 tenets of ML Engineering.
- ML Engineering is a guide; both a toolbox and a map that can help guide project work involving Data Scientists to ensure that their work is adhering to good engineering principles and is focused on solving a business need.
- We've seen the core components at a very high level. Throughout the rest of this book, we'll be delving quite deep into each of these topics, through the use of active examples, project solutions, and simulations of decisions to help give you the tools that you need to build successful, maintainable, and resilient code bases that employ machine learning.

2

Your Data Science could use some Engineering

This chapter covers

- **Defining the skills, tasks, and elements of ML Engineering that augment Data Science project work**
- **Explaining the principle of seeking the simplest approach to solving Data Science problems and why a focus on simplicity is important**
- **Defining a manifesto for Data Science project work**
- **Explaining the core functional components of ML Engineering and how they complement DS knowledge and skills**

In the previous chapter, we covered the justification for ML Engineering, focusing entirely on the 'what' aspect of this professional approach to Data Science (DS) work. This chapter is focused on the 'why'. The continuing maturity of DS in companies around the world has created not only fundamental methodologies in approaching DS project work (which is what this book is about), but has also created a frenzied hype in companies getting started in applying ML to their problems that more often than not leads to failed projects, disappointment, and frustration. We're going to define the ecosystem of ML Engineering and give the core reasoning about why the standards have been developed.

ML Engineering (also known by the co-opted term 'MLOps' as an adaptation of 'DevOps', itself an Agile-inspired collection of tools, methodology, and processes for general software development) is not intended to be a specific job title, a contradictory approach to Data Science (DS) work, nor a realm of responsibility wholly divorced from DS solutions. Rather, it is a *complementary* (and, arguably, critically necessary) additional set of tools, processes, and paradigms for conducting DS work, the end-goal of which is the creation of more

sustainable solutions to business problems at a much lower total cost of ownership. After all, the primary focus of all DS work is to solve problems. Conforming work patterns to a proven methodology that is focused on maintainability and efficiency translates directly to more problems getting solved and much less wasted effort.

2.1 Augmenting a complex profession with processes to increase success in project work

In one of the earliest definitions of the term 'Data Science' (as covered in the 1996 book "Data Science, Classification, and Related Methods", compiled by C. Hayashi, C. Yajima, H. H. Bock, N. Ohsumi, Y. Tanaka, and Y. Baba), the three main focuses are:

- Design for Data: specifically, the planning surrounding how information is to be collected and in what structure it will need to be acquired in order to solve a particular problem.
- Collection of Data: the act of acquiring said data.
- Analysis on Data: divining insights from the data through the use of statistical methodologies in order to solve a problem.

A great deal of modern Data Science is focused mostly on the last of these 3 items (although there are many cases where a DS team is forced to develop their own ETL), as the first two are generally handled by a modern Data Engineering team. Within this broad term, 'analysis on data', a large focus of the modern DS resides; applying statistical techniques, data manipulation activities, and statistical algorithms (models) to garner insights from and to make predictions upon data.

Figure 2.1 below, in the top portion, illustrates (in an intentionally brief and high-level manner) what the modern DS has as a focus from a technical perspective. These are the elements of the profession that most people focus on when speaking of what it is that we do; from data access to building complex predictive models utilizing a dizzying array of algorithmic approaches and advanced statistics. It isn't a particularly accurate assessment of what a Data Scientist actually does when doing project work, but rather focuses on some of the tasks and tools that are employed in solving problems. Thinking of Data Science in this manner is nearly as unhelpful as classifying the job of a software developer by listing languages, algorithms, frameworks, computational efficiency, and other technological considerations of their profession.

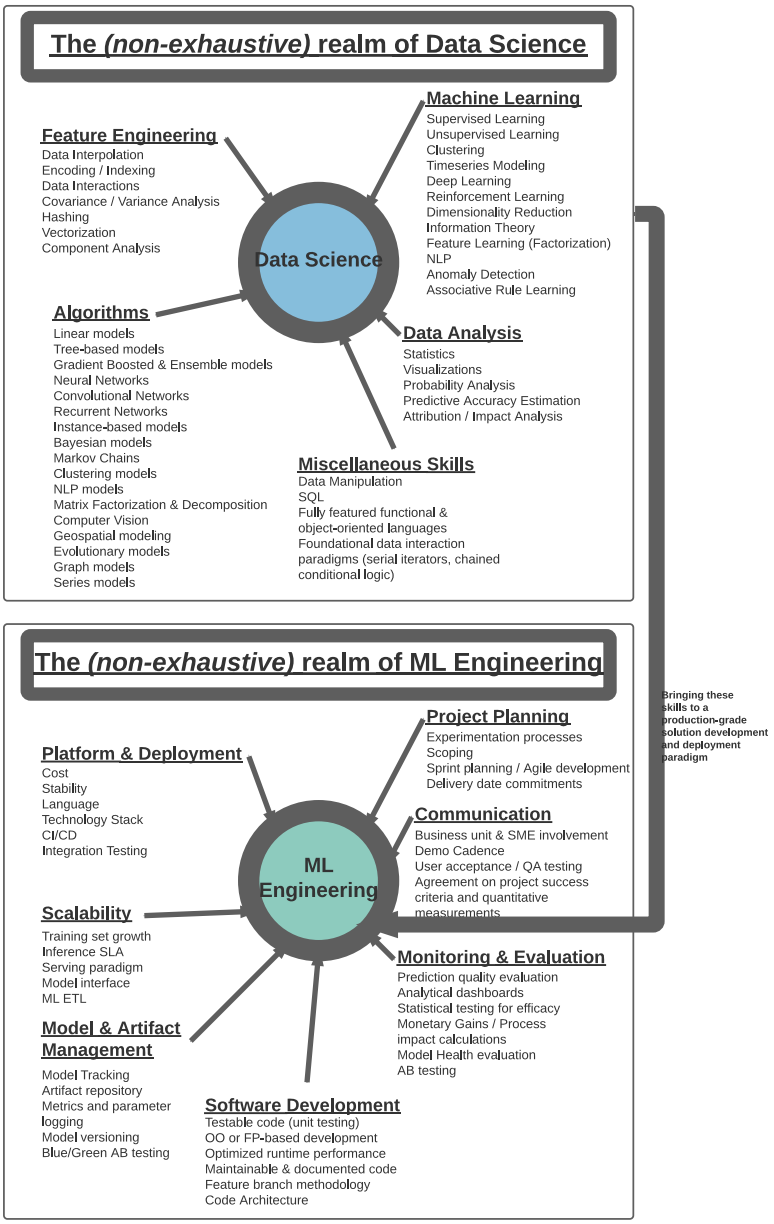


Figure 2.1 The 'core skills' of Data Science (above) and how they fit into the broader realm of methods, tools, and processes (below) that define successful DS project work. These broader set of skills, when mastered, can dramatically improve the chances of a project being declared successful (and having its results actually be used).

We can see in figure 2.1, how the technological focus of DS (that many practitioners focus on exclusively) from the top portion is but one aspect of the broader system that is shown in the bottom portion. It is in this region, ML Engineering, that the complementary tools, processes, and paradigms provide a framework of guidance, foundationally supported by the core aspects of DS technology, to work in a more constructive way. ML Engineering, as a concept, is a paradigm that helps practitioners to focus on the only aspect of project work that truly matters - providing solutions to problems that actually work.

Where to start, though?

2.2 A foundation of simplicity

When it comes down to truly explaining what it is that a DS actually does, nothing can be more succinct than, "they solve problems through the creative application of mathematics to data." As broad as that is, it reflects the wide array of solutions that can be developed from recorded information (data). There is nothing proscribed (at least that I'm aware of) regarding expectations of what a DS does regarding algorithms, approaches, or technology while in the pursuit of solving a business problem. Quite the contrary, as a matter of fact. We are problem solvers, utilizing a wide array of techniques and approaches.

Unfortunately for newcomers to the field, many Data Scientists believe that they are only providing value to a company when they are using the latest and 'greatest' tech that comes along. Instead of focusing on the latest buzz surrounding some new approach catalogued in a seminal white paper or advertised heavily in a blog post, a seasoned DS realizes that the only thing that really matters is the act of solving problems, regardless of methodology. As exciting as new technology and approaches are, the effectiveness of a DS team is measured in the quality, stability, and cost of a solution that they provide.

As figure 2.2 below shows, one of the most important parts of ML work is in navigating the path of complexity when facing any problem. By approaching each new ask from a business with this mindset as the veritable cornerstone of ML principles (focusing on the simplest solution possible that solves the problem that the business has), the solution itself can be focused on, rather than a particular approach or fancy new algorithm.

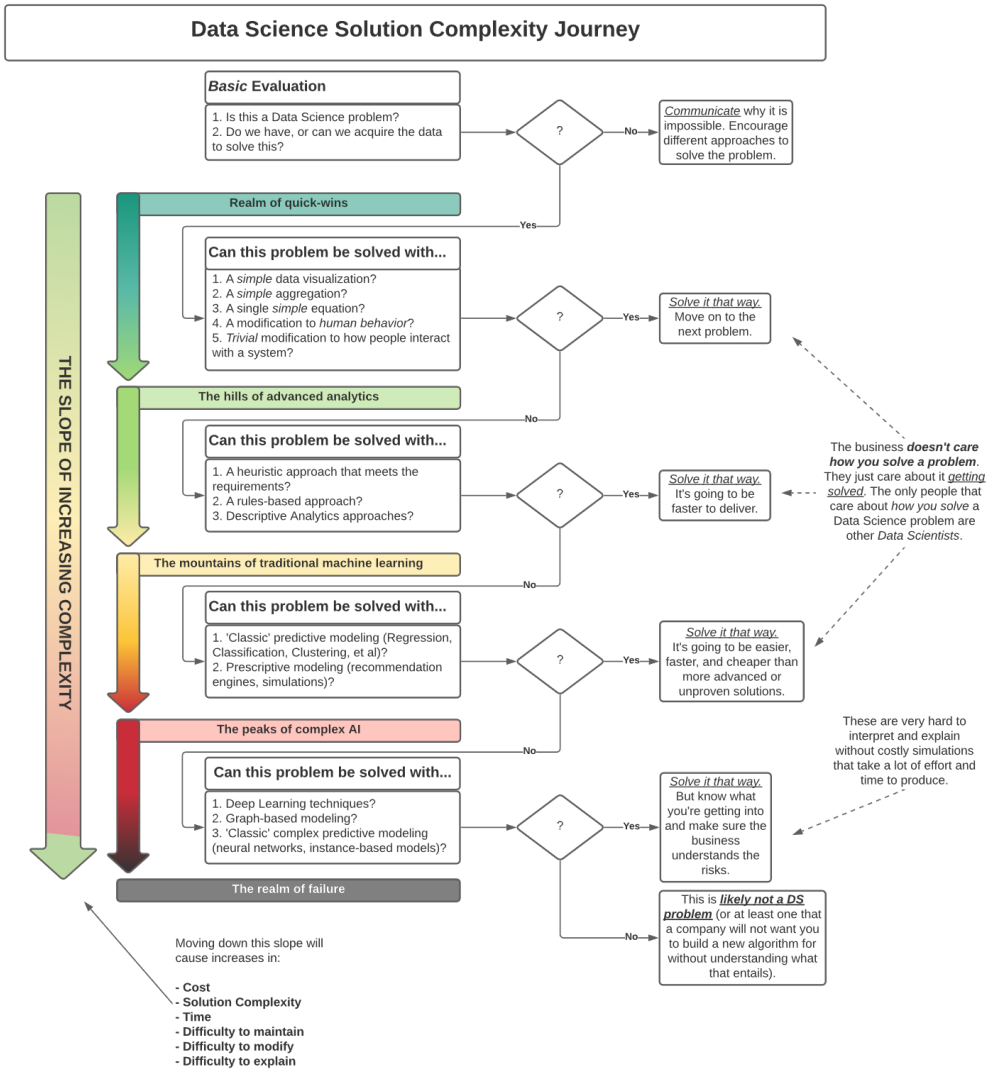


Figure 2.2 The decision path for Data Science projects to minimize complexity in implementations. From the point of basic evaluation (is this even a DS problem?) to the end (completely impossible to solve with current technology), the gating decisions in increasing order of complexity is the recommended approach for tackling any project work. The simplest solution that can still solve the problem is always going to be the best one. Heading for the 'most impressive' or 'latest fad' will always risk a project failing to materialize (or be used by the business) due to complexity, cost, time to develop, or interpretability.

Having a focus built around the principle illustrated in figure 2.2 (of pursuing the simplest possible implementation to solve a problem) is the foundation upon which all other aspects of

ML Engineering are built upon. It is by far the single most important aspect of ML Engineering, as it will inform all other aspects of project work, scoping, and implementation details. Striving to 'exit the path as early as possible' can be the single biggest driving factor in determining whether a project will fail or not.

In my own personal experience, the vast majority of problems that I've solved for companies have been in the first two blocks, with only a few dozen in the final two blocks.

2.3 Co-opting principles of Agile software engineering

DevOps brought guidelines and a demonstrable paradigm of successful engineering work to software development. With the advent of the Agile Manifesto, seasoned industry professionals recognized the failings of how software had been developed. Some of my fellow colleagues and I took a stab at defining an adapted version of these guiding principles, adapted to the field of Data Science, shown below in figure 2.3

Figure 2.3 The result of a thought experiment among Data Science colleagues. The left column are the guiding principles of the Agile Manifesto, while the right column are our own adaptations to these fundamental rules, applied to DS work. These guiding principles of project work have served all of us well through hundreds of projects that have been successful as production deployments.

As figure 2.3 shows, with a slight modification to the principles of the Agile development, we can come up with a base of rules surrounding the application of DS to business problems. We will be covering all of these topics, why they are important, and giving examples of how to apply them to solve problems throughout this book. While some are a significant departure from the principles of Agile, the applicability to ML project work has provided repeatable patterns of success for us and many others.

2.4 The foundation of ML Engineering

Now that we've seen the bedrock of DS work, let's take a brief look at the entire ecosystem of this system of project work that has proven to be successful through my many encounters in industry with building resilient and useful solutions to solve problems.

As mentioned in the introduction to this chapter, the idea of ML Engineering (MLOps) as a paradigm is rooted in the application of similar principles that DevOps has to software development. Figure 2.4 below shows what the core functionality of DevOps is.

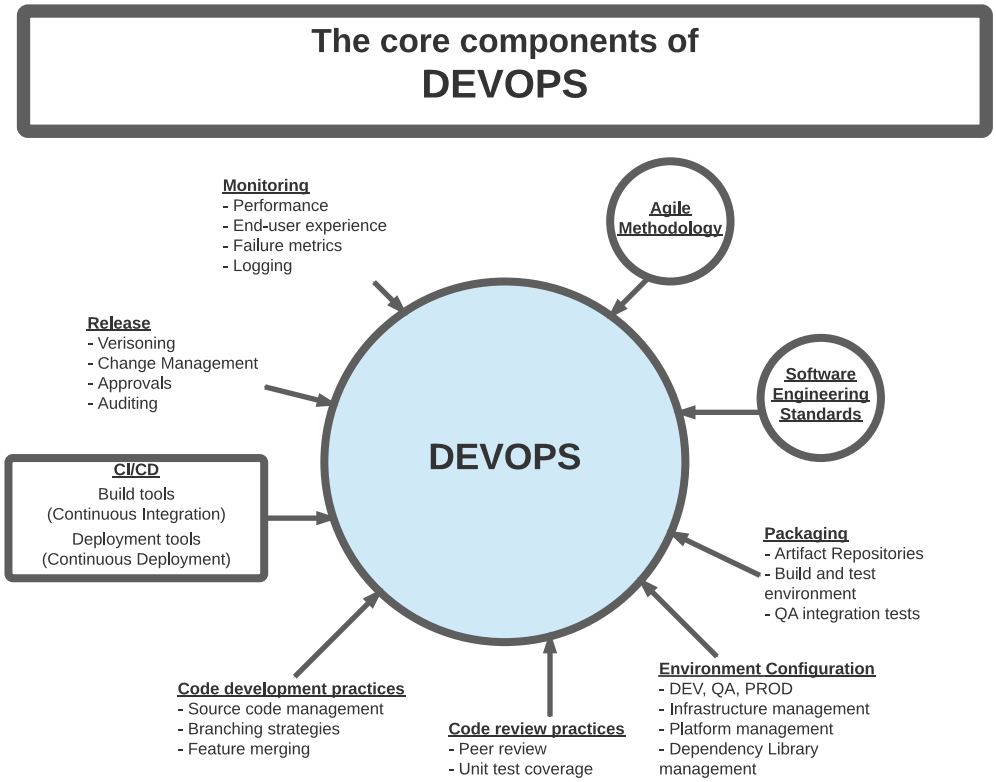


Figure 2.4 The components of DevOps. From processes such as Agile Methodology, a focus on fundamentals, tooling to assist development efforts, and methodologies to help developers produce higher quality code in a

shorter period of time, DevOps has revolutionized modern software development.

Comparing these core principles, as we did in section 2.3 to Agile, figure 2.5 shows the 'Data Science version' of DevOps – MLOps. Through the merging and integration of each of these elements, the most catastrophic events in DS work can be completely avoided: the elimination of failed, cancelled, or non-adopted solutions.

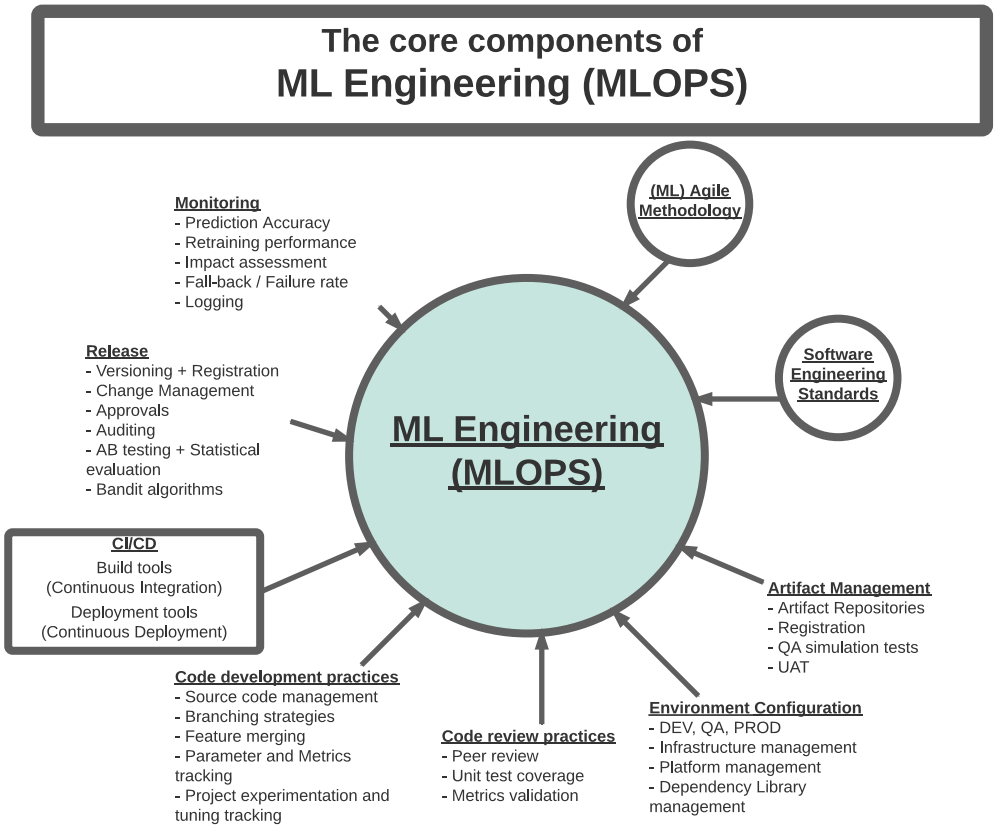


Figure 2.5 Adaptation of DevOps principles to ML project work. While some blocks are rather similar to DevOps (fundamentals, software engineering standards, and CI/CD), the additions that are specific to ML are all designed to aid in describing the elements that are part of successful production-grade ML solutions.

Throughout this book we will be covering not only the reasoning of why each of these elements shown in figure 2.5 are important, but also will show useful examples and active implementations that you can follow along with to further cement the practices in your own work. The goal of all of this, after all, is to make you successful. The best way to do that is to help you make your business successful by giving a guideline of how to address project work

that will get used, provide value, and be as easy as possible to maintain for you and your fellow DS team members.

2.5 Summary

- The application of standard processes, tools, and methodologies as an augmentation to Data Science skill sets help to ensure a higher rate of project success.
- The goal of any DS project should not be to use specific tooling or algorithms. Rather, striving for the simplest approach to solving a problem should always be a primary goal in all DS projects (even if it is little more than an analytics visualization).
- Focusing on an adapted set of principles from Agile development can help to define successful patterns of DS work for teams that has been proven to work, ensuring higher quality and more maintainable solutions.
- The ecosystem of ML Engineering (MLOps) is an adaptation of many of the processes and standards from within DevOps, adding in specific tooling and domain-specific elements in the pursuit of building resilient, maintainable, and production-capable DS solutions.

3

Before you model: Planning and Scoping a project

This chapter covers

- **How to effectively plan ML projects with a diverse team to reduce the chances of rework or project abandonment**
- **Scoping and how to estimate the amount of effort and the minimum number of features required to deliver demonstrations of projects on time and with the least amount of rework**

The two biggest killers in the world of ML projects are ones that have nothing to do with what most Data Scientists ever imagine. They aren't related to algorithms, data, or technical acumen. They have absolutely nothing to do with which platform you're using, nor with the processing engine that will be optimizing a model. The biggest reasons for projects failing to meet the needs of a business are in the steps leading up to any of those technical aspects: the planning and scoping phases of a project.

It's understandable why this might be, as most ML practitioners focus the vast majority of their training in university, their project work in research, and much of their lives leading up to working as a Data Scientist in focusing on the 'how' to build something. We study theory, algorithms, and write a great deal of code to answer specific questions that are presented. The focus on the complex nature of ML is always on the minute technical details and understanding how things work. This focus is counter-productive to project work at companies, and failing to set aside the focus on the 'how' in building something in order to instead focus on the 'what' and 'when' is what frequently causes so many ML projects to either fail out-right, or be so much less than what a business had expected that the wasted time, effort, and money in building the project is quickly regretted.

The only 'how' we're going to be covering in this chapter is in answering the question of how to prevent this from happening. We'll be covering why this paradigm shift in the thinking

of ML teams to focus less on 'how' and more on 'what' in project work can help reduce experimentation time, focus the team on building a solution that will work for a company, and planning out phased project work that incorporates subject matter expert (SME) knowledge from cross-functional teams to help to dramatically increase the chances of a successful project.

At the root of the issue in ML work, when building a solution for a company, is a failure in the business and the ML team to understand how one another think about projects that use predictive modeling. For a relatively simplistic project (a self-contained solution that an ML team can handle entirely on their own), it might not seem as these processes are essential, but figure 3.1 below shows how even simple-seeming project work can generate so much waste and frustration.

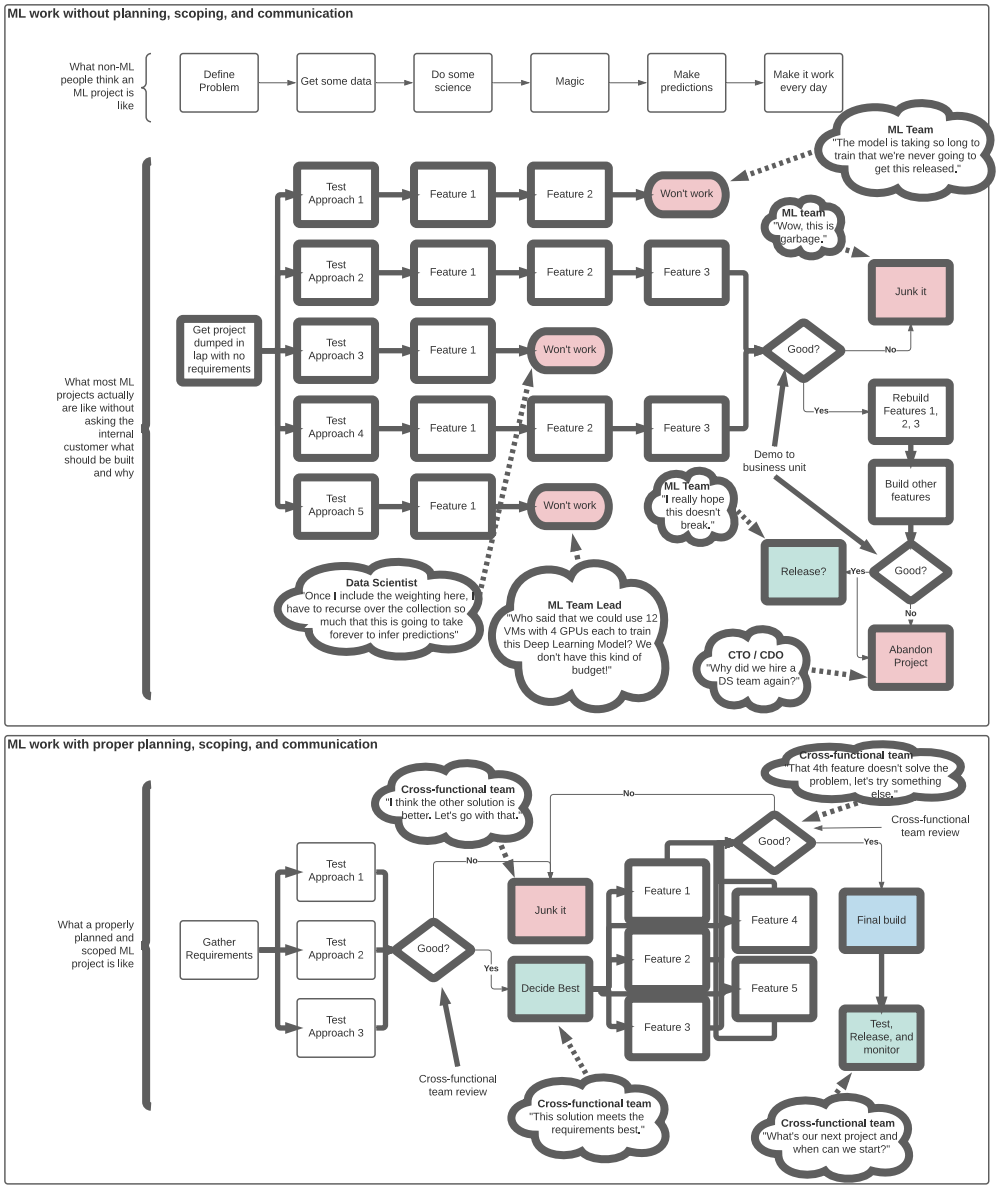


Figure 3.1 Contrasts of ML project development between incorporating proper planning, scoping, and cross-functional team communication (bottom), and that of pure chaos and assumptions (top). Projects in the top portion are both far more numerous and far more likely to fail than those adopting the bottom methodology.

The main question that many, who have endured the upper portion of figure 3.1 (from either side) ask is, "why does this happen, and what can we do about it?". While I can't make conjectures as to all of the myriad reasons about why this happens at companies (although I suspect it's due some to the complexity of ML, but mostly due to the hype that has surrounded it), what I can do is discuss how to go about getting into that bottom portion with project work: how to work more effectively to raise the chances of getting ML solutions into production.

As tooling in the ML community has grown more mature, the success rate of projects making their way into production has, counterintuitively, decreased. With all of these improvements to productivity, ease of 'getting a solution out the door', and the large-scale adoption of more aggressive Agile fundamentals being pushed onto ML teams, the core elements that were involved in the earlier period (the 'uncool' period of ML, back before anyone really cared about what the nerds in the back room of the 7th floor annex were doing), namely planning of projects and meticulous scoping, have largely been swept away as pointless wastes of time or have been overlooked out of sheer ignorance. We're going to be talking about those two key elements in this chapter – the absolutely critical parts of planning and scoping, focusing as well on the successful communication patterns amongst diverse teams of varying expertise that bind the success of the early (and subsequent) phases together.

Isn't planning, scoping, brainstorming, and organizing meetings a Project Manager thing?

While some ML practitioners may balk at idea of covering planning, communication, brainstorming, and other project-management-focused elements in a discussion on ML projects, in response I can only muster another piece of anecdotal evidence: that the most successful projects that I've been involved in have had the ML team lead work very closely with the other involved team leads and the project manager to deliver the project.

As a result of having the leads be so involved with the project management aspects of solutions, the teams typically endure far less work-related churn and rework, allowing them to focus on development with a holistic approach to get the best possible solution shipped to production.

Comparatively, the teams that operate in silos typically struggle to make a project work. Whether that is due to a failure to keep discussions in abstract terms (the ML team focusing discussions around implementation details or getting too 'in the weeds' about algorithms during meetings), thereby isolating others from contributing ideas to the solution, or due to the attitude of "we're not PMs… our job is to build models", the end result of working in a cross functional team without proper and effective communication patterns invariably leads to scope creep, confusion, and a general social antagonism between the warring factions of sub-teams within a project.

While it is mostly true that the vast majority of ML practitioners don't put much thought into efficiency of their work. It's completely understandable why this is, as the primary focus in academia, research, and the content of most algorithmic white papers is solely on the predictive capabilities of ML. They focus on accuracy, predictive capabilities, and runtime efficiencies of the training phases of models. With so much emphasis placed on unique solutions within academia, and an insular focus within the ML community placed solidly on the algorithm-centric focus of attaining leader-board ranked solutions it's not particularly surprising why most Data Scientists are pretty horrible at shifting their focus to solving a

problem in a way that has a measured balance between accuracy, cost, and timeliness of implementation. These are the core tenets of software development, after all, and most ML practitioners have not had to consider these aspects before (certainly not sacrificing accuracy for the sake of cost or delivering a solution on-time).

Even though ML practitioners don't generally place a focus on efficiency by their own accord (at least not until they're getting yelled at for taking forever to build a prototype) and instead focus on the quality of the work that they produce, eventually, it's inevitable to prioritize speed of delivery, at the expense of exploration of options and building out solutions to their fullest potential before showing it to anyone (as is wont in academia, research, and algorithm development). Unfortunately for those who are accustomed to a paradigm of wanting to have a fully fleshed-out solution or iterating through too many options comes with it a degree of frustration when applied to an application of ML at a company as the business unit and customers who are waiting for that model will rapidly run out of patience.

These are avoidable problems though. Shifting from having either insufficient or wholly absent planning (particularly in the experimentation and prototyping phases) and moving towards a pattern of conducting the early 'pre-work' planning of a project will have the most influence toward increasing the probability of delivering a successful project.

I can imagine fully the number of eyes rolling at these statements and can admit that, prior to getting a few dozen projects completely in corporate environments, I would have been right there with such a disdainful take. I may have objected with a statement such as, "But how can I possibly plan anything if I don't know if a model is going to work?" or "how could I possibly have an answer on the efficacy of an approach if I'm not trying every possible solution and fully building it out?". I hear you. I really do. But in the world of real-world production ML, time is not infinite, accuracy is not always the most important thing, and setting restrictions and controls on a project help focus everyone on the real goal of any project: getting it into production.

3.1 Planning: you want me to predict what?!

Before we get into how successful planning phases for ML projects are undertaken, let's go through a simulation of the genesis of a typical project at a company that doesn't have an established or proven process for initiating ML work. This may sound quite familiar to you, either now, or in your previous positions.

Let's imagine that you work at an e-commerce company that is just getting a taste for wanting to modernize their website. After seeing competitors tout massive sales gains by adding personalization services to their websites for years, the demand from the C-level staff is that the company needs to go all-in on recommendations. No one in the C-suite is entirely sure of the technical details about how these services are built, but they all know that the first group to talk to is the ML nerds. The business (in this case, the sales department leadership, marketing, and product teams) calls a meeting, inviting the entire ML team, with little added color to the invite apart from the title, "Personalized Recommendations Project Kick-off".

Management and various departments that you've worked with have been very happy with the small-scale ML projects that the team has built (fraud detection, customer valuation

estimation, sales forecasting, and churn probability risk models). Each of the previous projects, while complex in various ways from an ML perspective, were largely insular – handled within the scope of the ML team to come up with a solution that could be consumed by the various business units. At no point in any of these projects was there a need for subjective quality estimations or excessive business rules to influence the results; the mathematical purity of these solutions simply were not open to argument or interpretation – they are either right, or they are wrong.

Victims of your own success, the business approaches the team with a new concept: modernizing the website and mobile applications. They've heard about the massive sales gains and customer loyalty that comes along with personalized recommendations and they want you and the team to build one for incorporation to the website and the apps. They want each and every user to see a unique list of products greet them when they login. They want these products to be relevant, interesting to the user, and, at the end of the day, they want to increase the chances that the user will buy these items.

After a brief meeting where examples from other websites are shown, they ask how long before the system will be ready. You estimate about 2 months, based on the few papers that you've read in the past about these systems, and set off to work. The team creates a tentative development plan during the next scrum meeting, and everyone sets off to try to solve the problem.

You, and the rest of the ML team, assume that what management is looking for is the behavior shown in so many other websites, in which products are recommended in a main screen. That, after all, is personalization in its most pure sense: a unique collection of products that an algorithm has predicted will have relevance to an individual user. It seems pretty straight-forward, you all agree, and begin quickly planning how to build a data set that shows a ranked list of product keys for each of the website and mobile app's users, solely based on the browsing and purchase history of each member.

Hold up a minute. Isn't planning a project fully Waterfall Development? We're AGILE

Yes, I know this is going to be controversial.

I'm going to propose a radical idea here that many people don't seem to fully grasp when they claim that they're doing agile development in the ML space: you can still use scrum, extreme programming, feature-driven development, or whatever the next buzzword-worthy acronym-friendly term you want to use. Even in the Agile methodology, you still have a plan of the end state of the functionality of your software. You iterate, modify, and the minute details of each sprint are mutable as time goes on. I'm all for this methodology.

However, for an ML project, it's simply not feasible to swap between the core model implementation that you're going to choose. Finding out 3 months into a project that your scikit-learn solution can't scale to production data volumes, or that you're not finding much success with your TensorFlow implementation that you liberally borrowed from someone's blog is a show-stopper. The ML team, if they find an issue like this, will have to start over from line 1 of their code. Everything that they worked on up until that point is absolutely throw-away code.

This is why, in the interests of actually getting something out the door, some planning and concrete decisions need to be made after the experimentation phase is done. Plan what you're going to test, use agile methodologies to perform this experimentation (and use them throughout development too – they're great) but don't think that having a solid plan in place prior to writing code is a "waterfall" approach. It's just common sense to approach ML in this way for anyone who's had to throw away 4 months of code or realize that the project has to be abandoned and people are

going to get fired because of the simple fact that a team wanted to "be purely scrum focused and iterate during the project". There's enough chaos in ML development as it is; don't add more by being cavalier.

For the next several sprints (see callout above on sprints and agile development in ML), **you all studiously work in isolation**, **testing dozens of implementations** that you've seen in blog posts, consumed hundreds of papers worth of theory on different algorithms and approaches to solve an implicit recommendation problem, finally building out a minimum viable product (MVP) solution using alternating least squares (ALS) that achieves a root mean squared error (RMSE) of 0.2334, along with a rough implementation of ordered scoring for relevance based on prior behavior. Brimming with confidence that you have something amazing to show the business team sponsor, you head to the meeting armed with the testing notebook, some graphs showing the overall metrics, and some sample inference data that you believe are going to truly impress the team.

You start by showing the overall scaled score rating for affinity, displaying the data as an RMSE plot, as shown in figure 3.2 below.

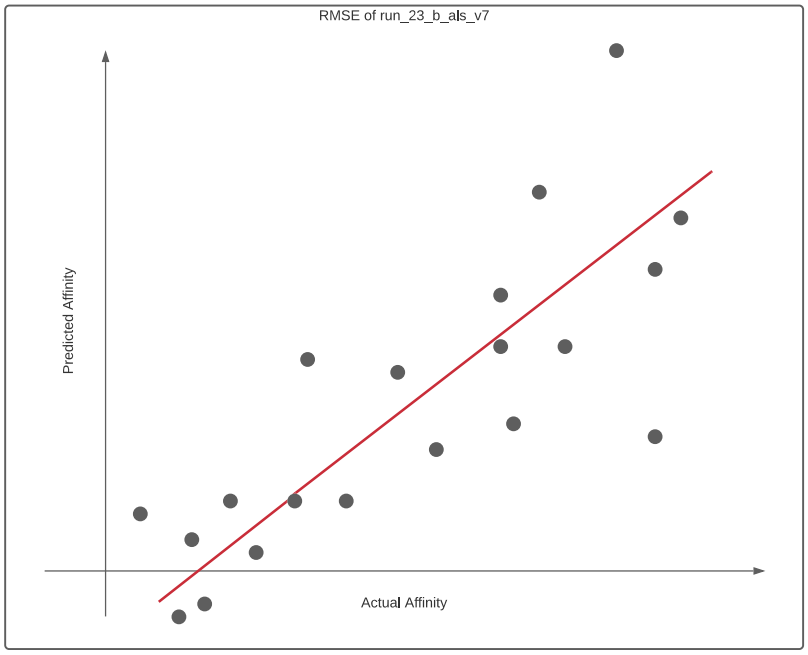


Figure 3.2 Displaying a metric chart of a model will do you no favors with a project team. Thinking about an audience and presenting a relatable simulation of the project's end goal will build confidence in a solution and let everyone feel as though they can contribute. It's best to keep charts, plots, and metrics such as these internal to the ML team. (Showing this to a non-technical audience is the best way to get people to ignore everything else that you're going to say). Instead, focus on visualizations that can convey that you're solving the problem with your modeling in terms that the business team will be familiar with.

The response to showing the chart in figure 3.2 is lukewarm at best. When display, a bevy of questions arise, focused on what the data means, what the line that intersects the dots means, and how the data was generated begins to derail the discussion. Instead of a focused discussion about the solution and the next phase of what you'd like to be working on (increasing the accuracy), the meeting begins to devolve into a mix of confusion and boredom. In an effort to better explain the data, you show a quick table of rank effectiveness using non-discounted cumulative gain (NDCG) metrics to show the predictive power of a single user that was chosen at random, as shown in figure 3.3.

Prediction DCG Scores - overall score 0.961

UserID	ItemID	Actual Ranking	Predicted Ranking	DCG value
38873	25	1	3	3.0
38873	17	2	2	1.26186
38873	23	3	6	3.0
38873	11	4	1	0.403677
38873	19	5	5	1.934264
38873	3	6	4	1.424829

Figure 3.3 NDCG calculations for a recommendation engine for a single user. Definitely not a recommended metric to show to a broader team of non-ML people (debatable whether one should show this to ML people as well in the interests of preventing philosophical debate of the efficacy of such scoring implementations). (Showing this to a non-technical audience is the 2nd best way to get them to ignore you. If shown with figure 3.2, you're doomed as the amount of time spent explaining what these mean is going to derail the demo so egregiously that the audience will no longer be thinking about the project, but rather about what these scores mean to the success of providing accurate predictions.)

The first chart created a mild sense of confusion, but the table in figure 3.3 brings nothing but overt hostility and annoyance. No one understands what is being shown and cannot see the relevance to the project. The only thing on everyone's mind is: "Is this really what weeks of effort can bring? What has the Data Science team been doing all this time?".

During the explanation that the DS team is doing for these two visualizations, one of the marketing analysts begins looking up the product recommendation listing for one of the team members' accounts in the sample data set that was provided for the meeting. The results of their recommendations (and their thoughts as they bring up the product catalog data for each of the recommendations in their list) are in figure 3.4 below.

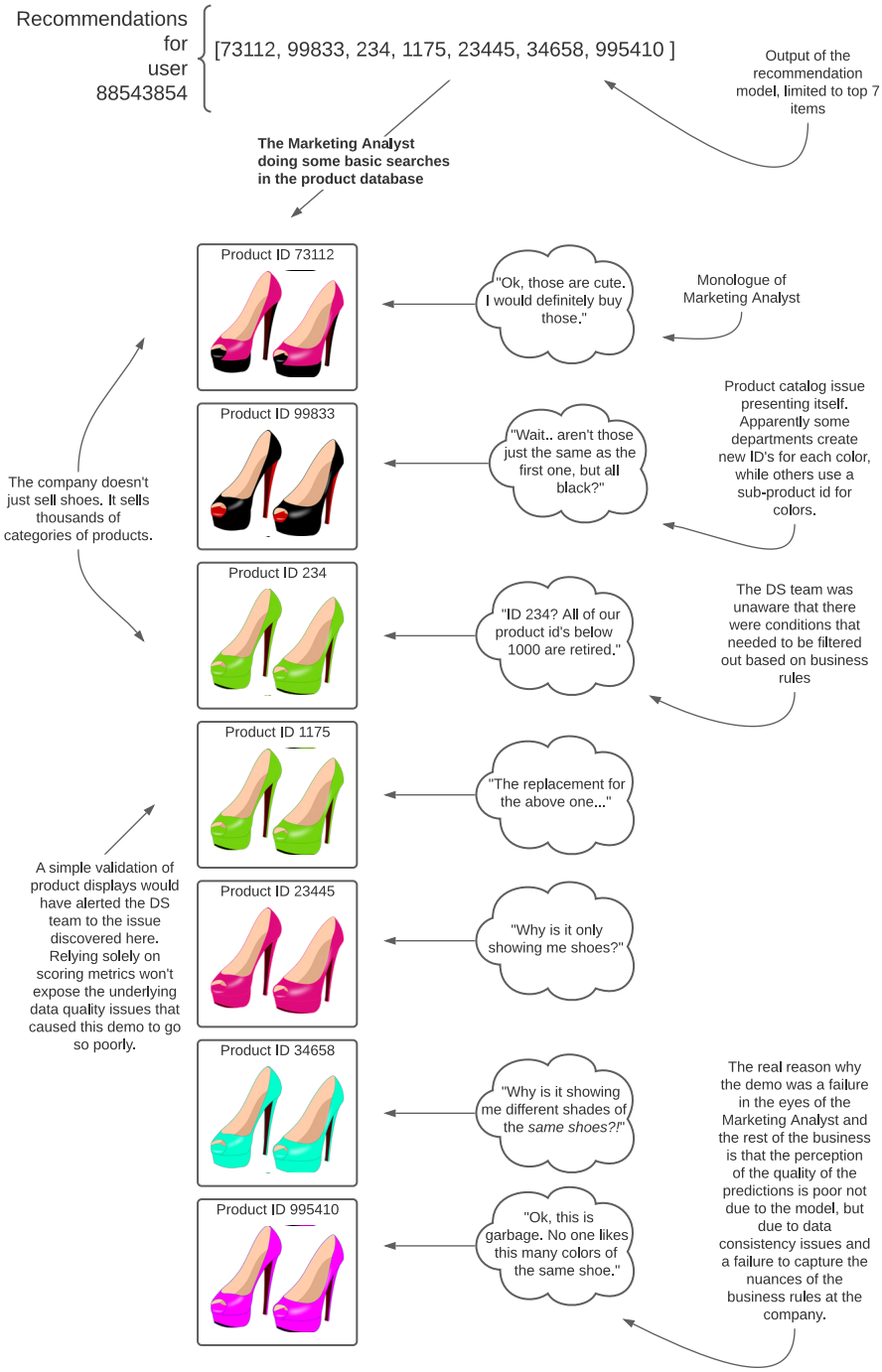


Figure 3.4 The marketing analyst analyzing their recommendations with the power of imagery. If the DS team had visualized a sampling of the recommendations, they might not have known intuitively the reasons for the duplicated products and the details around why showing recommendations of 6 different colors of the same shoe would be odd to a fashion expert, but it could have allowed them to bring in an expert to discuss the results with. As shown by the monologue on the right of the image, the analyst knew why the results were so poor for them, but each of these reasons were unknown to the DS team. Both a lack of an inspection of the predictions and a lack of adequate project planning were the causes of this ignorance that led to the demo being an outright flop.

The biggest lesson that the DS team learned from this meeting was not, in fact, the necessity to validate the results of their model in a way that would simulate how an end-user of the predictions would react. Although an important fact, and one that is discussed in the callout below, it is trumped quite significantly in the realization that the reason that the model was received so poorly is that they didn't plan for the nuances of this project properly.

Don't blindly trust your metrics

It is incredibly tempting, when doing particularly large-scale ML, to rely quite heavily upon error metrics and validation scores for models. Not only are they the only truly realistic means of measuring objective quality for predictions on large data sets (which many of us deal with quite frequently these days), but they're many times the only real valid quantitative means of adjudicating the predictive quality of a particular implementation.

However, it is incredibly important to not just rely on these model scoring metrics alone. Do use them (the appropriate ones for the work at hand, that is), but supplement them with additional means of getting subjective measurements of the efficacy of the prediction.

As shown in figure 3.4, a simple visualization of the predictions for an individual user uncovered far more both objective and subjective quality assessments than any predictive ordering scoring algorithm or estimation of loss could ever do.

One thing to keep in mind for this additional 'end-use simulation sample' evaluation is that it shouldn't be done by the DS team, unless they are adjudicating the quality of the prediction for data that they themselves are considered to be subject matter experts (SME's) in regard to. For the use case that we're discussing in this chapter, it would behoove the DS team to partner up with a few of the marketing analysts to do a bit of informal quality assurance (QA) validation before showing results to the larger team.

The DS team simply hadn't understood the business problem from the perspective of the team members in the room who knew where all of the proverbial 'bodies were buried' in the data and who have cumulative decades of knowledge around the nature of the data and the product.

How could they have done things differently?

The analyst that looked up their own predictions in figure 3.4 uncovered a great many problems that were obvious to them once seeing the predictions for their account. They saw the duplicated item data due to the retiring of older product ID's and they likewise instantly knew that the shoe division used a separate product ID for each color of a style of shoe, both core problems that caused a very poor demo.

All of the issues found, causing a high risk of project cancellation, were due to improper planning of the project.

3.1.1 Basic planning for a project

It's incredibly easy to assign blame in hindsight. Depending on the team member asked, in a post-mortem for the failed presentation for the recommendation engine prototype, dramatically different reasons would be presented.

Let's start by breaking down that first meeting: the planning phase. With limited knowledge of *how* similar e-commerce websites actually implement personalized recommendation features, the business simply stated, "We want personalization." While a rather succinct and efficient means of describing a general desire for the company to provide a modern personalized experience, this blanket request is entirely too vague.

The business, in their mental map of what personalization is, has a great deal of specific and detailed assumptions that define what they are expecting. They possess a deep knowledge of the rules that their products have, how their inventory management system works, and process information regarding contractual agreements that they have with the makers of the products that they sell that dictate placement of certain brands on the website and app. Carried along with these specific business rules, they have general expectations of how they want the products to look. There is, to put it simply, and inherent bias in their expectations.

For the other team at that initial meeting (the ML team), they were simple doing as nerds do: spending their time thinking about how to solve the root of the problem, buried in thoughts about algorithms, APIs, and requisite data for those algorithms. They were focused mostly on the technical details around generating a recommendation list for each user, imagining the chance to try out new technologies and solutions that they haven't had a reason to use before. Their excitement and problem-solving focus from a modeling perspective made them blind to the nuances of business rules that may govern how their solution should work. They weren't asking the right questions to the business.

The first elemental issue that happened in this failed project was that the basic details of expectations were not communicated. Figure 2.5 below shows the divergence in expectations from the business side and the ML side. Since so little was discussed about what the plan was, there was very little overlap between what the ML team built (they did build a recommendation engine, after all) and what was expected by the business (a recommendation engine that could generate personalized results *for this company*).

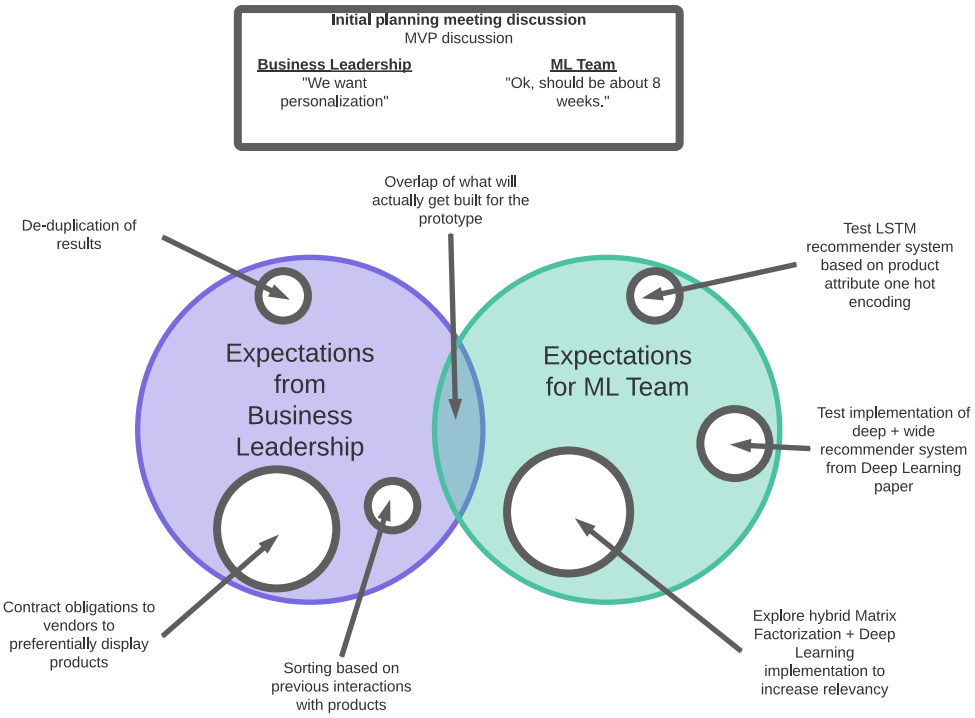


Figure 3.5 A visual representation of the first main issue with the planning meeting. The internal customer group (the business leadership team) communicates, in vague terms, what they want. Their assumptions of how the prototype should function and the critical features are not communicated to the ML team. Likewise, the ML team is focusing on what they know: the algorithmic-focused aspects of testing. It's little wonder why, based on the lack of planning out the important aspects of the project, the MVP is considered a failure.

Figure 3.5 sums up the planning process for the MVP. With extremely vague requirements, a complete lack of thorough communication about what is expected as minimum functionality from the prototype, and a failure to reign in the complexity of experimentation, the demonstration is considered an absolute failure. Preventing this level of annoyance from the business can be achieved only in these early meetings when the project's ideas are being discussed.

Why are you so mad, though?

There is a direct correlation between the depth of discussion in early planning phases (collecting details of how the business expects something to work) and the level of frustration that is felt when a prototype is shown.

Due to the complexity of ML, many business teams don't want to get involved in the gritty details of implementation (nor should they) and as a result the conversations are generally very vague. The ML teams facing this interaction, if not pushing for detailed explanations of expectations and requirements, will do as they would do when doing pure research: test all the things, make sure that the algorithmic purity of the solution can be measured

as well as it can be, and put little regard into the details that they are wholly ignorant of (since they're data and ML experts, not business experts in the domains that they're building solutions for).

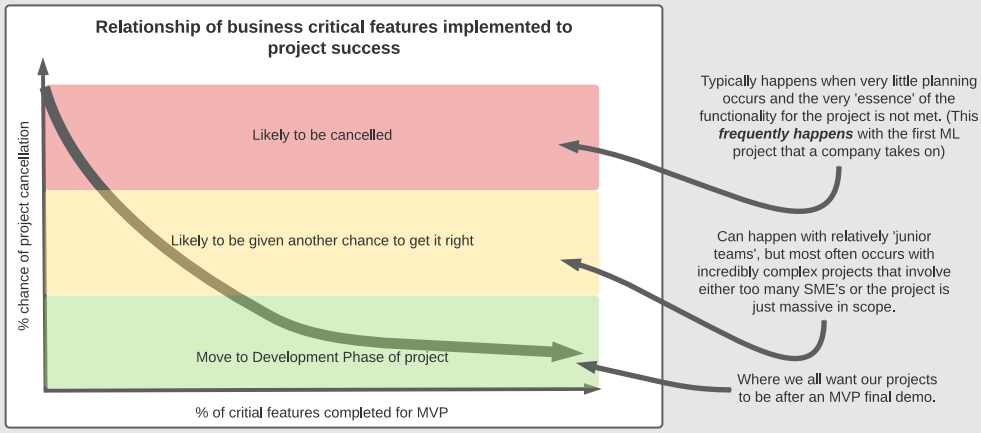


Figure 3.6 The relationship of prototype critical feature coverage (whether defined during planning or not) to the chance of the internal customer team bailing from the project. It is worth noting, with extreme emphasis, that the total sum of critical features is not guaranteed (or expected) to be known even with the most thorough of planning meetings and discussions. Knowing this fact can allow an ML team to ask even more questions during the exploratory discussions with internal teams. The more that you know about what is expected, the more time and effort can be allotted to ensure that these critical aspects of the project are met. Knowing these aspects of the project can certainly be the difference between building out a full solution and packing up the project.

Figure 3.6 shows the relationship that I've experienced while working with teams that are building prototype minimum viable product (MVP) solutions to a problem (as well as ones that I've presented as well). The fewer of these critical but nuanced requirements that are known before writing code and building a solution, the greater the chances that everyone is going to be absolutely pissed-off, increasing the chances that the project will be abandoned with little to show of it apart from some example scripts of half-baked implementations and a gigantic cost in time, salary, and computing costs. The greater the level of planning discussions that are had, the more chances that more of these critical features will be identified so that the subjective estimation of the quality of the solution will be higher, leaving the team to move forward in building out the full project solution.

In the scenario mentioned before (the unfortunate dumpster fire), the utter lack of communication and assumptions that were made were the root of all of the complications in the delivery of the experimental demo. Instead of openly discussing the expectations, the two groups had expected priorities in their minds of what they assumed the other group would be thinking of as priorities.

Let's take a look at breaking down where the planning phases went wrong through a hindsight exercise of this simulated failed planning meeting, where not even the basic expectations of functionality were discussed and some of the key aspects of why they went wrong. Figure 2.7 below shows the inevitable discussions that would happen as a result of so many of the assumptions detailed in figure 3.4 not being scoped into the MVP.

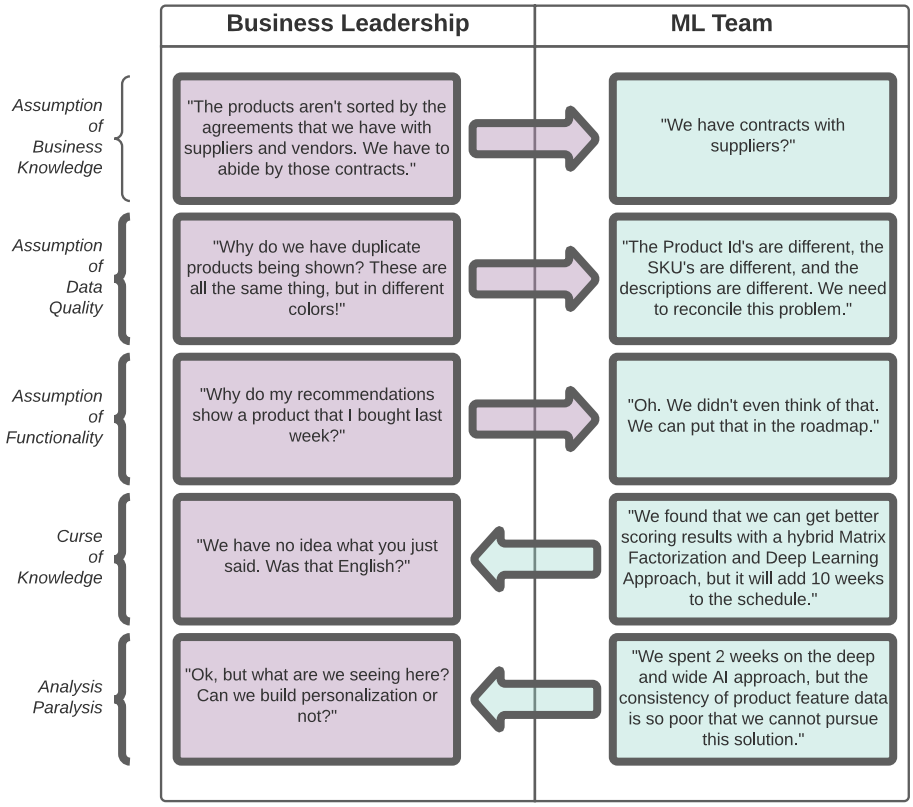


Figure 3.7 A simulated disastrous questions and comments session between the business and the ML team during a demo for a poorly planned project. While in hindsight, all of these elements can seem quite obvious to everyone in the meeting, nearly all of them are the result of understandable ignorance due to the fact that the larger team didn't properly discuss what needed to be built and how it needed to be built.

Although the example in this case is intentionally hyperbolic in nature, I've found that there are elements of this confusion present in nearly all ML projects (those outside of primarily ML-focused companies), and this is to be expected. The problems that ML frequently intends to solve are quite complex, full of details that are specific and unique to each business (and business unit within a company), and fraught with disinformation surrounding the minute nuances of said details. What is important is to realize that these struggles are going to be an inevitable part of any project and that the best way to minimize their impact is to have a very thorough series of discussions that aim to capture as many details about the problem, the data, and the expectations of the outcome as possible.

ASSUMPTION OF BUSINESS KNOWLEDGE

On the business leadership side, that 'side of the table' assumed, since it is considered within their team as such a critical portion of the business that vendor contracts dictate the presentation of certain products on their platform, that the ML team would factor in the ability to enforce selective ordering based on the business needs. In reality, the ML team was wholly ignorant of these needs. To them, just as to the customers, the platform simply looks like a web store where the seemingly arbitrary positioning of products appear to do nothing more than promote popular trends, when the truth of the matter couldn't be further from the truth.

An assumption of business knowledge is, many times, a dangerous path to tread for most companies. In the vast majority of organizations, the ML practitioners are insulated from the inner workings of a business. With their focus mostly in the realm of providing advanced analytics, predictive modeling, automation tooling, there is scant time to devote to understanding the nuances of how and why a business is run in the way that it is run. While some rather obvious aspects of the business are known by all (i.e. "we sell product 'x' on our website"), there is no reasonable expectation that the modelers should know that there is a business process around the fact that some suppliers of goods would be promoted on the site over others.

A good solution for arriving at these nuanced details is by having an SME from the group that is requesting a solution be built for them (in this case, people from the product marketing group) explain how they decide the ordering of products on each page of the website and app. Going through this exercise would allow for everyone in the room to understand the specific rules that may be applied to govern the output of a model.

ASSUMPTION OF DATA QUALITY

The onus of the issue of duplicate product listings in the output of the demo is not entirely on either team. While the ML team could certainly have planned for this to be an issue, they weren't aware of it precisely in the scope of its impact. Even had they known, they likely would have wisely mentioned that correcting for this issue would not be a part of the demo phase (due to the volume of work involved in correcting for that and their request that the prototype not be delayed for too long).

The principle issue here is in not planning for it. By not discussing the expectations, confidence erosion happens in the business for the capabilities of the ML team, and as such, the objective measure of the success of the prototype will largely be ignored as the business members focus solely on the fact that for a few users sample data, the first 300 recommendations show nothing but 4 products in 80 different available shades and patterns.

Figure 3.8 below shows a slight modification to the very old tribal-knowledge reference of what a Data Scientist spends their time on. The Pareto Principle (or the 80/20 rule), although not based in any way on reality, is a tongue-in-cheek reference to the fact that we, as a profession, spend a great deal of time messing around with just getting the data ready for analytics and modeling. On the left is this unofficial 'law' of Data Science, while on the right is what I've seen in industry as a measure of the percentage of companies that have truly immaculate data that is completely ready for encoding and vectorization of data to directly go into a model training run.

For our use case here, the ML team believed that the data they were using was, as told to them by the Data Engineering (DE) team, quite clean. They believed that they were in the 1% in figure 3.8. Every product had a unique primary key that had been meticulously generated whenever a new product had been entered into their system. From a DE perspective, the tables were immaculate. However, what they failed to check, was the quality of the source data coming in to get those unique primary keys.

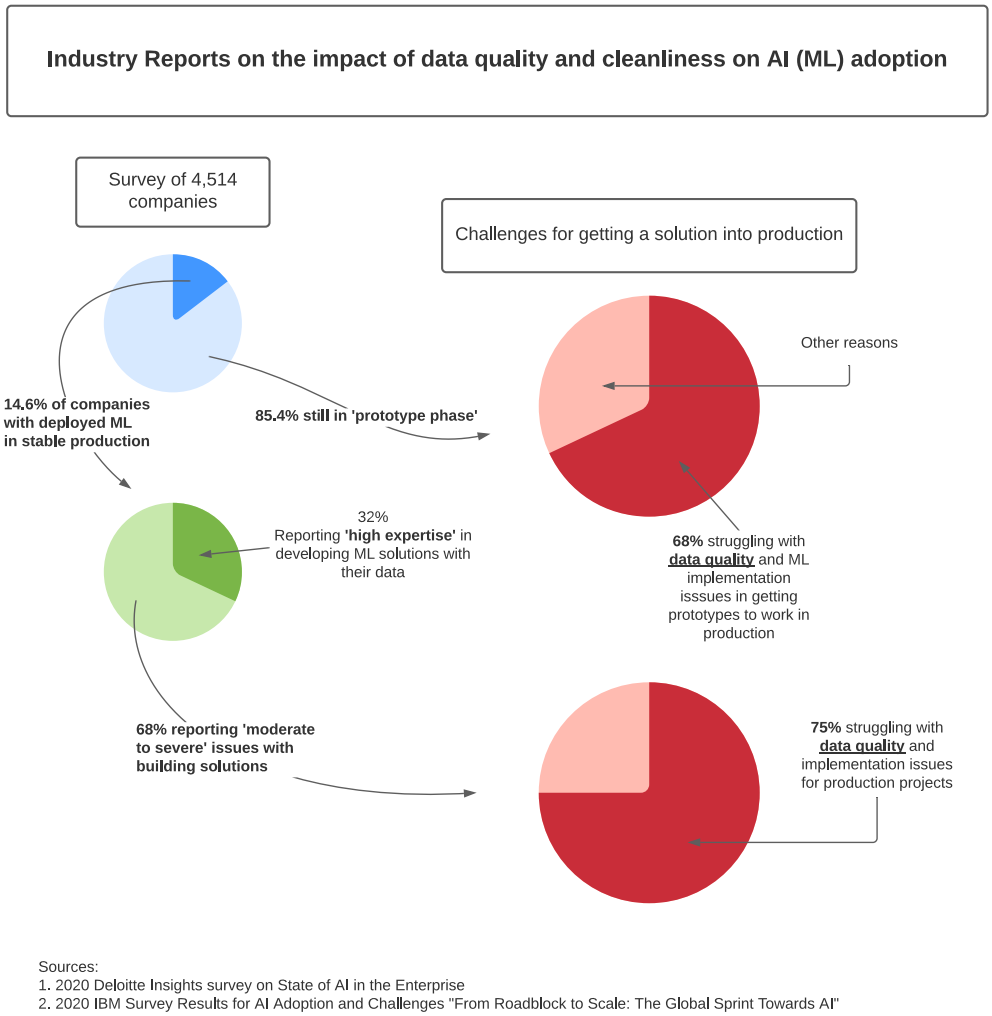


Figure 3.8 An analysis of the struggles that companies are having in realizing the benefits of ML as of 2020. Of particular interest is in the disparity of the effects of data quality issues and implementation difficulties with ML projects between the prototype phase companies and the production companies. Both have data quality

as a significant hurdle to creating production ML solutions, but the proportion of data problems for companies with successfully deployed solutions is actually higher than in the prototype phase companies. Nearly every company faces these issues. It's important to realize that even seemingly sacrosanct data is likely not going to be as clean as it needs to be for predictive modeling. Having a process in place of validating data in the earliest stages of a project (both statistically and visually) can save an otherwise doomed project.

It's not important to have 'perfect data'. Even amongst the companies in figure 3.8 above that are successful in deploying many ML models to production, they still struggle (75% as reported) with data quality issues regularly. These problems with data are just a byproduct of the frequently incredibly complex systems that are generating the data, years (if not decades) of technical debt, and the expense associated with designing systems that do not generate data that has issues with it. The proper way to handle these known problems is to anticipate them, validate the data that will be involved in any project before modeling begins, and to ask questions about the nature of the data to the SME's who are most familiar with it.

Validating the data and ensuring that the minimum expectation for each project has in its early stages a thorough statistical evaluation as well as sampled 'manual' inspection of the data can truly help to ensure that crippling errors like the one presented here (showing identical products on a demo) are prevented as much as possible.

For the case of this recommendation engine, the ML team failed to not only ask questions about the nature of the data that they were modeling (namely, "do all products gets registered in our systems in the same way?"), but also failed to validate the data through analysis. Pulling some rather quick statistical reports may have uncovered this issue quite clearly, particularly if the unique product count of shoes was orders of magnitude higher than any other category. "Why do we sell so many shoes?", posed during a planning meeting, could have instantly uncovered the need for resolution of this issue with the shoes, but also a deeper inspection and validation of all product categories to ensure that the data going into the models was correct.

ASSUMPTION OF FUNCTIONALITY

In this instance, the business leaders are annoyed that the recommendations show a product that was purchased the week before. Regardless of what the product may be (consumable or not), the planning failure here is in expressing how off-putting this would be to the end-user to see this happen. The ML team's response of ensuring that this key element needs to be a part of the final product comes with a degree of politicking; in reality, the ML team would be annoyed at what they would see as a large scope creep (more on scope creep and its productivity-killing nature in chapter 4) item. Forcing an entirely new solution to be developed (that is, if the business actually decides to continue with the project after this disastrous showing) with only so much leeway that could be used to accommodate the new work is only going to put additional pressure on the team to deliver on time.

CURSE OF KNOWLEDGE

The ML team, in this discussion point, instantly went 'full nerd'. We will discuss this at length in chapter 4, but for now, realize that when communicating the inner details of things that have been tested will always fall on deaf ears. Assuming that everyone in a room understands the finer details of a solution as anything but a random collection of pseudo-

scientific buzz-word babble is doing a disservice to yourself as an ML practitioner (you won't get your point across) and to the audience (they will feel ignorant and stupid, frustrated that you assume that they would know such a specific topic).

The better way to discuss the fact that you have tried a number of solutions that didn't pan out: simply speak in as abstract terms as possible. "We tried a few approaches, one of which might make the recommendations much better, but it will add a few months to our timeline. What would you like to do?" would work much better. If your audience is interested in more in-depth technical discussion, then gradually ease into deeper technical depth until the question is answered. It's never a good idea to buffalo your way through an explanation by speaking in terms that there is in no way a reasonable expectation for them to understand.

ANALYSIS PARALYSIS

Without proper planning, the ML team will likely just experiment on a lot of different approaches, likely the most state of the art that they can find in the pursuit of providing the best possible recommendations possible. Without focus on the important aspects of the solution during the planning phase, this chaotic approach of working solely on the 'model purity' can lead to a solution that misses the point of the entire project.

After all, sometimes the most accurate model isn't the best solution. Most of the time, a good solution is one that incorporates the needs of the project; and that generally means keeping the solution as simple as possible to meet those needs. Approaching project work with that in mind will help to alleviate the indecisions and complexity surrounding complex alternatives of 'the best solution' from a modeling perspective.

3.1.2 That first meeting

As we discussed earlier in our simulation about starting the recommendation engine project in the worst possible way, we found a number of problems with how the team approached planning. How did they get to that state of failing to communicate what the project should focus on, though?

While everyone on the ML team was quietly thinking about algorithms, implementation details, and where to get the data to feed into the model, they were too consumed to ask the questions that should have been posited. No one was asking details about how it should work, what types of restrictions need to be in place on the recommendations, or if there should be a consideration to how the products are displayed within a sorted ranked collection.

Conversely, the internal marketing team that was bringing the project to the ML team were not discussing their expectations clearly. With no malicious intent, their ignorance of the methodology of developing this solution coupled with their intense knowledge of the customer and how they want the solution to behave created a perfect recipe for a perfect implementation disaster.

How could this have been handled differently? How could that first discussion have been orchestrated to ensure that the most amount of hidden expectations (as we discussed in section 3.1.1) that the business unit team members hold can be openly discussed in the most productive way? It can be as easy as starting with a single question: "What do you do now to decide what products to display in what places?". In figure 3.9 below, let's look at

what posing that question may have revealed and how it could have informed the critical feature requirements that should have been scoped for the MVP.

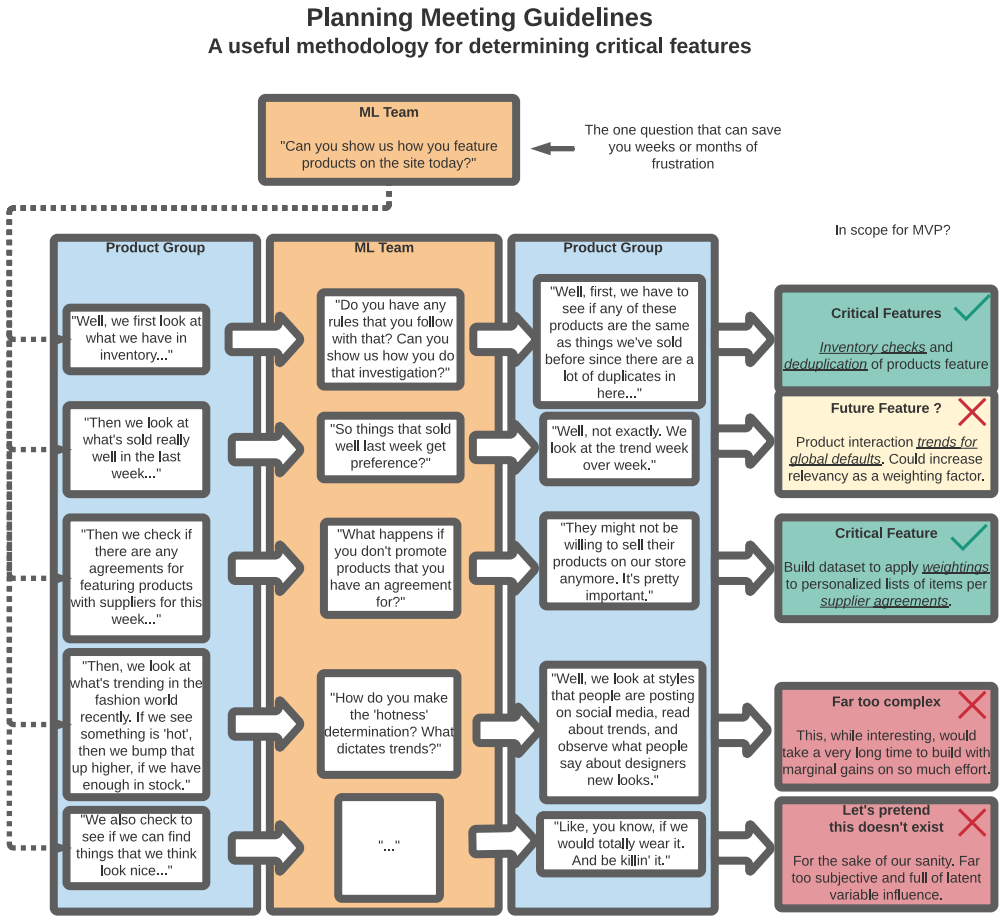


Figure 3.9 An example planning meeting discussion between the ML team and a business unit for the recommendation engine project. Keeping the discussion focused on how people do the job that you are working to augment or replace can help to identify nuances for the project that are just as critical to the solution's success as anything involving the algorithm or feature engineering work. Discovering requirements that are critical, as well as ones that may be slated for incorporation later can help prevent unexpected problems that would otherwise derail a project.

As figure 3.9 shows, not every idea that comes from a discussion is a great one (note: this is not an exaggeration. This is a transposition of an actual planning meeting that I was involved in). Sometimes you may uncover processes that are so subjective that they defy the ability to implement with non-sentient AI (as in the case above, with the final block "we pick

things that are cute") or objective measurements that would require months or years to implement (the "we look at trends and visually see similar styles that people are talking about" mention). Sometimes the ideas are mundane, while other times they are bordering on truly enlightened thought. The point of the exercise is to both uncover the critical details that might not have been discovered until much later on (e.g. the duplicate items) when it's too late to fix, or the items that could inform approaches that are worthy of testing and potentially including in the final build of the solution.

The one thing to make sure to avoid in these discussions is speaking about the ML solution. Keep notes so that the nerds can discuss later; you don't want to drag the discussion away from the primary point (gaining insight into how the business solves the problem currently) just so that you can start thinking of approaches to emulate what they're doing in code.

One of the easiest ways to approach this subject is, as shown in the callout below, by asking how the SME's that currently do (or interact with the data supporting this functionality) their jobs. This methodology is precisely what informed the line of questioning and discussion in figure 3.9.

Explain how you do it so I can help automate this for you

Although not every piece of ML is a direct replacement for boring, error-prone, or repetitive work done by humans, I've found that the overwhelmingly vast majority of it is. Most of these solutions that are worked on are either being done to replace this manual work, or, at the very least, do a more comprehensive job at what people have been attempting to do without the aid of algorithms.

In the case of this recommendation engine we've been discussing, the business had been attempting to work on personalization; it was just personalization by way of attempting to appeal to as many people (or themselves) as much as they could when selecting products for prominent feature and display. This applies to ML projects as far ranging from supply chain optimization to sales forecasts. At the root of most projects that will come your way, there is likely someone at the company that is making their best effort to accomplish the same thing (albeit without the benefit of an algorithm that can sift through billions of data points and draw an optimized solution from relationships that are far too complex for our minds to recognize in an acceptable amount of time).

I've always found it best to find those people and ask them, "teach me how you do it now, please." It's truly staggering how a few hours of listening to someone who has been working through this problem can eliminate wasted work and rework later on. Their wealth of knowledge about the task that you're going to be modeling and the overall requirements for the solution will help to not only get a more accurate project scoping assessment, but also help to ensure that you're building the right thing.

The appropriate usage of an ideation discussion is in talking. It is in drawing pictures, explaining how something should work as an ideal end-state. Instead of thinking about the algorithmic solutions to build the required inference to solve the problem, the primary focus of ideation and planning is to start at the solution side, then work backwards through functionality to ensure that the critical aspects of the project that will mean most to the internal (and external) customers are met. Only after these details are ironed out, then a plan for model implementation experimentation can be developed.

We'll be discussing the process of planning and examples on setting periodic ideation meetings later in this chapter in much more depth.

3.1.3 Plan for demos. Lots of demos.

Yet another cardinal sin that the ML team violated in their presentation of their personalization solution to the business was in the fact of them attempting to show the MVP only once. Perhaps their sprint cadence wasn't such that they could generate a build of the model's predictions at times that were convenient, or they didn't want to introduce slow-down into their progress towards having a true MVP to show to the business. Whatever the reason may be, the team actually wasted time and effort while trying to save time and effort. They were clearly in the top portion of figure 3.10 below.

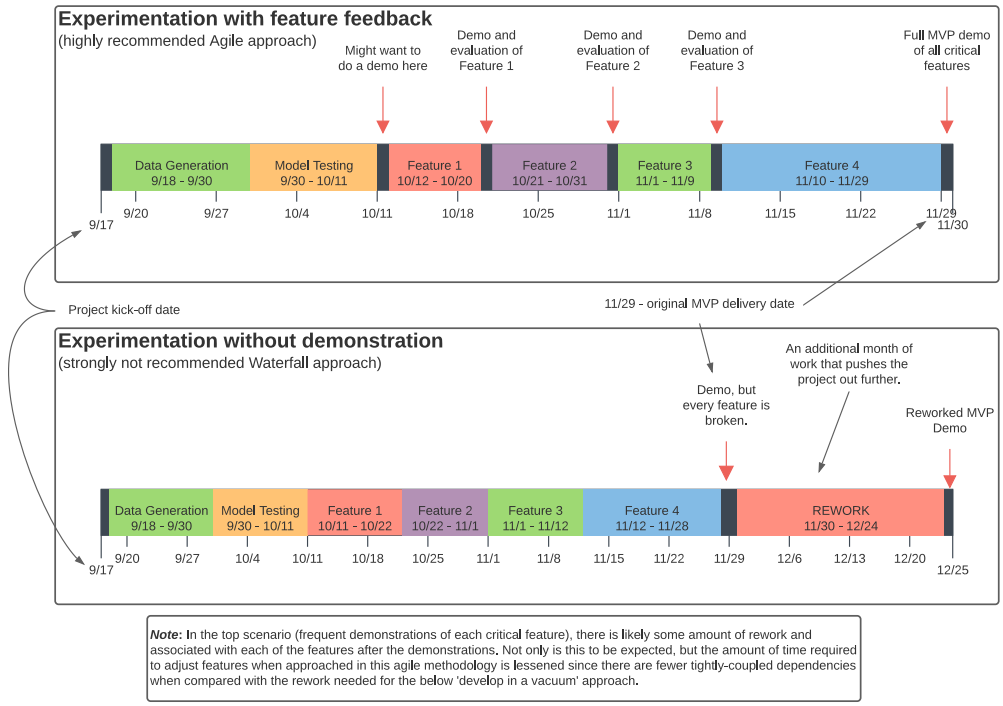


Figure 3.10 The two opposing schools of thought with ML development: demonstration or operating within a silo. The main risk with silo-development (top) is a heightened and almost inevitable risk of large amounts of rework. Sticking with an Agile approach to feature development, even during experimentation and prototype-building will help to ensure that the features that have been added actually meet the requirements of the SME team.

Even though Agile practices were used within the ML team (well, sort of… they did call their work 'sprints' and they had scrum meetings), to the marketing team, this seemed as though they would get a solution provided to them in 2 months of a fully functional pane on

the website and app that would serve individualized personalized recommendations. At no point in those 2 months did a meeting take place to show the current state of experimentation, nor was there a plan that was communicated about the cadence of seeing results from the modeling efforts. As figure 3.10 above shows, the top portion of (to an external viewer) waterfall development and release will only result in a massive period of rework (notice the different time scales in the figure above).

Without frequent demos as features are built out, the team at large is simply relying on the word of the ML team, and the ML team, not having SME members who are developing the features, is relying on their memories and the notes that they took when asking questions in the planning meetings. There are, for most projects involving ML of sufficient size, far too many details and nuances to confidently approach building out dozens of features without having them reviewed. Even if the ML team is showing metrics for the quality of the predictions, aggregate ranking statistics that 'conclusively prove' the power and quality of what they're building, the only people in the room that care about that are the ML team. In order to effectively produce a complex project, the subject matter expert (SME) group – the marketing group – needs to provide feedback based on data that they can consume. Presenting arbitrary or complex metrics to that team is bordering on intentional obtuse obfuscation, which will only hinder the project and stifle the critical ideas that are required to make the project successful.

By planning for demos ahead of time, at particular cadences, the ML-internal agile development process can adapt to the needs of the business experts to create a more relevant and successful project. They can embrace a true Agile approach: of testing and demonstrating features as they are built, adapting their future work and adjusting elements in a highly efficient manner. They can help to ensure the project will actually see the light of day.

But I don't know front-end development. How can I build a demo?

There's a phrase I've used before.

If you do happen to know how to build interactive lightweight apps that can host your ML-backed demos, that's awesome. Use those skills. Just don't spend too much of your time building that portion. Keep it as simple as possible and focus your energy and time on the ML problem at hand.

For the other 99% of ML practitioners out there, you don't need to mockup a website, app, or microservice to show content. If you can make slides (notice I'm not asking if you want to – we all know that all of us hate making slides), then you an display representations of how your project will work by displaying a simulation of what something will look like to the end-user.

If you communicate the fact clearly that the final design of what the UX team and front-end developers or application designers will be completely different than what you're showing; that you're just here to show the data, then something as simple as a slide deck or .pdf of a layperson-friendly layout will work just fine. I can promise you that converting arrays of primary keys or matplotlib area under ROC curves into something that tells the story of how the model performs in a digestible way will always go over better in meetings involving non-technical audiences.

3.1.4 Experimentation by solution building: wasting time for pride's sake

Looking back that the unfortunate scenario of the ML team building a prototype recommendation engine for the website personalization project, their process of experimentation was troubling, but not only for the business.

Without a solid plan in place about what they would be trying and how much time and effort they would be placing on the different solutions they agreed on pursuing, a great deal of time (and code) was unnecessarily thrown away.

Coming out of their initial meeting, they went off on their own as a team, beginning their siloed ideation meeting with a brainstorming about which algorithms might best be suited for generating recommendations in an implicit manner. 300 or so web searches later, they came up with a basic plan of doing a head-to-head comparison of 3 main approaches: an ALS model, an SVD model, and a Deep Learning recommendation model. Having an understanding of the features that were required to meet the minimum requirements for the project, 3 separate groups began building what they could in a good-natured competition.

The biggest flaw in approaching experimentation in this way is in the sheer scope and size of the waste involved in doing bake-offs like this. Approaching a complex problem by way of a hackathon-like methodology might seem fun to some, not to mention being far easier to manage from a process perspective by the team lead (you're all on your own – whoever wins, we go with that!), but it's an incredibly irresponsible way to develop software.

This flawed concept, solution building during experimentation, is juxtaposed with the far more efficient (but, some would argue 'less fun') approach of prototype experimentation in figure 3.11 below. With periodic demos (either internally to the ML team or to the broader external cross-functional team), the project's experimentation phase can be optimized to have more hands (and minds) focused on getting the project as successful as it can be as fast as is possible.

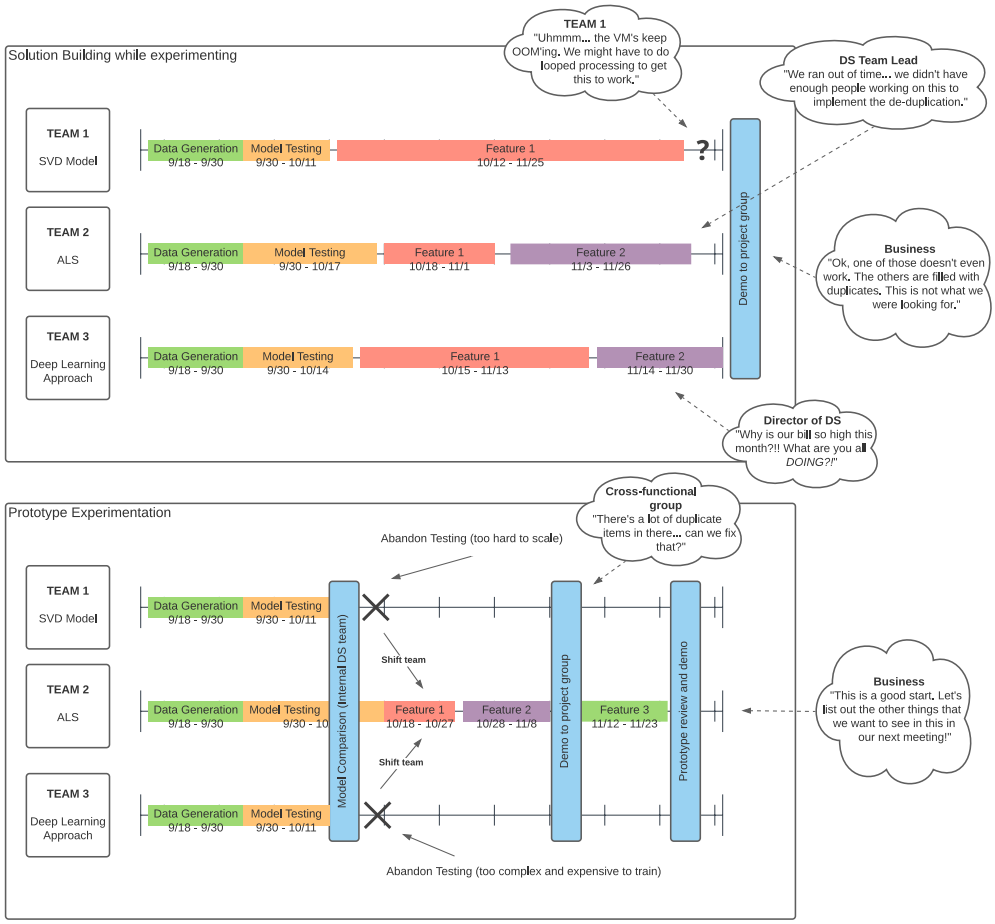


Figure 3.11 Juxtaposition of the 'sin' of solution building during experimentation phase and a proper prototype building through experimentation. Since there is always a finite resource pool of humans to develop solutions, the faster that a judgement is made regarding the implementation path, the more resources can be allocated to actually building out the prototype. With human effort always being the universal denominator in efficiency, the less work that the team does, the more focused that a team (or individual) can be, the better it is for the chances of project success.

The final critical failure in the recommendation engine scenario above was in the assumption by the ML team that they needed to build out full MVP solutions for each of the implementations that they wanted to test. By delaying the presentation of the model outputs to the business team, they were effectively forcing themselves to figure out all 3 approaches in the span of time that only 1 could realistically be accomplished with the resources that they had What they were engaging in, as shown in figure 3.11 above, was Solution Building while Experimenting.

As shown in the top section of the figure, approaching the problem of a model bake-off without planning for prototype culling runs two primary risks. Firstly, in the top portion, Model A was found to be very challenging to incorporate the first primary feature that the business dictated was critical. Since no evaluation was done after the initial formulation of getting the model to work, a great deal of time was spent by the sub-team in the ML group in trying to get the feature built out to support the requirements. After that was accomplished, when moving on to the 2nd most critical feature, the team members realized that they actually couldn't implement the feature in enough time for the demo meeting, effectively guaranteeing that all of the work that was put into model A is going to be thrown away. The other two approaches both, due to being short-staffed on the implementation of their prototypes, were unable to complete the third critical feature. As a result, none of the three approaches would satisfy the requirements and will lead to frustration on the part of the larger project team and delays to the project. What the team should have done instead was follow the path of the bottom section 'Prototype Experimentation'. In this approach, the teams met with the business units early, communicating ahead of time that the critical features wouldn't be in at this time, but that they wanted to make a decision on the raw output of each model type that was under test. After the decision from this meeting being made to focus on a single option, the entire ML team's resources and time could be focused on implementing the minimum required features (with an added check-in demo in between the presentation of the core solution to ensure that they were on the right track) and get to the prototype evaluation sooner.

This habit of building too much with insufficient requirements and details is remarkably common in the ML world, as practitioners typically put upon themselves an overwhelming burden of wishing to show functional results. Sometimes this pressure is placed on the team due to ignorance ("we asked the web team to make this feature look this way and it worked just fine and they delivered it in 2 weeks. Why is this thing you built so broken after taking so long to build it?"), and other times it is self-imposed.

To alleviate all of this pressure, and plan properly for a state of the solution to be presented that is merely a 'work in progress', it's as easy as defining this to the business. Let them know what the ML development cycle is like. Educate them on how many failures you're going to be going through before you find a solution that works. Assuage their fears that if they see hot garbage during a demo, that it can be adjusted and improved. If you fail to let them know about what the process is like, the loss of faith can be irreparable to the point of project cancellation. Whether it be pride, ego, ignorance, or fear, whatever motivates the team to not discuss these items should be addressed and overcome if you ever want your project to actually see the light of day.

As part of the planning phase and initial ideation discussions, it is the responsibility of the ML team to clearly communicate a process that their work will follow *and stick to it*. The concepts of research, prototyping, developmental milestones, and feedback collection are, when taken together, all aspects that set ML projects apart from other software development, and as such, this needs to be clearly communicated to the broader team. In the process of building a solution, there will be unknowns. At the start, there will be unknown unknowns. If everyone is not aware of this and prepared with plans to mitigate the more

disastrous wholly unknown issues, the project can be quickly deemed a failure and abandoned.

3.2 Experimental Scoping: Setting expectations and boundaries

We've now been through planning of the recommendation engine. We have the details of what is important to the business, we understand what the user expectation of behavior should be when interacting with our recommendations, and we have a solid plan for the milestones of what we'll be presenting at what date throughout the project. Now it's time for the fun part for most of us ML nerds. It's time to plan our research. With an effectively limitless source of information at our fingertips on the topic, and only so much time to do it in, we really should be setting some guidelines on what we're going to be testing and how we're going to go about it. This is where scoping of experimentation comes into play.

The team should, by this time, after having had the appropriate discovery sessions with the SME team members, know the critical features that need to get built:

- We need a way to de-duplicate our product inventory
- We need to incorporate product-based rules to weight implicit preferences per user
- We need to group recommendations based on product category, brand, and specific page types in order to fulfill different structured elements on the site and app
- We need an algorithm that will generate user-to-item affinities that won't cost a fortune to run

After listing out the absolutely critical aspects for the MVP, the team can begin planning out what work is estimated to be involved in solving each of these 4 critical tasks. Through setting these expectations and providing boundaries on each of them (for both time and level of implementation complexity), the ML team can provide the one thing that the business is seeking: *an expected delivery date and a judgement call on what is or isn't feasible*.

This may seem a bit oxy-moronic to some. "Isn't experimentation where we figure out how to scope the project from an ML perspective?" is likely the very thought that is coursing through your head right now. We'll discuss throughout in this section why, if left without boundaries, the research and experimentation on solving this recommendation engine problem could easily fill the entire project scoping timeline. If we plan and scope our experimentation, we'll be able to focus on finding, perhaps not the 'best solution', but hopefully a 'good enough solution' to ensure that we'll eventually get a product built out of our work.

Once the initial planning phase is complete (which certainly will not happen from just a single meeting) and a rough idea of what the project entails is both formulated and documented, there should be no talk about scoping or estimating how long the actual solution implementation will take, at least not initially. Scoping is incredibly important and is one of the primary means of setting expectations for a project team as a whole, but even more critical for the ML team. However, in the world of ML (which is very different from other types of software development due to the nature of the complexity of most solutions), there are actually two very distinct scoping's that need to happen. For people who are accustomed to interactions with other development teams, the idea of 'experimental scoping' is completely foreign, and as such any estimations for the initial phase scoping will be

misinterpreted. With this in mind, however, it's certainly not wise to not have an internal target scoping for experimentation.

WHAT IS EXPERIMENTAL SCOPING?

Before you can begin to estimate how long a project is going to take, you need to do some research (I mean, it's recommended – sort of a 'what puts the science in Data Science' thing). If the project is a revenue forecast model based on basic seasonal trends, you might not need to do much research (there are entire packages that automate nearly all of the complexity of that for you with a few lines of high-level API calls). However, for the project that we've been discussing in this chapter, a complex recommendation engine, the difficulty of the problem should be directly proportional to the amount of research that will be done.

Based on the initial user-centric design that was created in section 3.1.1, there are a great many moving pieces that need to be investigated. Not relegated to the realm of 'high level APIs' is this project.

The ML team will be reviewing white papers, reading the latest research about what others have built in this space, and evaluating not only algorithms, but platforms that are capable of handling the monumental volume of data that these algorithms need in order to produce useful results.

The front-end team will be researching caching technologies that are elastic and scalable, designing proof-of-concept tests to simulate what the performance impact will be to the website and app.

The data engineering team will be working closely with the ML team, working on generating data sets that are pre-formatted to minimize the amount of feature engineering that would need to be done by the ML team when the project moves to the development phase.

Everyone is busy and there is a lot to uncover for a project of this magnitude.

Someone should really tell the business what's going on and why they're going to have to wait a bit to see some recommendations. In order to have that conversation, it is rather useful to have a plan for experimentation, to set boundaries on what will be researched, attempted, and what risks are present in each of these phases. Once identified, these processes can be codified for each individual team, directing the tasks for the sprint(s) associated with the preliminary work prior to development.

3.2.1 Experimental scoping for the ML team - Research

In the heart of all ML practitioners (at least those still passionate about what they do) is the desire to experiment, explore, and learn new things. With the depth and breadth of all that there is in the ML space, one could spend a lifetime learning only a fraction of what there has been, is currently being, and will be worked on as novel solutions to complex problems. It is because of this innate desire that is shared among all of us that it is of the utmost importance to set boundaries around how long and how far we will go when researching a solution to a new problem.

In the first stages following the planning meetings and general project scoping, it's now time to start doing some actual work. This initial stage, experimentation, can vary quite differently between projects and implementations, but the common theme for the ML team is that it has to be time-boxed. This can feel remarkably frustrating for many PhD graduates

that are heading out to companies for the first corporate work that they've done in their lives. Instead of focusing on researching a novel solution to something from the ground up, they are now thrust into the world of "we don't care how it's build, so long as it's built correctly and on time". A great way to meet that requirement is to set limits on how much time the ML team has to research possibilities for solutions.

For the recommendation engine project that we've been discussing during this chapter, a research path for the ML team might look something like what is shown below, in figure 3.12.

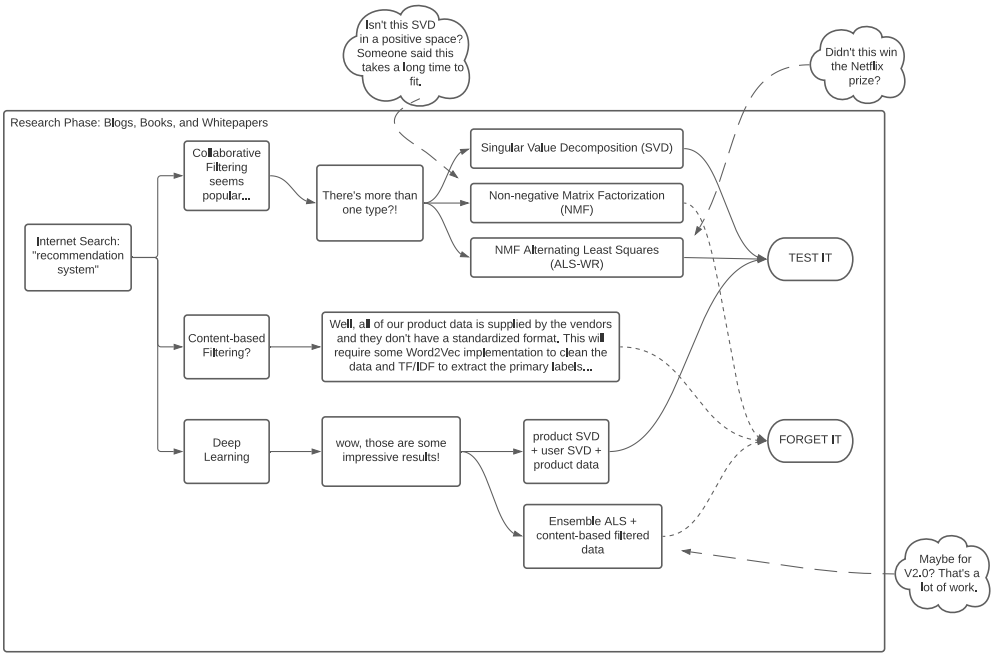


Figure 3.12 Research planning phase diagram for an ML team determining what potential solutions to pursue testing. Defining structured plans such as this can dramatically reduce the time spent iterating on ideas. Having targeted elements of experimentation to test controls the otherwise highly probably research-scope-creep that can result in later rework.

In the simplified diagram of figure 3.12, there are restrictions placed on the branching paths of where research might take the team. After a few cursory internet searches, blog readings, and whitepaper consultations, the team can identify (typically in a day or so) what the "broad strokes" are for existing solutions in industry, as well as the state of current active research. For a search on technical implementations for a recommender system, one might find millions of results. It's not important to read even 0.001% of them. What is important, however, is recognizing the frequently mentioned algorithms and general approaches that are mentioned. Once the common approaches are identified, read a bit more

on those and determine what the most widely used direct applications are. The important thing to keep in mind is: you're solving a business problem; not creating a new algorithm or implementation. For the vast majority of use cases of ML that a business is looking to employ, chances are that many people have done it before. Use that wisdom to your advantage (and if you're moving down the path of a problem that it appears no one has solved before, note that and realize that you're going to be building and owning ***an awfully lot more*** than you would if a package already exists for your use).

As the diagram in figure 3.12 shows, the approaches that are candidates for testing (within the limits of the team's capacity to test based on the number of people and other responsibilities that the team may have) are culled at the first stage. There are many alternate solutions available for not only ensemble hybrid solutions, but also other matrix factorization techniques. After a short period of research these other candidates are dismissed, resulting on a limited number of alternatives to test against one another. Having a limited scope of sufficiently different approaches to test will prevent the chances of either Over-choice (a condition in which making a decision is almost paralyzing to someone due to the over-abundance of options), or suffering the fate of the Tyranny of Small Decisions (in which an accumulation of many small, seemingly insignificant choices that are made in succession can lead to an unfavorable outcome). It is always best, in the interests of both moving a project along and in creating a viable product at the end of the project to limit the scope of experimentation.

The final decision in figure 3.12, based on the team's research, is to focus on 3 separate solutions (one with a complex dependency): ALS, SVD, and a Deep Learning (DL) solution that incorporates the input of the individual aspects of an SVD model. Once these paths have been agreed upon, the team can set out to attempt to build prototypes. Just as with the research phase, the experimentation phase is time-boxed to permit only so much work to be done, ensuring that a measurable result can be produced at the conclusion of the experimentation.

3.2.2 Experiment scoping for the ML team – Experimentation

With a plan in place, the ML team lead is free to assign resources to the prototype solutions. At the outset, it is important to be very clear about what is expected from experimentation. The goal is to produce a simulation of the end product that allows for a non-biased comparison of the solutions being considered. There is no need for the models to be tuned, nor for the code to be written in such a way that it could ever be considered for use in the final project's code base. The name of the game here is a balance between two primary goals: speed and comparability.

There are a great many things to be considered when making a decision on which approach to take, which will be discussed at length through several chapters later in this book, but for the moment, the critical aspect of what this stage is trying to estimate both the performance of the solutions as well as the difficulty of developing the full solution. Estimates for total final code complexity can be created at the conclusion of this phase, thereby informing the larger team what an estimate of development time will be required to produce the project's code base. In addition to the time commitment associated with code complexity, this can also help to inform what the total cost of ownership will be for the

solution; what the daily run cost will be in order to retrain the models, generate inferences of affinity, host the data, and serve the data.

Before setting out planning the work that will be done through the generally accepted best methodology (Agile) by writing stories and tasks, it can be helpful to create a testing plan for the experimentation. This plan, devoid of technical implementation details and the verbose nature of story tickets that will be accomplished throughout the testing phases, can be used to not only inform the sprint planning, but can also be used to track the status of the bake-off that the ML team will be doing. It is something that can be shared and utilized as a communication tool to the larger team, helping to show what it was that was done, what the results were, and can accompany a demo of the two(or more!) competing implementations that are being pursued for options.

Figure 3.13 below shows a staged testing plan for the experimentation phase of the recommendation engine.

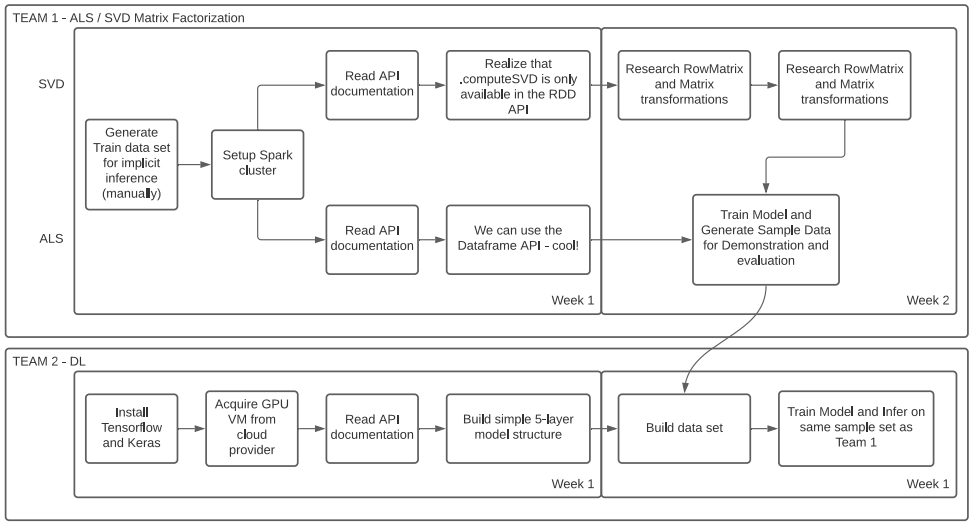


Figure 3.13 An experimentation tracking flow chart. Tools like this serve as inclusive means of showing what the ML team had done at a very high level (omitting the complexity and confusion surrounding implementation details but providing just enough detail to show what is involved and what interrelationships may exist to support the solutions). Here, the DL implementation for the core recommendation engine has an entirely separate model that needs to be generated (SVD) in order to create the vector data to feed into the DL model.

The testing paths clearly show the results of the research phase in figure 3.13. Team 1's matrix factorization approach shows a common data source that needs to be generated (manually for this phase, not through an ETL job), and based on the research and understanding that the team has on the computational complexity of these algorithms (and the sheer size of the data), they've chosen Apache Spark to test out solutions. From this phase, they both had to split their efforts to research the APIs for the two different models,

coming to two very different conclusions. For the ALS implementation, the high-level Dataframe API implementation from SparkML makes the code architecture far simpler than the lower-level RDD-based implementation for SVD. The team can define these complexities during this testing, bringing to light for the larger team that the SVD implementation will be significantly more complex to implement, maintain, tune, and extend.

All of these steps for Team 1 help to define the development scope later on. Should the larger team as a whole decide that SVD is the better solution for their use case, they should weigh the complexity of implementation against the proficiency of the team. If the team isn't familiar with writing a scala implementation that utilizes Breeze, can the project and the team budget time for team members to learn this technology? If the experimentation results are of significantly greater quality than the others being tested (or is a dependency for another better solution), then the larger team needs to be aware of the additional time that will be required to deliver the project.

For Team 2 in figure 3.13, their implementation is significantly more complex, as well as also requiring as input the SVD model's inference. If the results for this are amazing when compared to the other solutions being shown (doubtful), then the team should be scrutinizing a complex solution of this nature. Even if the ML team is adamant about this being the best possible solution, latching onto a solution like this should come with it an edge of skepticism from every team member, particularly if the qualitative and quantitative results of the experimental stage are not obviously substantially better than anything else. 99 out of 100 times when a situation like this comes up, the ML team is pursuing the worst possible development strategy: resume-driven development (RDD). If the goal is to create something to get noticed in the ML community, or to get the chance to speak on stage somewhere to show off how smart they are, then it's highly likely that the long-term success of the project is what is fueling their passion. It's the short-term success and notoriety associated with completing something novel that, although it does solve the problem, is done in such a way that the entire team is hoping they will have a much higher paying and prestigious job landed elsewhere before they ever have to improve or fix the absolute eldritch horror of a code base that they have created. However, if the team is comfortable with the risk and agrees to the added burden in building something of this nature, that's ok. They just need to know what they're getting into and what they'll potentially be maintaining for a few years.

An additional helpful visualization to provide to the larger team when discussing experimental phases is a rough estimate of what the "broad strokes" of the solution will be from an ML perspective. A complex architectural diagram isn't necessary at this point, as it will change so many times during the early stages of development that creating anything of substantial detail is simply a waste of time at this point of the project. However, a high-level diagram, such as the one shown below in figure 3.14 that references our personalization recommendation engine, can help explain to the broader team what needs to be built to satisfy the solution.

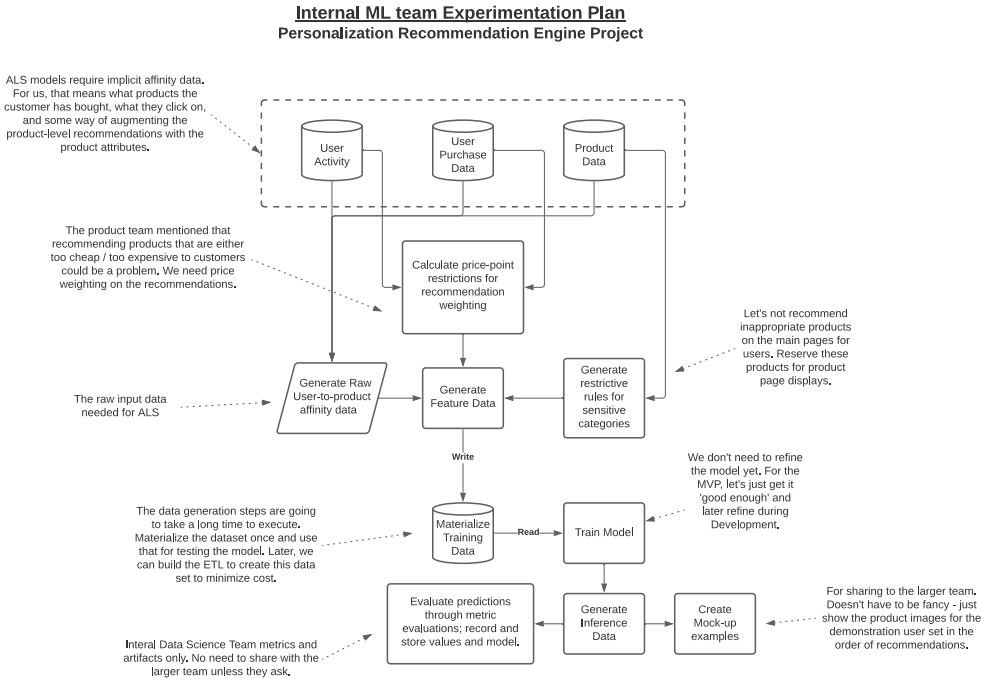


Figure 3.14 A high-level experimental phase architecture that is very helpful for explaining the steps involved in the prototype to the members of the team who don't have an ML background. This becomes a critical diagram that can be used and annotated if a project has an experimental phase that is particularly long due to the complexity of the project. It's far more effective, after all, to say, "We're on the model training phase this week" rather than, "we're attempting to optimize for RMSE through cross validation of hyper parameters and we will evaluate the results after the optimal settings are discovered". As the age-old cliché states, a picture is worth a thousand words.

The annotations in figure 3.14 above can help communicate with the broader team at large. Instead of sitting in a status meeting that could have a dozen or more people in it, working diagrams like the one above can be used by the ML team to communicate through efficient means with everyone. Various explanations can be added to answer any aspects of what the team is working on at a given time to give added information that explains what is being worked on and why, as well as providing context to go along with chronologically-focused delivery status reports (which are inherently difficult for laypersons to visually see the connected nature of the different aspects of a plan presented in that format).

WHY IS SCOPING A RESEARCH (EXPERIMENT) PHASE SO IMPORTANT?

Had the ML team, working on the personalization project, all the time in the world (and an infinite budget), they might have the luxury of finding an optimal solution for their problem. They could sift through hundreds of white papers, read through treatises on the benefits of

one approach over another, and even spend time finding a novel approach to solve the specific use case that their business sees as an 'ideal solution'. Not held back by meeting a release date or keeping their technical costs down, they could quite easily spend months, if not years, just researching the best possible way to introduce personalization to their website and apps.

Instead of just testing two or three approaches that have been proven to work for others in similar industries and use cases, they could work on building prototypes for dozens of approaches and, through careful comparison and adjudication, select the absolutely best approach to create the optimal engine that would provide the finest recommendations to their users. They might even try out a new technical stack that is being hyped in the nerd-blog-o-sphere, or develop their prototype solutions in a new ML framework that just recently released into Open Source (and help be the alpha tester of it!).

If the team were allowed to be free to test whatever they wanted for this personalized recommendation engine, the 'ideas' white board might look something like figure 3.15, below.

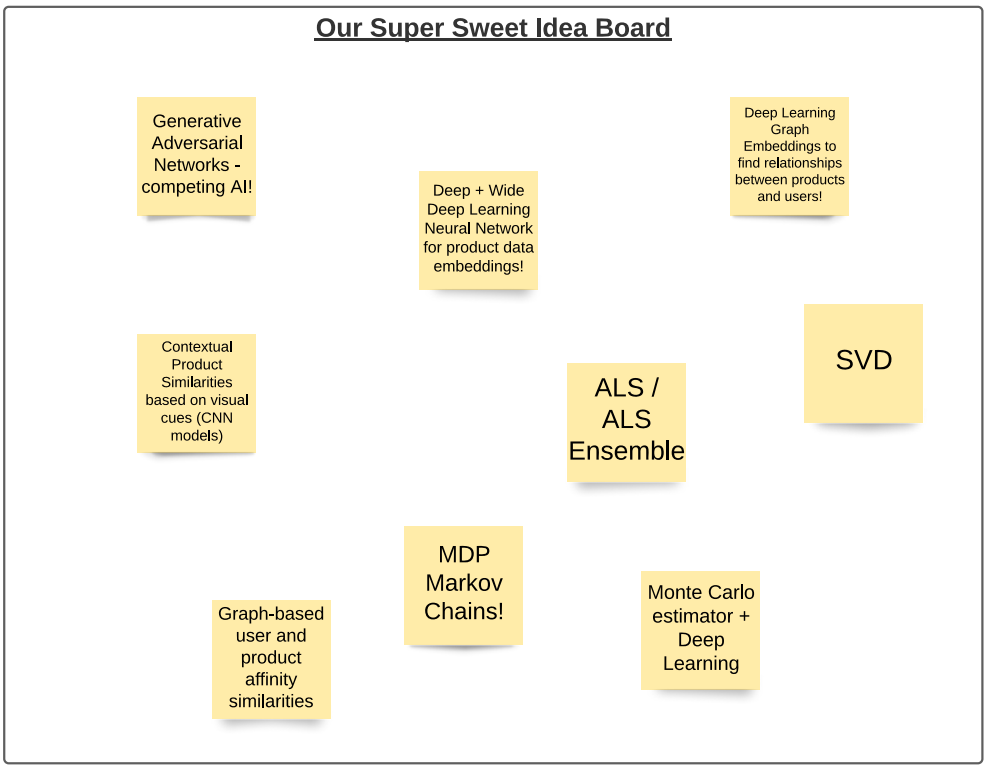


Figure 3.15 The results of ideation without restrictions. Provided a novel problem, the ML team goes a little bit nuts, listing out plausible approaches to solve the business request along with a healthy dose of new research white papers and a bit of science fiction thrown into the mix. None of this is bad, per se, provided that the

team knows that they won't ever deliver the solution if they try to implement all of these.

After a brain-storming session that generated the ideas from figure 3.15 (which bears striking resemblance to many ideation sessions I've had with large and ambitious Data Science teams), the next step that the team should take collectively is to start making estimations of these different implementations. Attaching comments to each of these alternatives can help to formulate a plan of the two or three most likely to succeed within a reasonable time of experimentation. The commentary in figure 3.16, below, can assist the team with making a decision on what to test out to meet the needs of *actually shipping a product to production*.

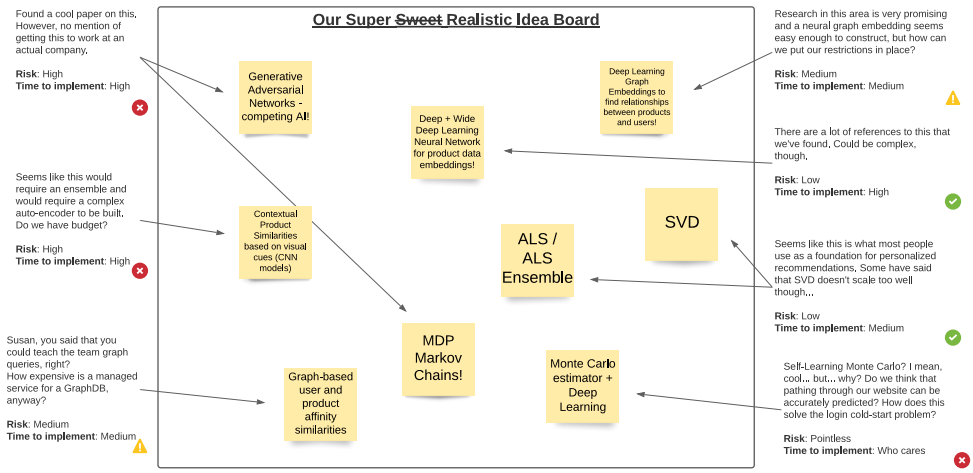


Figure 3.16 The idea board for the personalized recommendation engine project, updated. Each idea is ranked with a severity associated with it for both riskiness of not completing the implementation by the due date for the MVP and an overall guess at how long it will take to implement for the team. (Do keep in mind that this is for demonstrative purposes only. I'm sure that someone will eventually build a recommendation system that uses Monte Carlo simulations with Deep Learning once Quantum Computing becomes mainstream)

After the team goes through the exercise of assigning risk to the different approaches, as shown in figure 3.16, the most likely and least risky options can be decided on that fit within the scope of time that they have allocated for testing.

In a perfect world in which a team of ML professionals had infinite budget, infinite time, and no other pressing responsibilities, scoping would be easy. It would be as simple as stating, "get it to me when you find the perfect solution". Here in the real world, however, companies aren't particularly keen on the idea of waiting too long to let 'nerds be nerds', conducting test after laborious test in the pursuit of the finest application to solve a problem. The main reason for this is clearly outlined in figure 3.17 below, in the top portion of the chart: time.

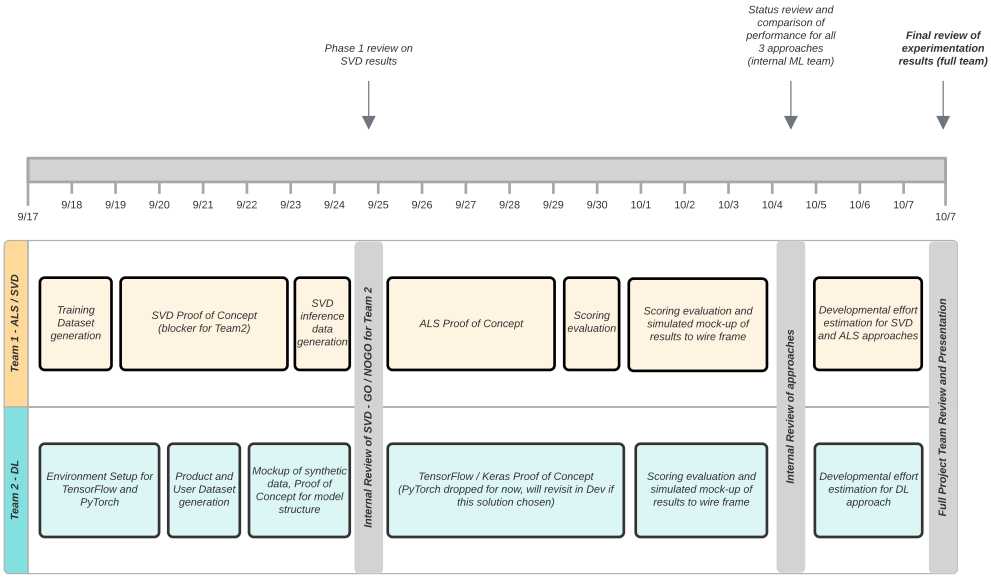


Figure 3.17 Scoped Experimentation for ML teams during experimental phases. Internal and external (larger team) reviews marked as major milestones for scheduled delivery, forcing the team to focus on a minimalistic test for each solution in order to weigh the benefits and drawbacks to each approach, as well as to estimate each solution's complexity in terms of development, maintenance, and total cost of ownership.

Experimentation takes time. Building a proof of concept (PoC) is a grueling effort of learning new APIs, researching new ways of writing code to support the application of a model, and wrangling all of the components together to ensure that at least one run succeeds can take a staggering amount of effort (depending on the problem).

Figure 3.17 shows some (questionably short in reality for a project of this complexity but abbreviated for argument's sake) timelines for experimentation for the recommendation project. Each of these boxes represents a 'best effort' estimation of how much time the team will need to build out the PoC stages of each element. These abstract time boxes are there not as a hard-limit stop imposed on work, but rather as a reminder to the ML team of a few critical parts of this experimentation phase. Firstly, that the longer that times is spent in each of these stages, the more amount of money is going to be spent to make this project a reality (the combined salaries of this team are likely going to be quite substantial in most companies). Secondly, the shortened time is intended to arrive at a decision boundary sooner. Instead of spending weeks wrestling with a poorly documented or just flat-out complex API for a model's implementation, if the team can set a 'soft cap' on how long that they're going to be trying to make it work, they can get a solid estimate on how hard it would be to build out the entire solution. If it takes 2 weeks to get even the first portion of SVD working at-scale on synthetic data, then how long will it take to integrate all of the other requirements into this solution? It may be worthwhile to abandon one such overwhelmingly complex or expensive approach in favor of others. Conversely, if the APIs

are incredibly simple, but the training time is astronomically large, that may inform decisions that need to be made on a cost-benefit analysis to a solution. Perhaps the model is going to be powerful – but will it pay for the cost to run it each day?

Total cost of ownership

While an analysis of the cost to maintain a project of this nature is nigh impossible to estimate accurately at this stage, it is an important aspect to consider.

During the experimentation, there will inevitably be elements that are missing from the overall data architecture of the business. There will likely be services that need to be created for capturing data. There will be serving layers that need to be built. If the organization has never dealt with modeling around matrix factorization, they will need to potentially use a platform that they've never used before.

What if data can't actually be acquired to satisfy the needs of the project, though?

This is the time to identify show-stopping issues. Identify them, ask if there are going to be solutions provided to support the needs of the implementation, and, if not, alert the team about the fact that without investment to create the needed data, the project should be halted.

Provided that there aren't issues as severe as that, some questions to think about during this phase, when gaps and critical issues are discovered, are…

What additional ETL do we need to build?

How often do we need to retrain models and generate inferences?

What platform are we going to use to run these models?

For the platform and infrastructure that we need, are we going to use a managed service, or are we going to try to run it ourselves?

Do we have expertise in running and maintaining services for ML of this nature?

What is the cost of the serving layer plans that we have?

What is the cost of storage and where will the inference data live to support this project?

You don't have to have answers to each of these before development begins (other than the platform-related questions), but they should always be in mind as elements to revisit throughout the development process. If there isn't sufficient budget to actually run one of these engines, then perhaps a different project should be chosen.

We time-block each of these elements for a final reason as well: to move quickly into making a decision for the project. This concept may very well be significantly insulting to most Data Scientists. After all, how could someone adequately gauge the success of an implementation if the models aren't fully tuned? How could someone legitimately claim how well the predictive power of the solution is if all of the components of the solution aren't completed?

I get it. Truly, I do. I've been there, making those same arguments. In hindsight, after having ignored the sage advice that software developers gave me during those early days of my career about why running tests on multiple fronts for long periods of time is a bad thing, I realize what they were trying to communicate to me.

> ***If you had just made a decision earlier, all that work that you spent on the other things would have been put into the final chosen implementation.***

An anecdote on morale

The time blocking isn't intending to force unrealistic expectations on the team; rather, it's there to prevent the team from wasting time and energy on shelf-ware. Restricting time on potential solutions also helps with team morale with respect to such never-to-be-realized implementations (after all, you can only really build one solution for a project). Setting restrictions on how long people can work on a solution to a problem makes it far less painful to throw away what they've worked on if they've only been working on it for a week. If they've been working on it for months, though, it's going to feel rather demoralizing when they're informed that their solution is not going to be used.

One of the most toxic things that can happen when testing out implementation is for tribes to form within a team; each team has spent so much time researching their solution and has been blinded by factors that might make it less than desirable for using as a path to solve the problem. If the experiments are allowed to go from the PoC phase to materialize as a true MVP (which, to be honest, if given enough time, most ML teams will build an MVP instead of a PoC), when it comes to deciding on which implementation to use, there will be tension. Tempers will flare, lunches will disappear from fridges, arguments will break out during standups, and the general morale of the team will suffer. Save your team, save yourself, and make sure that people don't get attached to a PoC.

The final reason as to why time-blocking is so critical to experimentation is particularly applicable if there is no one on the team who has solved a problem of the nature being presented before. For a novel (to the team or company) ML problem, it is common for the team, during their research phase, to find a large list of possible solutions that they may have a poor intuition about the applicability of the approach to their data and use case. It is rather daunting if the team realizes that the only information out there about solving the problem is through the use of a platform that no one is familiar with, a language with which no one is particularly skilled, or a field of ML that no one has experience in. As a result of ignorance (which, to be clear, is not a bad thing; if anyone on this planet can claim to be an expert in all aspects of applied ML, algorithms, and statistics, I will gladly accuse them of falsehood), some potential solutions may be explored that simply do not solve the problem. It's best to find this out early so that the team can pivot their energies to solutions that are more worthwhile.

> *Don't ever be afraid to say, "nope!" when testing something out. Also, don't listen to others saying, "nope" until you've tested it.*

HOW MUCH WORK IS THIS GOING TO BE, ANYWAY?

At the conclusion of the experimental phase, the 'broad strokes' of the ML aspect of the project should be understood. It is important to emphasize that they ***should be understood, but not implemented yet***.

The team should have a general view of what features will need to be developed to meet the project's specifications, as well as what additional ETL work will need to be defined and developed. There should be a solid consensus among the team as to what the data going into and coming out of the models will be, how it will be enhanced, and the tools that will be used to facilitate those needs.

At this juncture, risk factors can begin to be identified. Two of the largest questions are:

- How long is this going to take to build?

- How much is this going to cost to run?

These questions should be part of the phase of review between experimentation and development. Knowing a rough estimate can inform the discussion with the broader team about why one solution should be pursued over another. But should the ML team be deciding alone what implementation to use? There is inherent bias present in any assumptions that the team may have and to assuage these factors, it can be very useful to create a weighted matrix report that the larger team (and the project leader) can use to decide which implementation to use.

Owner bias in ML

We've discussed bias in development choices and how they can affect the preferences of some ML practitioners to favor one methodology over another (chiefly: familiarity, RDD style projects, and a siloed belief that their idea is the best one).

In the case of presenting a report on the findings of an experimentation phase for a large project, though, I've found that it's useful to enlist the assistance of a peer or tech lead in ML to draft a comparative analysis – particularly one who is not only familiar with gauging the cost and benefit of different approaches, but one whom has not been involved with the project up until this point. The objective opinion, without bias, can help to ensure that the data contained in the report is accurate so that the larger team can evaluate options honestly.

Below, in figure 3.18, is an example of one such weighted matrix report (simplified in the name of brevity) to allow for active participation by the greater team. Tools like these (wherein the per-element ratings are 'locked' by the expert reviewer who is doing the unbiased assessment of the relative attributes of different solutions, but the weights are left free to modify actively in a meeting) give a data-driven decision to the team to select amongst the various tradeoffs that each implementation would have.

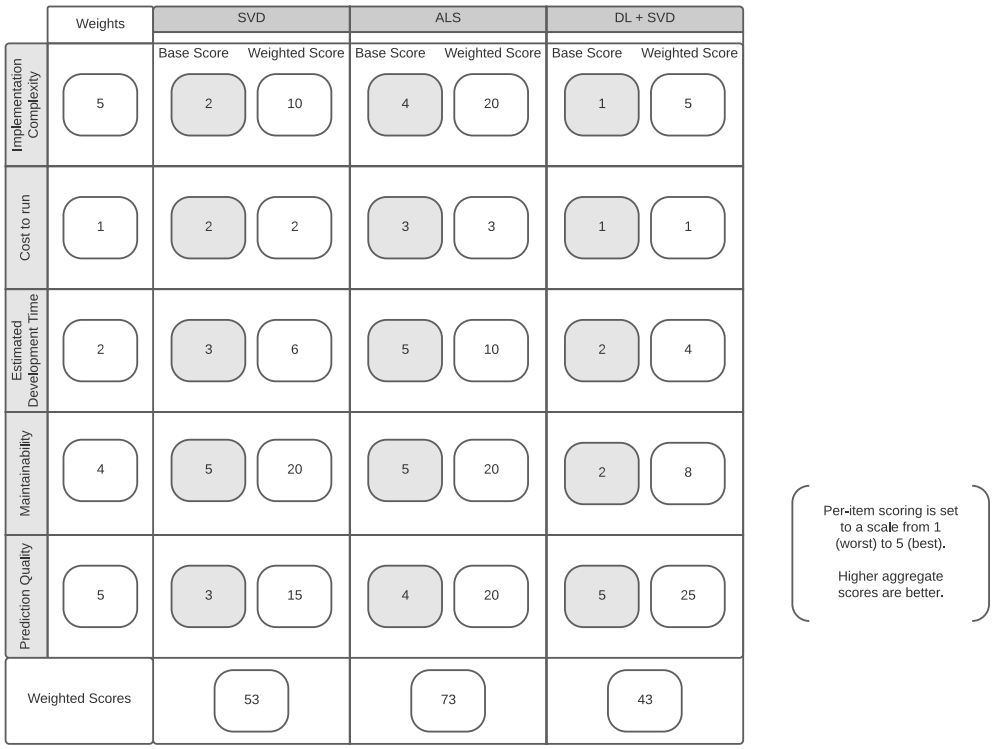


Fig 3.18 Weighted Decision Matrix for evaluating the experimental results, estimation of development complexity, cost of ownership, ability to maintain the solution, and the rough estimation of comparative development time between the three tested implementations for the recommendation engine. In practice, these interactive tools allow for a broader team to make informed decisions (with the weights fields being open to dynamic manipulation) and see what is most important to the project's success. This matrix is intentionally short, for demonstration purposes (as well as arbitrarily filled with values, as the scores are highly dependent on the project, and more importantly, on the team that is building it), and for highly complex projects, these can be dozens of attributes in length.

If this matrix, shown in figure 3.18, were to be populated by an ML team that has never built a system as complex as this, they might employ heavy weightings to "Prediction Quality" and little else. A more seasoned team of ML Engineers would likely over-emphasize "Maintainability" and "Implementation Complexity" (no one likes never-ending Epics and pager alerts at 2am on Friday). The Director of Data Science might only care about "Cost to run", while the project lead is only interested in "Prediction Quality". The important thing to keep in mind is that this is a balancing act. With more people who have a vested interest in the project coming together to debate and explain their perspectives, a more informed decision can be arrived at that can help to ensure a successful and long-running solution.

At the end of the day, as the cliché goes, there is no free lunch. Compromises will need to be made and they should be agreed upon by the greater team, the team leader, and the engineers who will be implementing these solutions as a whole.

3.3 Summary

- The most important phase of an ML project is in the planning and scoping of work. In these critical stages, with the right preparation, many of the leading causes of project failures and abandonment can be eliminated.
- Introducing time-boxing of experimentation can help to focus on a solution rather than on insignificant details, leading to a more robust solution with less rework.
- Setting a phased approach for full team meetings that have concrete deliverables ensures that there are no last-minute surprises. Making the presentations and demos about the solution, rather than the algorithms, keeps the team focused on what matters: the project, not the science.
- Planning a project properly doesn't lock you into a waterfall development style, but rather defines critical aspects of the project that were you to change during development could cause major disruptions.

4

Before you model: Communication and Logistics of projects

This chapter covers

- Communication strategies and techniques for ensuring that the early stages of an ML project (from planning to prototype of a solution) capture the elements that actually solve a business problem
- Determining how to interact with a larger team throughout development to gain feedback and make adjustments through the use of phased planned meetings that focus on relatable demonstrations of the in-progress version of a solution
- Why setting limits on research, experimentation, and prototyping are important, as well as how to define what they should be so that you can actually ship your project to production eventually
- Reasons why and how to incorporate business rules and restrictions on ML projects to ensure that a code base is flexible to adapt to last-minute unexpected changes that need to be made
- How to communicate effectively to a non-technical audience and how to explain what your model can and can't do to reduce the friction of complex projects and ensure that meetings are shorter and more efficient

In my many years of working as a Data Scientist, I've found that one of the biggest challenges that DS teams have in getting their ideas and implementations to be used by a company is rooted in a failure to communicate effectively. Since this field is so highly specialized, there doesn't exist a common layperson's rubric that distills what it is that we do in the same way that other software engineering fields have.

If you were to ask a non-technical person at your company what they think that a Data Engineer does, they might say something along the lines of, "they move data into places that we can query it" or, "they create tables that bring together all of the information of our business so that we can gain insights". Although there may be a great deal of technical nuance that is left out of those statements (the world of Data Engineering is far more complex than just those two things), the essence of what the profession is can be captured quite accurately. A similar case for familiarity could be made about front-end software developers ("they create our web pages") or back-end developers ("they build the core product" or "they own our payment system").

The same can't honestly be said about Data Science. For some companies, who have a fully mature ML ecosystem with dozens of solutions in production that have gained the trust of the general member of the company, this isn't the case. But for the remaining >90% of corporations out there (see study figure 3.8 in Chapter 3, section 3.1.1) however, the general layperson is quite unfamiliar with what a DS team does. This is through no fault of their own, nor of the DS team that works at that company. It's understandable, as the nature of problems that can be tackled are so varied and the work that a DS team at a company might do is so wildly different from another company.

It is of paramount importance to understand the fact that this ignorance exists. It can be alleviated over time as trust and confidence in what you do is built up within an organization, but for the vast majority of practitioners of ML, the fear of the unknown and the deep complexity that exists within the field means that project work for ML teams needs to focus quite heavily on communications. For a project to be successful (read: actually running in production ***and used by the business***), communicating to the business, the SME's in the project group, and even amongst other members of the Engineering organization needs to be a top priority, from the initial ideation sessions and all the way to a model's maintenance period when it's been deployed.

4.1 Communication: defining the problem

As we covered in Chapter 3, we're going to be continuing to discuss the product recommendation system that our DS team was tasked with building. We've seen a juxtaposition of ineffective and effective ways of planning of a project and setting a scoping for an MVP, but we haven't seen ***how*** they got to the point of creating an effective project plan with a reasonable project scope.

In the first example meeting, as we discussed earlier in Chapter 3, part 3.1, the discussion revolved around the final stage of what was wanted, in highly abstract terms. The business wanted personalization of their website. The ML team, when brought in as the experts in prediction, instantly started doing solution-thinking, focusing on the best possible way to generate a data set that would consist of an ordered list of products for each customer that they had. What they didn't do was start asking what exactly the business wanted.

As a retrospective, anyone who had been involved in that disastrous result would eagerly agree that the reason why so much time, money, and effort was wasted was in a failure to communicate.

It all begins with a simple question, that in my experience, is nearly never asked at the outset of a project that involves ML.

Why are we here, talking about this?

Most people, particularly technical people (likely the vast majority of the people that will be in a room discussing this during an initial project proposal and 'brainstorming session'), prefer to focus on the "how" of a project. How am I going to build this? How is the system going to integrate to this data? How frequently do I need to run my code to solve the need?

In the case of the recommendation engine project, if anyone had posed this question, it would have opened the door towards an open and frank conversation about what needs to be built, what the expected functionality should be, and what is expected at the various presentations and ideation sessions that will be scheduled throughout the project (more on these presentation boundaries will be covered in section 4.1.2).

It's important to understand why 'the why' question is so important to ask, if we take a look at the sorts of questions that the different team members are 'programmed' to ask themselves before they enter the room for the first kick-off meeting for a project. Understanding these motivations can help the entire team (particularly the ML members) to focus the conversation for project discussion towards topics that will help inform the minimum requirements for what needs to be built.

The Project Manager in the room will be focused on the "when" of the problem. When can we see a prototype of the solution? When will the minimum viable product (MVP) be delivered? When can we do AB testing?

The project owner (executive sponsor) will be focused on the "what". What lift is this going to give the business? What effect is this going to have on our customers? What is the risk if this goes poorly? What sort of bonus can I expect this year if this goes *really well*?

The ML team will be thinking of the "how" of the problem. How can I research solutions for this problem? How can I use different ML algorithms to generate the data to provide personalized product recommendations? How am I going to do this project when I have dozens of other responsibilities from the business?

None of these questions matter at this point (although they certainly will later on). The first question that everyone should discuss is the answer to the "why we are building this" question. This informs all other discussions, gives a sense of focus to the larger team, and sets the stage for a discussion on the what, how, when, etc. that will be covered in subsequent meetings.

For the recommendation engine initial meeting, if the team had asked one another, "why are we here?", the organic conversation about personalization, about what the business group wants it to do and how they expect it to function can actually take place. These discussions about design requirements would have uncovered all of the issues that the business had with the initial prototype. Instead of generating, as the ML team had done, lists of largely identical (or actually identical) products one after another for users, knowing that this is an issue in the way that data was categorized within the database would have let them know that duplicate resolution and cleansing needed to be done for the prototype

demo. The ordering of products based on vendor contracts could have been added in, preventing the issues seen with excluding this for the demo.

To illustrate the differences in the approaches of handling the communication of project planning, let's look at figure 4.1 below. The integration of the full team's expectations during the planning phase can dramatically reduce the chances of having an unsuccessful MVP delivery.

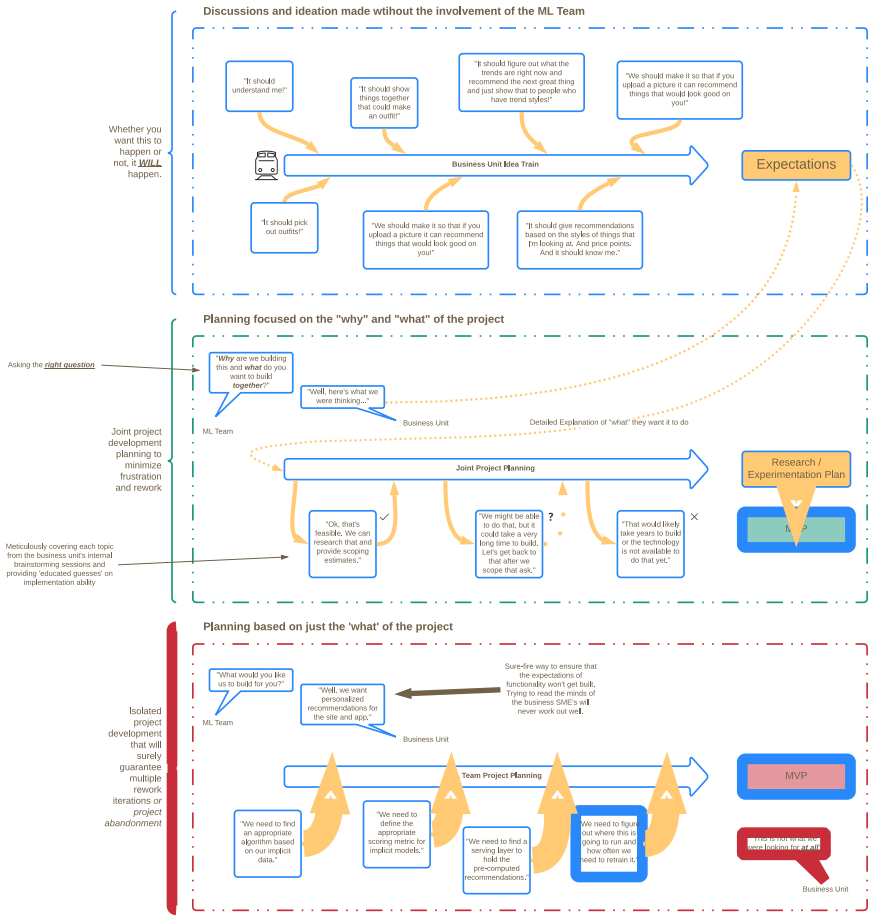


Figure 4.1 A comparison of project planning communication for the site personalization problem. At top, the element that is effectively 'hidden' from the ML team working on the solution is the source of the business unit's expectations. In the middle, these expectations from their internal brainstorming session are incorporated into the planning of the MVP, giving a better chance that the MVP, when delivered, will meet the needs of the business. The bottom section, however, shows what actually happened in our scenario. The ML team, without asking questions about what expected functionality there should be, ended up building something that, though technically correct as a recommendation engine, did not align to the business expectations.

Why are we here and talking about this? A simple question whose answer can inform design considerations for what needs to be built in a prototype. A single question that opens the door to additional questions that can save months of needless throw-away work due to ambiguity of the original intention.

Why are we talking about this first here? It's because I've personally been a part of projects and seen dozens at clients that I've worked with wherein entire quarters worth of work have been spent trying to implement a blog post or a keynote speech that someone gave. ML buzzwords were spouted at the project kickoff meeting, solidifying in the minds of the ML team and all other developers that "we need to build thing 'x' because that's what we've been asked to implement". Sometimes, I've seen these teams rally during development, pivoting to a simpler solution to meet the base required 'need' of the project while throwing away months of early pointless work. Most the of time, however, I've seen a team push an ML solution to production that doesn't do much of anything. It makes predictions, it integrates to the product, it checks all of the boxes for the project design, but it doesn't do what the Executive sponsor envisioned. Either it doesn't make money, or it has the disastrous effect of annoying customers with a functionality that they never asked for.

For the ML practitioners, this is possibly the most critical line of questioning to explore at the start of the project. Don't pay attention to the buzzwords, and certainly don't listen to regurgitated solutions that someone had heard or read about. Listen to the core problem and remember to stick to fundamentals of good ML development regarding solutions.

> ***A great rule of thumb for ML Development***
>
> ***Always build the simplest solution possible to solve a problem. Remember, you have to maintain this thing.***

4.1.1 Understanding the problem

Were we to ask a dozen consultants this question of what went wrong for the MVP-phase of this recommendation engine (e-commerce personalization) project, we would likely get a dozen different answers. The vast majority of which would likely have to do with failed processes of 'best practices' that weren't adhered to (which the consultants would gladly offer to analyze and write a report on that is customized for the company at an exorbitant fee). Others might offer an expensive solution that they've read about that can completely revolutionize their business in this space (technology as a solution to process issues – which never works).

Were we to ask the ML team that presented the simplistic (but as mathematically accurate as possible) solution to the business team during that ill-fated meeting what the issue was, we would likely get an answer along the lines of, "the data is flawed and the model can only predict what is in the dataset". While a valid point, from the perspective of those working on the algorithmic solution to the problem that they had framed within their minds (build a personalization recommendation engine by means of generating a collection of products for each user based on their implicit preferences, shown by their browsing and purchase behavior), it's a very myopic one. What the ML team failed to do was ***understand what the business was expecting in a solution***. They didn't understand it because they didn't ask what the business wanted.

The only real answer to the question of why the MVP was so terrible, however, would be the one given by the business leader for the project. They would simply say:

"This isn't what we asked for."

Specifically, that the group at large didn't *talk about what the model should be doing* and what the customers would want to see from a personalization service within the bounds of how the business operates. At its core, the problem was in a fundamental breakdown in communication. These breakdowns can play out in a number of different ways (from my own personal experience), ranging from slow, simmering passive-aggressive hostility to outright shouting matches if the realization is made towards the end of a project.

What we have here is a failure to communicate

In the many dozens of ML projects that I've been a part of as either a developer, Data Scientist, Architect, or Consultant, the one consistent theme that is common among all projects that never made it to production was a lack of communication. This isn't a reference to a communication failure in the Engineering team, though (although I have certainly witnessed that more than enough for my liking in my career thus far).

The worst sort of breakdown is the one that happens between the ML team and the business unit that is requesting the solution being built. Whether it's a long, slow, drawn-out entropy of communication or a flat-out refusal to speak in a common form of dialogue that all parties can understand, the result is always the same when the 'customers' (internal) aren't being listened to by the developers.

The most destructive time for a lack of communication to come to the realization of everyone involved in the project is around the time of final release to production occurs and the end users consuming the predictions come to the conclusion that not only does something seem a bit off about the results coming from the predictive model, but that it's just fundamentally broken.

Breakdowns in communication aren't restricted only to production release though. The typically happen slowly during the development of the solution or when going through user acceptance testing. Assumptions on all sides are made; ideas are either unspoken or ignored, and commentary is dismissed as either being irrelevant or simple a waste of time during full team meetings.

Few things are as infinitely frustrating as a project failure that is due to communication breakdown amongst a team, but it can be avoided completely. These failures, enormous wastes of time and resources can be attributed to the very early stages of the project – before a single line of code is written – when the scoping and definition of the problem happens. It is entirely preventable, with a conscious and determined plan of ensuring that open and inclusive dialog is maintained at every phase of the project, starting at the first ideation and brainstorming session.

The answer to how to solve these fundamental flaws in the projects that never make it to production is in the form of questions. Specifically, it is in the content and the timing of the questions. The kick-off meeting that we discussed earlier for the recommendation engine project would have been the ideal time and place to start asking the fundamental questions that we'll be going through below.

WHAT DO YOU WANT IT TO DO?

The 'what' for this recommendation is far more important to everyone in the team than the 'how'. By focusing on the functionality of the project's goal (the 'what will it do?' question), the product team can be involved in the discussion. The front-end developers can contribute as well. The entire team can look at a complex topic and plan for the seemingly limitless number of edge cases and nuances of the business that need to be thought of planned for in the building of not just the final project, but the MVP as well.

The easiest way for the team building this personalization solution to work through these complex topics is through simulation and using flow-path models to figure out what the entire team expects from the project in order to inform the ML team on the details needed to limit the options for 'how' they're going to be building the solution.

What do you mean, you don't care about my struggles?

Yes, my fellow ML brethren, I can admit: the 'how' is complex, involves the vast majority of the work for the project, and is incredibly challenging. However, the 'how' is what we get paid to figure out. For some of us, it's the very thing that brought us to this profession. The 'how to solve problems' occupies a lot of the nerd-focused talk that many of us engage in whilst speaking amongst one another. It's fun stuff, it's complex, and it's fascinating to learn.

However, I can assure you, if you didn't already know this, that we're the only ones who find it interesting. Quite literally, no one else cares. Have you ever tried to talk about an ML solution to someone who's not in the field? Ever tried to explain to a friend, significant other, or family member about the details of how you solved a problem? If not, give it a try. See if you can predict when the glazed-over look comes to their eyes and slip in a question midway through the explanation; something along the lines of asking whether they prefer ketchup or mustard on hotdogs. If they just keep nodding and grinning painfully during your diatribe, you'll understand how little people care about the 'how' apart from us.

The rest of the team doesn't care (trust me, even if they feign interest, they're only asking questions about it so they can make it seem like they care – they don't really care) about what modeling approaches are going to be used. Keep these details out of group discussions if you want to have meaningful discussion.

[For the record, I don't condone slipping in ridiculous comments during team meetings to see if people are paying attention but can admit that I do it frequently to see if people are listening. If no one responds to my statements about how much I enjoy long baths in melted ice cream, I know that the conversation has gone 'full nerd' and I need to pivot back to something more approachable.]

The best way for the team working on this project to go through this conversation is to borrow liberally from the best practices of front-end Software Developers. Before a single feature branch is cut, before a single JIRA ticket is assigned to a Developer, front-end dev teams utilize wire frames that are made to simulate the final end-state. For the use case that we've been discussing, a recommendation engine, figure 4.2 below shows what a very high-level flow path might look like initially for a user journey on the website with personalization features applied. Mapping even simplistic architectural user-focused journeys like this can aid the entire team to think about how all of these moving parts will work and open up the discussion to the non-technical team members in a way that is far less confusing than looking at code snippets, key-value collections, and accuracy metrics plotted in highly confusing representations that they are unfamiliar with.

NOTE Even if you're not going to be generating a prediction that is interfacing with a user-facing feature on a website (or any ML that has to integrate with external services), it is incredibly useful to block out the end state flow of what the project is aiming to do while in the planning stage. Keeping the algorithms, code, and tech-focused aspects of the problem out of the equation at first can help to answer the questions around those topics later on, saving you research time and frustration in throwing away too many failed attempts. The name of the game here is to avoid the annoying statement of, "Ugh, it would have been nice if we had known about this before!" (typically salted with invectives, in real-world usage)

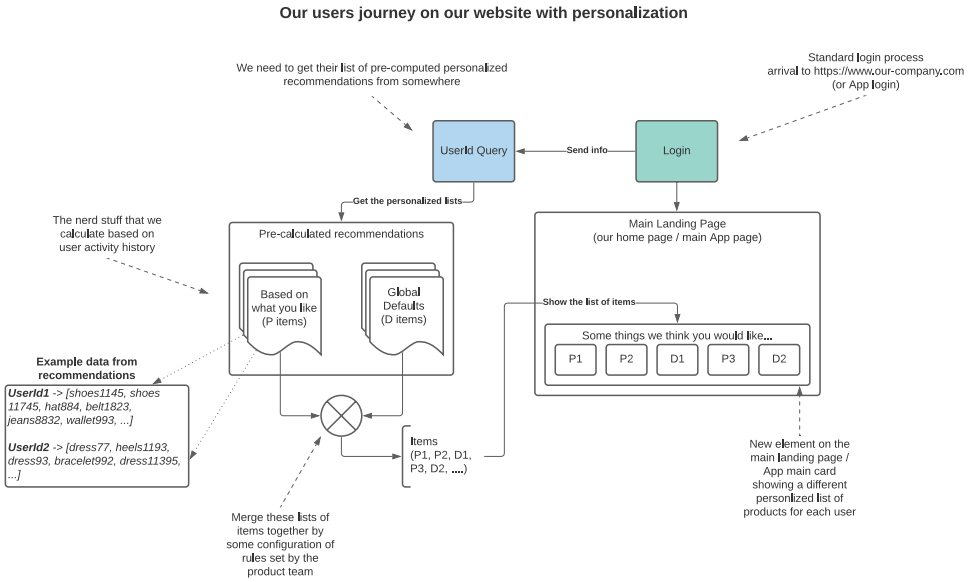


Figure 4.2 A simplified basic overview of the personalization recommendation engine to aid in planning behavior and features of a personalization project. Note the complete lack of technical details and modeling aspects. It just shows to the business team what the MVP will be focused on, rather than focusing on 'how' it will be built. Diagrams like this, used to engage SME's, are very useful to encourage all team members to ideate about what features are needed for the needs of the business.

Figures such as figure 4.2 are helpful for conducting a planning discussion with a broader team. Save your architecture diagrams, modeling discussions, and debates on the appropriateness of scoring metrics for a recommendation system to internal discussions within the ML team. Breaking out a potential solution from the perspective of a user not only keeps the entire team able to discuss the important aspects, but it also opens the discussion to the non-technical team members who will have insights into things to consider that will directly impact both the experimentation and the production development of the actual code. The fact that the diagram is so incredibly simple and easy to see the bare-bones functionality of the system, while hiding the complexity that is contained inside 'pre-calculated recommendations' in particular, the discussion can begin with every person in the room

being engaged and able to contribute to the ideas of what will define the project's initial state.

As an example, figure 4.3 shows what might come out from an initial meeting discussing what could be built with some input from broader team.

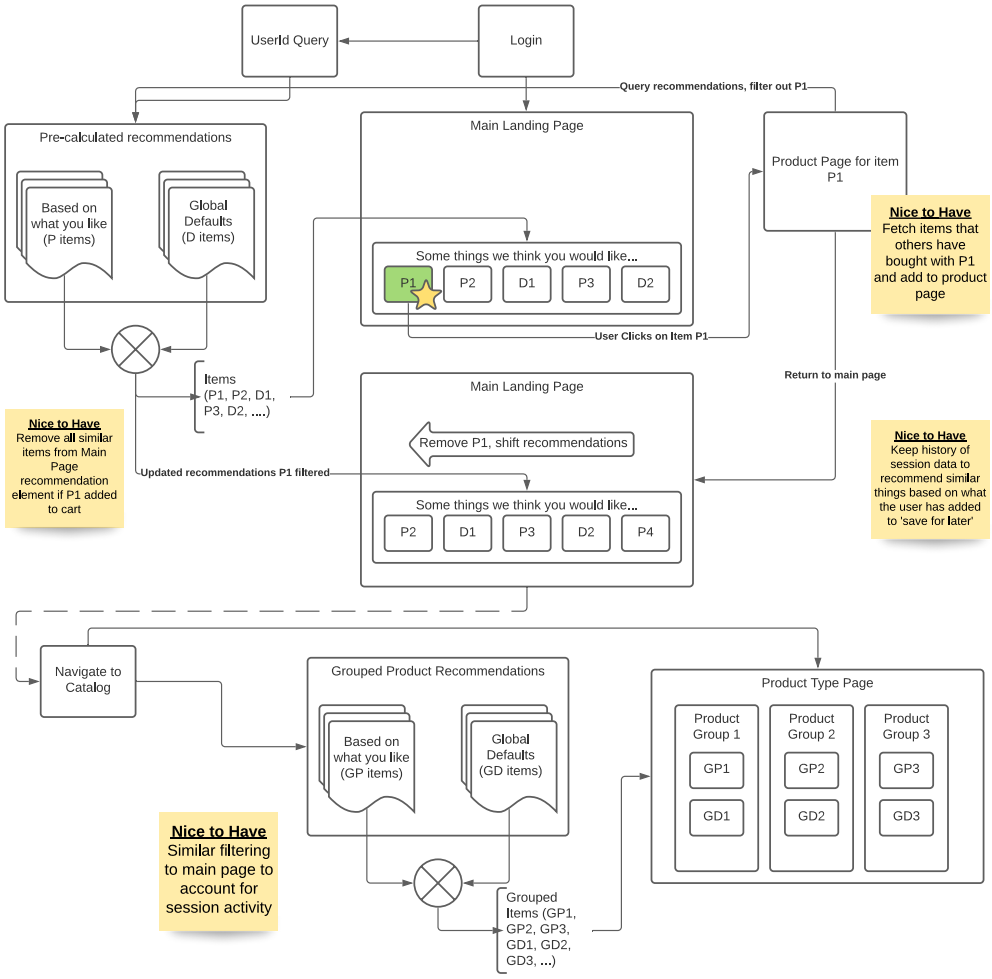


Figure 4.3 The results of an initial planning meeting for a recommendation engine. By running through a user experience journey, ideation on what would be nice compared to what is essential functionality, the discussion about the project can stay within the bounds of the project without having to spend time on the engineering aspects of the project.

Figure 4.3, when compared with figure 4.2, shows the evolution of the project's ideation. It is important to consider that many of the ideas that are presented would likely have not

been considered by the ML team had the product teams and SME's not been a part of the discussion. Keeping the implementation details out of the discussion allows for everyone to consider the biggest question: "Why are we talking about building this and how should it function for the end-user?".

What is a user experience journey?

Borrowed liberally from the field of product management in the B2C (business to customer) industry, a user experience journey (or a journey map) is a simulation of a product, exploring how a new feature or system will be consumed by a particular user.

It's a form of a map, of sorts, beginning with the user interacting with your system (logging in, in the example in figure 4.3) initially, and then following them through the user-facing interactions that they will have with different elements of the system.

I've found that these are useful for not only e-commerce and application-based implementations that ingest ML to serve a feature but can be quite helpful in designing even internal-facing systems. At the end of the day, you want your predictions to be used by someone or something. Many times, drawing a map out of how that person, system, or downstream process will interact with the data that you're producing can aid in designing the ML solution to best meet the needs of the 'customer'. It can help find areas that can inform the design of not only the serving layer, but also elements that may need to be considered as critical features during the development of the solution.

An important thing to notice on this diagram in figure 4.3 is the 4 items marked "Nice to have". This is a very important and challenging aspect of initial planning meetings. Everyone involved wants to brainstorm and work towards making the 'most amazing solution ever', but in the spirit of actually getting something into production, a bit of realism should creep in, even at early stages. A sincere focus should be made on the essential aspects of the project (the Minimum Viable Product (MVP)) to make it function correctly. Anything ancillary to that should be annotated as such; ideas should be recorded, referenceable, modified, and referred to throughout the experimentation and development phases of the project. What once may have seemed insurmountably difficult could prove to be trivial later on as the code base takes shape and it could be worthwhile to include these features even in the MVP.

The only bad ideas are the ignored ones. – me

Don't ignore ideas, but also don't allow every idea to make it into the core experimentation plan. If the idea seems far-fetched and incredibly complex, simply revisit it later once the project is taking shape and the feasibility of implementation can be considered when the total project complexity is known to a deeper level.

A note on planning meetings

I've been a part of many planning meetings in my career. They typically fall into 1 of 3 categories. The examples above, in figures 4.2 and 4.3, represent the planning events that I've seen and had the most success with using. The ones that are least useful (where follow-up meetings, off-line discussions, and resulting chaos ensues) are those that focus either entirely on the ML aspect of the project or on the Engineering considerations of making the system work.

If the model is the main point of concern, many of the people in the group will be completely alienated (they won't have the knowledge or frame of reference to contribute to a discussion of algorithms) or annoyed to the point that they disengage from the conversation. At this point it's just a group of Data Scientists arguing about whether they should be using ALS or Deep Learning to generate the raw recommendation scores and how to fold in historical information to the prediction results. It's pointless to discuss these things in front of a marketing team.
If the engineering aspects are the focus, instead of creating a diagram of a user experience flow path, the diagram will be an architectural one that will be alienating an entirely different group of people. Both an engineering and a modeling discussion are important to have; but they can be conducted without the broader team and can be iteratively developed later – after experimentation is completed.

During the process of walking through this user-experience workflow, it could be discovered that the assumptions that different people on the team have of how one of these engines work are in conflict. The marketing team assumed that if a user clicks on something, but doesn't add it to their cart, provides an inferred dislike of the product. They don't want to see that product in recommendations again for the user.

How does this change the implementation details for the MVP?

The architecture is going to have to change. It's a whole lot easier to find this out now and be able to attribute scoped complexity for this feature while in the planning phase than before a model is built which then has to be monkey-patched to an existing code base and architecture. It also may, as shown above in figure 4.3, start adding to the overall view of the engine: what it's going to be built to support and what will not be supported. It will allow the ML team, the front-end team, and the data engineering team to begin to focus on what they respectively will need to research and experiment with, in order to prove or disprove the prototypes that will be built.

Remember – all of this is before a single line of code is written.

Asking the simple question of 'how should this work' and avoiding focusing on the 'standard algorithmic implementations' is a habit that can help ensure success in ML projects more so that any technology, platform, or algorithm. This one question is arguably the most important question to ask to ensure that everyone involved in the project is on the same page. I recommend asking this question along with the necessary line of questioning to eke out the core functionality that needs to be investigated and experimented on. If there is confusion or a lack of concrete theories regarding what the core needs are, it's much better to sit in hours of meetings to plan things out and iron out all of the business details as much as possible in the early stages, rather than waste months of your time and effort in building something that doesn't meet the mental vision of what the project sponsor has.

WHAT DOES THE IDEAL END-STATE LOOK LIKE?

The ideal implementation is hard to define at first (particularly before any experimentation is done), but it's incredibly useful to the experimentation team to hear all aspects of ideal state. During these open-ended stream-of-consciousness discussions, a tendency of most ML practitioners is to instantly decide what is and isn't possible based on the ideas of people who don't understand what ML is. My advice in this is to simply listen. Instead of shutting down a thread of conversation immediately as being 'out of scope' or 'impossible', the path that creative ideation may follow could lend itself to a unique and more powerful ML solution that what you may have come up with on your own.

The most successful projects that I've worked on over the years have come from having these sort of creative discussions with a broad team of subject matter experts (and, when I've been lucky, the actual end-users) to allow me to shift my thinking into creative ways of getting as close as possible to what their vision is.

Having the discussion about an ideal end-state isn't just for the benefit of a more amazing ML solution though. It helps to engage the person asking for the project to be built and allows their perspective, ideas, and creativity influence the project in positive ways. Going hand in hand with that, it helps build trust and a feeling of ownership in the development of the project that can help bring a team together.

Learning to listen closely to the needs of a customer of your ML project is one of the most important skills of an ML Engineer – far more than mastering any algorithm, language, or platform. It will help guide what you're going to try, what you're going to research, and how to think differently about problems in order to come up with the best solution that you possibly can.

In the above scenario, shown in figure 4.3, the initial planning meeting resulted in a 'rough sketch of ideal state'. This likely *will not be what the final engine will be* (based on my experience, that most certainly is never the case), but what this diagram will do is inform how you will be converting those functional blocks into systems. It will help to inform the direction of experimentation, as well as the areas of the project that you and the team will need to research thoroughly to minimize or prevent unexpected scope creep, as shown in figure 4.4.

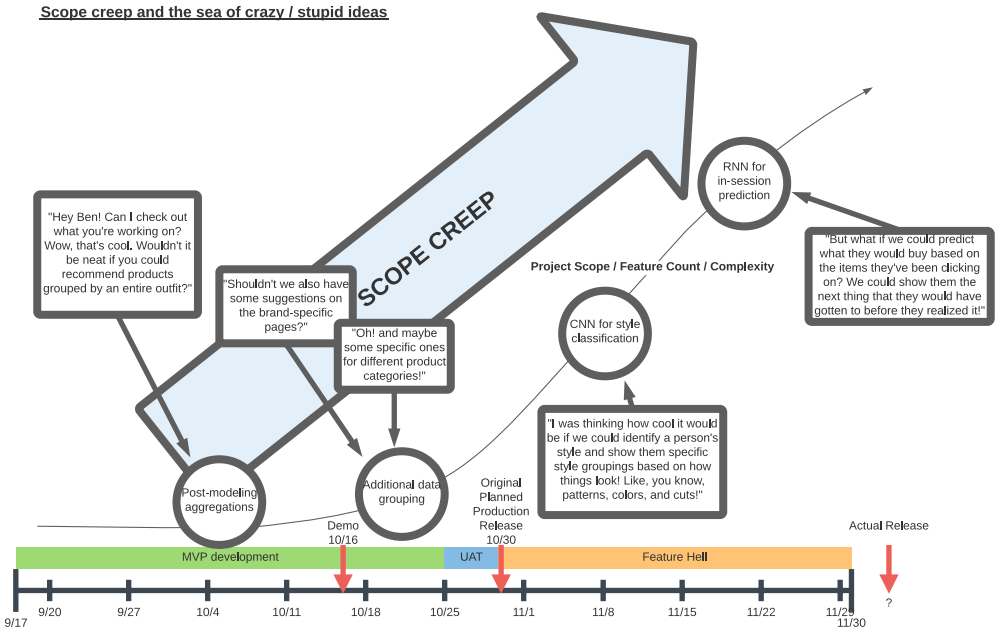


Figure 4.4 The dreaded scope creep. Be wary of ideas that come up, particularly amongst overzealous team members. Ideas a great. Features are great. But if you ever want to actually release a project to production, you need to identify what is in-scope for the agreed upon project, what the impact to the delivery will be if you add a feature, and what is absolutely out of scope for a project. Even if you can build a feature, doesn't mean you should.

Figure 4.4 should be familiar to any reader who has ever worked at a startup. The excitement and ideas that flow from driven and creative people who want to 'do something amazing' is infectious and can, with a bit of tempering, create a truly amazing company that does a great job at their core mission. However, without that tempering and focus applied, particularly to an ML project, the sheer size and complexity of a solution can quite rapidly spiral out of control.

> **NOTE** I've never, not even once, in my career allowed a project to get to this level of ridiculousness as shown in figure 4.4 (although a few have come close). However, on nearly every project that I've worked on, comments, ideas, and questions like this have been posed. My advice: thank the person for their idea, gently explain in non-technical terms how it's not possible at this time and move on to finish the project.

Scope creep – an almost guaranteed assassination of a project

Improper planning (or planning without involving the team that the project is being built for) is a perfect recipe for one of the most frustrating ways of having a project die a slow, whimpering death. Also known as 'ML death by a thousand requests', this concept materializes at later stages of development; particularly when a demo is shown off to a team who is uninformed about the details that went into building the project. If the customer (the internal business unit) was not party to the planning discussions, it is inevitable that they will have questions, and many of them, about what the demo does.

In nearly every case that I've seen (or caused in my earlier days of trying to 'hero my way through' a project without asking for input) the result of the demo session is going to be dozens of requests for additional features and requirements to be added.

This is expected (even in a properly designed and planned project), but if the implementation is unable to easily include critical features that relate to immutable business operation 'laws', then a potential full reimplementation of the project could be required, leaving decision makers with the difficult decision of whether to delay the project due to the decision that the ML team (or individual) made, or to scrap the project entirely to prevent the chances of a repeat of the initial failure.

There are few things that are more devastating to hear in the world of ML than intensely negative feedback immediately after something goes live in production. Getting a flood of emails from Executive-level staff about how the solution that you just shipped is recommending cat toys to a dog owner is laughable, but one that is recommending adult-themed products to children is about as bad as it can get. The only thing that is worse than that is, right before the project is shipped, during UAT, you realize that there is an insurmountable list of changes that need to be made in order to satisfy the urgent requirements of the business and that it would take less time to start the project over from scratch than it would to make the changes to the existing solution.

Identifying scope creep is important, but the magnitude of it can be minimized (in some cases eliminated) if the appropriate level of discussion and critical aspects of a project are discussed in (sometimes excruciating recursive and painful) detail well before a single character is typed in an experimentation notebook or IDE.

WHO IS YOUR CHAMPION FOR THIS PROJECT THAT I CAN WORK WITH ON BUILDING THESE EXPERIMENTS OUT?

While this may sound sacrilege to most Data Science practitioners, I can honestly state that the most valuable member of any team I've worked on a project with has been the subject matter expert (SME) – the person that was assigned to work with me or the team I was on to check our work, answer every silly question that we had, and provide creative ideation that helped the project grow in ways that none of us had envisioned. While usually not a technical person, they had a deep connection and expansive knowledge of the problem that we needed to solve. Taking a little bit of extra time to translate between the world of Engineering and ML to layperson's terms has always been worth it, primarily due to the fact that it creates an inclusive environment where that SME is invested in the success of the project since they see that their opinions and ideas are being considered and implemented.

I can't stress enough that the *last person* that you want to fill this role is the actual Executive-level project owner. While it may seem logical at first to assume that it will be easier to have the Manager, Director, or VP of a group be the one that can be asked for approval for ideation and experimentation, I can assure you that you will never get more than a 'pencil-whipping' on ideas (a Roman Emperor-esque thumbs-up / thumbs-down) that will stagnate a project. The reason being is that these people are incredibly busy dealing with dozens of other important time-consuming tasks that they have delegated to others. If you are expecting this person (who themselves may or may not be an expert in the domain that

the project is addressing) to provide extensive and in-depth discussions on minute details (all ML solutions are all about the small details, after all) then you will be sorely disappointed. In that first kick-off meeting, make sure that you have a resource from the team who is an SME and has the time and temperament to deal with this project and a bunch of nerdy engineers.

WHEN SHOULD WE MEET TO SHARE PROGRESS?

Due to the complex nature of most ML projects (particularly ones that require so many interfaces to parts of a business as a recommendation engine), meetings are critical. However, not all meetings are created equally. While it is incredibly tempting for people to want to have 'cadence meetings' that are on some weekly proscribed basis, project meetings should coincide with milestones associate with the project.

These project-based milestone meetings should:

- Not be a substitute for daily standup meetings
- Not overlap with team-focused meetings of individual departments
- Always be conducted with the full team present
- Always have the project lead present to make final decisions on contentious topics
- Be focused on presenting the solution as it stands at that point and nothing else

It's incredibly tempting for discussions to happen outside of these structured presentation and data-focused meetings. Perhaps people on your team who are not involved in the project are curious and would like to provide feedback and additional brain storming sessions. Similarly, it could be convenient to discuss a solution to something that you're stuck on with a small group from the larger team in an isolated discussion.

I cannot stress strongly enough how much chaos any of these outside-of-the-team discussions can be. When decisions are made in a large-scale project (even in the experimentation phase) that others are not being made aware of, it introduces an uncontrollable chaos to the project that is difficult to manage for everyone involved in the implementation. If the ML team decided in a vacuum, for instance, that in order to best serve the product group recommendations they would, instead of building an additional API to provide the recommendations, reuse the personalized data sets to respond to the REST request from the frontend engineers, without letting the frontend engineers know of this change, potentially cause weeks of rework for the frontend team. Introducing changes without notifying and discussing these changes in the larger group risks wasting a great deal of time and resources, which in turn erodes the confidence that the team and the business at large has in the process. It's a fantastically effective way of having the project become shelf-ware.

At the early meetings that happen, it is imperative for the ML team to communicate to the group the need for these event-based meetings; to let everyone know that changes that might seem insignificant to other teams could have rework risks associated with them that could translate to weeks or months of extra work by the ML team. Similarly, ML changes could have dramatic impacts on the other teams.

To illustrate the inter-connectedness of the project and how hitting different deliveries can impact a project, let's take a look at what this solution would look like in a relatively large company (let's say over 1000 employees) with the roles and responsibilities divided

amongst various groups. In a smaller company (a startup, for instance) many of these responsibilities would fall either on the front-end or ML team rather than separate Data Engineering teams. Figure 4.5 below shows how dependencies from the ML experimentation, for instance, affects the future work of both the Data Engineering and Front-end Development teams. With excessive delays or required rework being done, it would not be only the ML team that would be reworking their code; rather, it could be weeks of work being thrown away by an entire Engineering organization. This is why planning, frequent demonstrations of the state of the project, and open discussions with the relevant teams is so critical.

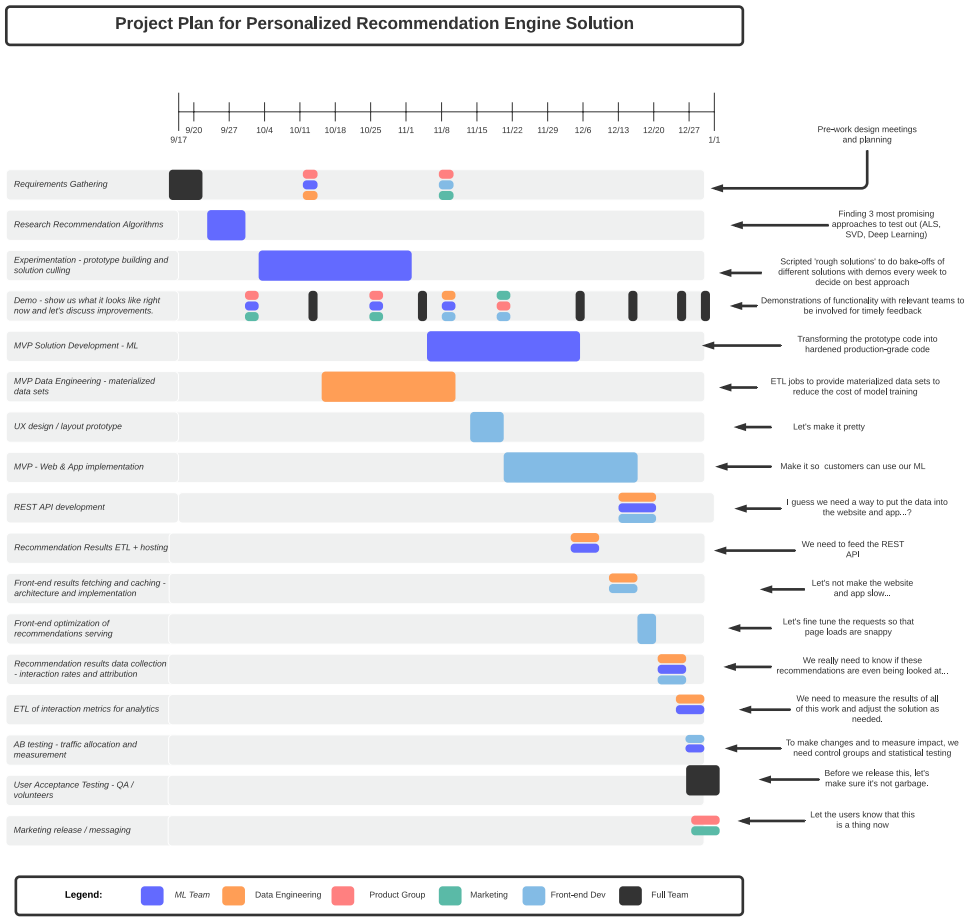


Figure 4.5 The project plan for our personalized recommendation engine showing the dependencies not only in the process of developing the model, but how the work being done by the ML team at various stages affects the work of other teams. Having repeated demonstration phases can help to ensure that the features being

developed and the direction of the project can be as seamless as possible with respect to ensuring that each team's work is done in concert and that major restructuring of the solution doesn't need to be done, effectively delaying or causing massive amounts of rework.

Figure 4.6, shown below, illustrates a very high-level Gantt chart of the milestones associated with a general e-commerce ML project, focusing solely on the 'main concepts'. Keeping charts like this as a common focused communication tool can greatly improve the productivity of all of the teams and reduce a bit of the chaos in a multi-discipline team, particularly across the walls of department barriers.

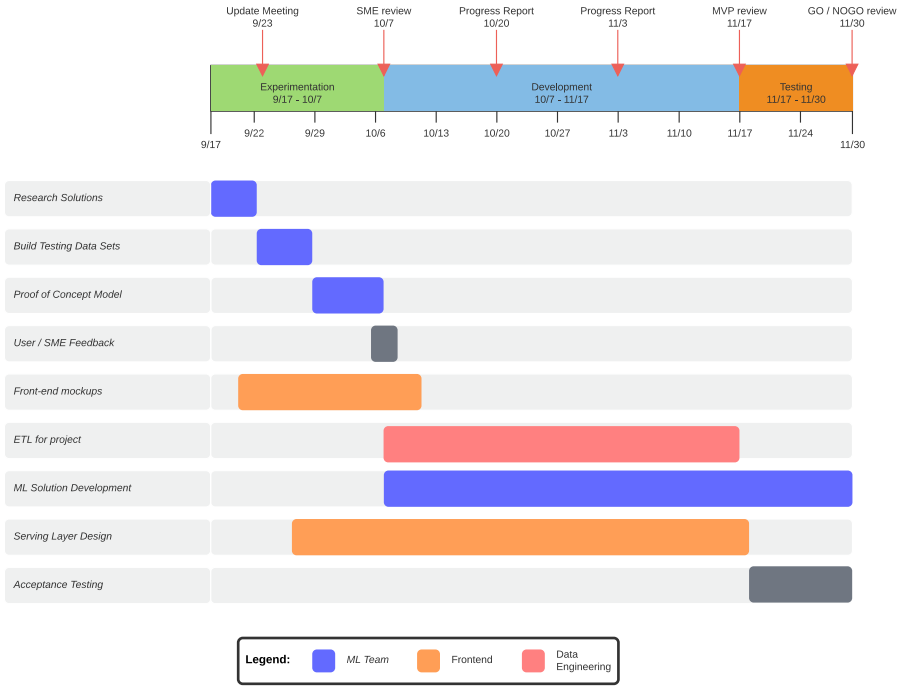


Figure 4.6 Simplified (extremely high-level) project timeline with critical progress meetings represented by the arrows along the top. Project boundary markers, with the entire team (SME and all engineering teams) meeting to discuss the status, discovered complexities, solutions, and future plans are important to ensure that everyone is available to review and understand the state of the project.

As figure 4.6 shows with the milestone arrows along the top, there are critical stages where the entire team should be meeting together to ensure that all members of the team understand the implications of what has been developed and discovered so that they may collectively adjust their own project work. These break points (most teams that I've worked

with do these meetings on the same day as their sprint planning, for what it's worth) allow for demos to be shown, basic functionality to be explored, and risks identified. It allows for a common communication point to happen with two main purposes:

- Minimize the amount of time wasted on rework
- Make sure that the project is on track to do what it set out to do

While it's not absolutely necessary to spend the time and energy to create Gantt charts for each and every project, it is advisable to create at least something to track progress and milestones against. Colorful charts and inter-disciplinary tracking of systems development certainly doesn't make sense for solo outings where a single ML Engineer is handling the entire project on their own. Even in situations where you may be the only person putting fingers-to-keyboard, figuring out where major boundaries exist within the project's development are and scheduling a bit of a show-and-tell can be extremely helpful. Do you have a demonstration test set from a tuned model that you want to make sure solves the problem? Set a boundary at that point, generate the data, present it in a consumable form, and show it to the team that asked for your help in solving the problem. Getting feedback – the right feedback – in a timely manner can save you and your customer a great deal of frustration.

4.1.2 Critical discussion boundaries

The next question that begs to be asked is: "Where do I set these boundaries for my project?" While each project is completely unique with respect to the amount of work that needs to get done to solve the problem, the number of people involved in the solution, and the technological risks surrounding the implementation, there are a few general guidelines of stages that are helpful to set as 'minimum required meetings'.

Within the confines of the recommendation engine that we're planning on building, we need to set some form of a schedule around when we will all be meeting, what we will be talking about, what to expect from those meetings, and most importantly, how the active participation from everyone involved in the project will help to minimize risk in the timely delivery of the solution.

Let's imagine for an instant that this is the first large-scale project involving ML that the company has ever dealt with. It's the first time that so many developers, engineers, product managers, and SMEs have worked in concert, and none of them have an idea of how often to meet and discuss the project. You realize that meetings do have to happen, since you identified this in the planning phase. You just don't know when to have them.

Within each team, people have a solid understanding of the cadence with which they can deliver solutions (assuming they're using some form of Agile, they're likely all having scrum meetings and daily stand-ups), but no one is really sure what the stages of development look like in other teams. The simplistic answer is, naturally, the worst answer: "Let's meet every Wednesday at 1pm." Putting a 'regularly scheduled program' meeting into place, with a full team of dozens of people, will generally result in only two things: annoyance and awkward silence. In the worst of all cases, it can result in people just ignoring the meeting, preventing the critical advice and ideas that they may have contributed to a key detail that could have increased the chances of the project's success.

A note on meeting fatigue for ML teams

Everyone wants to talk to the ML team. Sadly, it's not for the reasons that the ML team thinks it's for. It's not because you're cool, or that you're interesting to talk to. They likely don't want to hear about your algorithms or some new technique for distributed error minimization that someone came up with.

They want you in meetings because they're scared of you. They're scared that you're going to go full prison-riot on a project, inmates-running-the-asylum style development. They're scared because they don't know what you do, how you do it, and, most importantly, are afraid that you're going to rack up some ludicrous bill while testing out some theory.

These are understandable fears. However, they're largely unwarranted with proper scoping assurances, deadlines that are delivered upon, and solid layperson-level communication throughout every stage of the project.

What the business doesn't realize is that frequent status meetings are not simply a drain on the team's productivity for the hour that they spent on a call or in a meeting room discussing the details of what they've been working on. Rather, it's the two hours before the meeting that were spent preparing to present minor updates from the last meeting's slides, the hour of sitting in the meeting, and the 4 hours after the meeting where they're all trying to resume the actual work that they had been in the process of doing prior to preparing for the meeting.

The fewer and more focused meetings that are had, the better.

The more logical, useful, and efficient use of everyone's time is to meet to review the solution-in-progress *only when there is something new to review*. But when are those decision points? How do we define where these boundaries are in order to balance the need to discuss elements of the project with the exhaustion that comes with reviewing minor changes with too-frequent work-disrupting meetings?

As I mentioned before, each project is slightly different and were we to fully plot out all of the stages from a recommendation engine project's experimentation to production serving deployment in Figure 4.5, you, the reader, would need a scanning electron microscope to read the text. Figures 4.5 and 4.6 do cover some general critical discussion points, though, which we will cover below.

POST-RESEARCH PHASE DISCUSSION (UPDATE MEETING)

For the sake of example, let's assume that the ML team identified that 2 different models would be required to be built in order to satisfy the requirements from the planning phase "user journey" simulation. Based on their research, they decided that they would like to pit both collaborative filtering and FP-growth market basket analysis algorithms against Deep Learning implementations to see which provides a higher accuracy and lower cost of ownership for retraining.

The ML lead assigned two groups of Data Scientists and ML Engineers to work on these competing implementations and had them both generate simulations of the model results on the exact same synthetic customer data set, providing mock product images to a wireframe of what the different pages that these recommendations would be displayed on for the actual website.

While tempting it may be, this meeting should not focus on any of the implementation details. Instead, it should focus solely on the results of the research phase: the whittling down of nigh-infinite options that have been read about, studied, and played about with. The team has found a lot of great ideas, an even larger group of potential solutions that won't

work based on the data that is available and have reduced the list of great ideas to a bake-off of two implementations that they're going to pit against one another. Don't bring up all of the options that you've explored. Don't mention something that has amazing results but will likely take 2 years to build. Instead, distill the discussion to the core details that are required to get the next phase going: experimentation.

Show to the SME's what these two options are solely within the confines of presenting what can be done with each of the algorithmic solutions, what is impossible with one or both, and when they can expect to see a prototype from them to see which one they like better. It's best to leave all of the other nerd stuff out. The absolute last thing that the ML team should want to have happen is to alienate the non-technical audience with details that they both don't care about and are incapable of discussing due to a lack of common experience and technical background.

If there is no discernable difference to the quality of the predictions, the decision of which to go with should be made with the elucidation of drawbacks of the approaches, leaving the technical complexity or implementation details out of the discussion. Don't try to throw out statements like, "Well, the inference results are clearly skewed due to the influence of a priori implicit affinity due to the quantity of purchase events within the categorical grouping." (that pained me to even type that). Instead, opt for something along the lines of, "I think they bought a bunch of those shoes." Keep the discussion in these dense meetings focused on discussions in relatable language and references that your audience will comprehend and associate with. You can do the translating in your head and leave it there. All of the technical details should be discussed internally by the ML team, the architect, and Engineering management.

In many cases that I've been involved with, the experimental testing phase may test out a dozen ideas, but only present the two most acceptable to a business unit for review. If the implementation would be overly onerous, costly, or complex, it's best to choose to present options to the business that will *guarantee the greatest chance of project success*. Even if they're not as fancy or exciting as other solutions. Remember: the ML team has to maintain the solution (so something that sounds 'really cool' during experimentation can turn into a nightmare to maintain).

POST-EXPERIMENTATION PHASE (SME / USER ACCEPTANCE TESTING(UAT) REVIEW)

Following from the phase of experimentation, the sub-teams within the ML group have built two prototypes for the recommendation engine. In the previous milestone meeting, the options for both of these were discussed, their weaknesses and strengths presented in a way that the audience could understand. Now it's time to lay the prediction cards out on the table and show off what a prototype of the solution looks like.

Before, during reviews of the potential solutions, some pretty rough predictions were shown. Duplicate products with different product id's were right next to one another, endless lists of one product types were generated for some users (there's no way that anyone likes belts that much), and the list of critical issues with the demo were listed out for consideration. In those first early pre-prototypes, the business logic and feature requirements weren't built out yet, since those elements directly depended on the models' platform and technology selection.

The goal of the experimentation phase completion presentation should be in showing a mock-up of the core features. Perhaps there were requirements for ordering of elements based on relevancy. There may have been special considerations for recommending of items based on price point, recent non-session-based historical browsing, and the theory that certain customers have implicit loyalty to certain brands. Each of these agreed upon features should, at this point, be shown to the entire team. The full implementation, however, should not be done by this point, but merely simulated to show what the eventual designed system would look like.

The results of this meeting should be similar to those from the initial planning meeting; additional features that weren't recognized as being important can be added to the development planning and if any of the original features are found to be unnecessary, they should be removed from the plan.

Revisiting the original plan, and updated user experience might look something like figure 4.7 below.

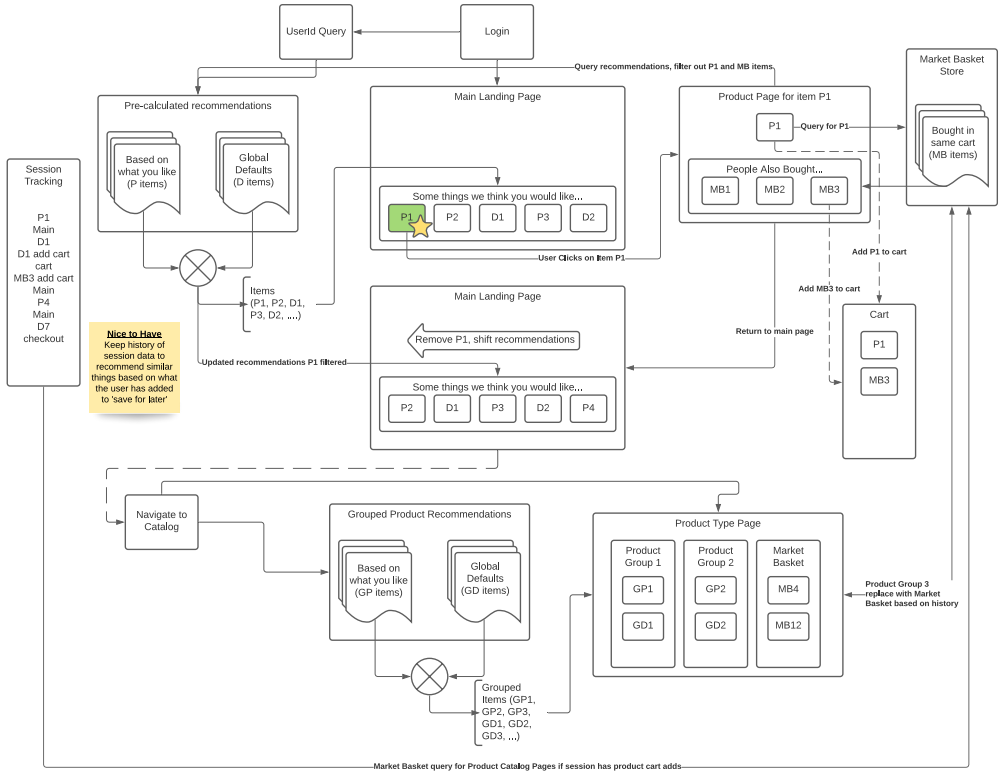


Figure 4.7 The evolution of user-centric functionality for the recommendation engine after the initial experimentation review. Note the addition of the "Market Basket Store" (a new model), the acceptance of earlier 'nice to have' elements into the design requirements, and a single earlier 'nice to have' that was determined to be out-of-scope for the initial release of the engine. Refining plans in this manner allows for

both inclusive discussion from the larger team and a referenceable template for building an architectural plan for the system.

With the experimentation phase out of the way, the ML team can explain that the 'nice to have' elements from earlier phases are not only doable but can be integrated without a great deal of extra work. Figure 4.7 shows the integration of those ideas (Market Basket Analysis, dynamic filtering, and aggregated filtering), but also maintains one idea as a 'nice to have'. If it is found that, during development, the integration of this feature would be attainable, then it is left as part of this living planning document.

The most important part of this stage's meeting is that everyone on the team (from the front-end developers who will be handling the passing of event-data to the server to conduct the filtering to the product team) is aware of the elements and moving pieces that are involved in the project at this stage. It ensures that the team understands what elements need to be scoped, what the general epics and stories are that need to be created for sprint planning, and that a collaborative estimation of the implementation is arrived at.

DEVELOPMENT SPRINT REVIEWS (PROGRESS REPORTS FOR A NON-TECHNICAL AUDIENCE)

Conducting recurring meetings of a non-engineering focused bent are useful for more than simply just passing information from the development teams to the business. They can serve as a bellwether of the state of the project and at what stages integration of disparate systems can begin. What they should still be, though, is a high-level project-focused discussion.

The temptation for many cross-functional teams that work on projects like this are to turn these update meetings into either an 'uber retrospective' or a 'super sprint planning meeting'. While there are uses for such discussions (particularly for integration purposes between different engineering departments), those topics should be reserved for the engineering teams to meet and discuss with one another. A full-team progress report meeting should make the effort to generate a current-state demonstration of what the progress is up to that point. Simulations of the solution should be shown to ensure that the business team and SME's can provide relevant feedback to details that might have been overlooked by the engineers working on the project.

These periodic meetings (either every sprint or every other sprint) can help prevent the aforementioned dreaded scope creep and the dreaded 11th hour finding that a critical component that wasn't noticed as necessary is missing, causing massive delays in the project's delivery.

MVP REVIEW (FULL DEMO WITH UAT)

"Code complete" can mean a great many different things to different organizations. In general, it is widely accepted to be a state in which:

- Code is tested (and passes unit / integration tests)
- The system functions as a whole in an evaluation environment using production-scale data (models have been trained on production data)
- All agreed-upon features that have been planned are complete and perform as designed

This doesn't mean that the subjective quality of the solution is met, though. This stage simply means "the system will pass recommendations to the right elements on the page" for this recommendation engine example. The MVP review and the associated UAT that goes into preparing for this meeting is the stage at which subjective measures of quality are done.

What does this mean for our example, a recommendation engine? It means that the SME's log in to the UAT environment and navigate the site. They look at the recommendations based on their preferences and make judgements on what they see. It also means that high-value accounts are simulated, ensuring that the recommendations that the SME's are looking at through the lens of these customers are congruous to what they know of those types of users.

For many ML implementations, metrics are a wonderful tool (and most certainly should be heavily utilized and recorded for all modeling), but the best gauge of determining if the solution is qualitatively solving the problem is to use the breadth of knowledge of internal users and experts who can use the system before it's deployed to end users.

I've been party to meetings of this topic (evaluating the responses to UAT feedback of a solution that they developed over a period of months) and have seen arguments break out between the business and the ML team about how one particular model's validation metrics are higher, but the qualitative review quality is much lower than the inverse situation. This is exactly why this particular meeting is so critical. It may uncover glaring issues that were missed in not only the planning phases, but in the experimental and development phases as well. Having final sanity checks on the results of the solution can only make the end-result better.

To my fellow ML brethren, I can only offer the advice to embrace the feedback, particularly if it is negative, and come up with ideas collectively to address these concerns. In the end, it will only make a solution better.

There is a critical bit of information to remember about this meeting and review period dealing with estimates of quality. Nearly every project (at least enjoyable and excitingly complex ones) are accompanied with a large dose of creator bias. When creating something, particularly an exciting system that has a sufficient challenge to it, the creators can overlook and miss important flaws in it due to familiarity and adoration of it.

> ***A parent can never see how ugly or stupid their children are. It's human nature to unconditionally love what you've created. – Every rational parent, ever.***

If, at the end of one of these review meetings, the only responses are overwhelming positive praise of the solution, it should raise some concerns with the team. One of the side effects of creating a cohesive cross-functional team of people who all share in a collective feeling of ownership of the project is that emotional bias for the project may cloud judgement of its efficacy. If ever you notice a summarization meeting of the quality of a solution and hear nary an issue, it would behoove you and the project team to pull in others at the company who have no stake in the project. Their unbiased and objective look at the solution could pay off dividends in the form of actionable improvements or modifications that the team, looking through the bias of their nigh-familial adoration of the project would have completely missed.

PRE-PRODUCTION REVIEW (FINAL DEMO WITH UAT)

This final meeting is right before "go time". Final modifications are complete, feedback from the UAT development-complete tests have been addressed, and the system has run without blowing up for several days. The release is planned for the following Monday (pro tip: never release on a Friday) and a final look at the system is called for. System load testing has been done, responsiveness measured through simulation of 10x the user volume at peak traffic, logging is working, and the model retraining on synthetic user actions have shown that the models adapt to the simulated data. Everything from an Engineering point has passed all tests.

So why are we meeting again? – everyone who is exhausted by countless meetings

At this final meeting before release, a comparison between what was originally planned, what features were rejected for out of scope, and what had been added should be reviewed. This can help inform expectations of the analytics data that should be queried upon release. The systems required for collecting the data for interactions for the recommendations have been built, an AB testing data set created that can allow for analysts to check the performance of the project.

This final meeting should focus on where that data set is going to be located, how engineers can query it, and what charts and reports are going to be available (and how to access them) for the non-technical members of the team. The first few hours, days, and weeks of this new engine powering portions of the business is going to receive a great deal of scrutiny. To save the sanity of the analysts and the ML team, a bit of preparation work to ensure that people can have 'self-service' access to the metrics and statistics for the project will ensure that critical data-based decisions can be made by everyone in the company, even those not involved in the creation of the solution.

A note on patience

With a new project releasing of the scope of a recommendation engine for a e-commerce company, it is important to communicate the virtue of patience in the analysis of the results. There are a great many latent factors that can affect the perceived success or failure of project, some of them potentially within the control of the design team, others completely out of control and wholly unknown.

Due to the abundance of latent factors, any judgement about the efficacy of the design needs to be withheld until a sufficient quantity of data is collected about the performance of the solution in order to make a statistically valid adjudication.

Waiting, particularly for a team that has spent so much time and effort in seeing a project shift into production use, is challenging. People will want to check the status constantly, tracking the results of interactions in broad aggregations and trends with the speed and ferocity of laboratory mice pressing a lever for cheese to be dispensed.

It is in the best interests of the ML team to provide a crash course in statistical analysis for the decision makers in charge of this project. For a project such as a recommendation engine, explaining topics such as ANOVA, degrees of freedom in complex systems, RFM-based cohort analysis, and confidence intervals at a relatively high level (focusing mostly on how confident an analysis will be at short time-intervals (well, specifically, how confident it will not be) will help those people make informed decisions. Depending on the number of users that you have, the number of

platforms you're serving, and the frequency at which customers arrive at your site, it may take several days (or weeks) to collect enough data to make an informed decision about the impact that the project has to the company.

In the meantime, work studiously to assuage worries and tamp expectations that seeing a substantial rise in sales may or may not be directly attributable to the project. Only with careful and conscientious analysis of the data will anyone know what lift in engagement and revenue the new features may have.

4.2 Don't waste our time: critical meetings with cross-functional teams

Chapter 3 discussed the planning and experimentation phases of a project at length. One of the most important aspects to keep in mind during those phases (aside from the ML work itself) is in the communication *during those phases*. The feedback and evaluation that is received can be an invaluable tool to ensure that the MVP gets delivered on time and as correct as can be so that the full development effort can proceed.

Let's take a look at our Gantt chart that we generated in figure 4.6 for keeping track of the high-level progress for each of the teams' work throughout the phases. For the purposes of communication, however, we're only concerned with the top portion, as shown in figure 4.8 below.

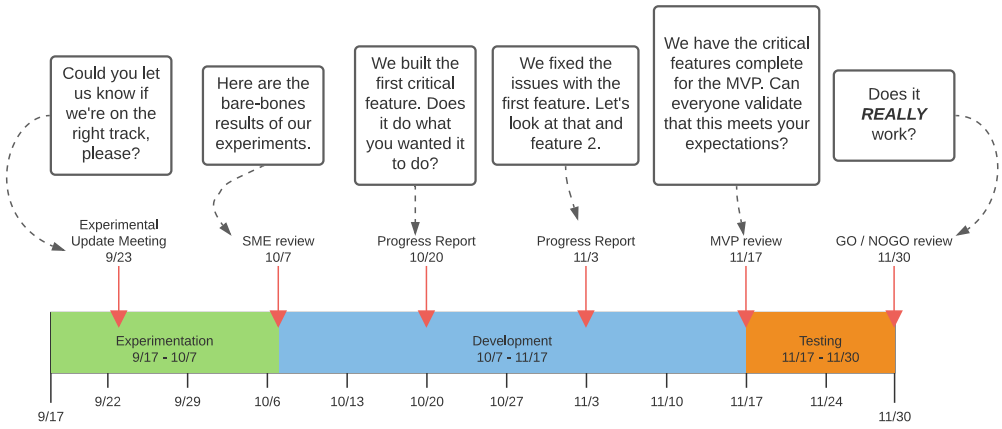


Figure 4.8 Meeting schedules throughout a project's schedule with realistic translations of what those meetings are actually for. The goal of setting progress report meetings (and, more specifically, tying these meetings to milestones of the project instead of chronological periodicity) is to get feedback and share the progress of the project. Eliminating periodic schedule meetings and focusing on presentations that have an actual point (unlike many periodic meetings which are largely pointless) keeps the group engaged on the discussion and doesn't risk meeting fatigue.

Depending on the type of project that is being built, there may be countless more meetings spread throughout the phases (as well as follow-up meetings for months after release to review metrics, statistics, and estimations of the resiliency of the solution). Even if

the development phase takes 9 months, for instance, the bi-weekly progress report meetings are just repetitive discussions on the progress of what has been accomplished during the previous sprint. We're going to break down these phases in detail below.

4.2.1 Experimental update meeting (Do we know what we're doing here?)

The meeting that the ML team dreads more than any other, and the meeting that everyone else is incredibly excited for. The ML team is interrupted in the midst of half-baked prototype implementations, unfinished research, and the state of what is in flux is at nearly peak entropy.

This meeting is perhaps the second most important meeting in a project, though. This is the 2nd to the last time for the team to have the ability to graciously raise a white flag in surrender if they've discovered that the project is untenable, will end up taking more time an money than the team has allocated, or is of such complexity that technologies will not be invented within the next 50 years to meet the requirements set forth. This is a time for honesty and reflection. It's a time to set one's ego aside and admit defeat it calls for it.

The overriding question that should be dominating this discussion is: "Can we actually figure this out?". Any other discussions or ideations about the project at this point are completely irrelevant. It is up to the ML team to report on the status of what they've discovered (without going too much into the weeds of model-specific details or additional algorithms that they will be employing for testing, etc). Namely:

- How is the progress towards the prototype coming along?
    - Have you figured out any of the things that you're testing yet?
    - Which one looks like it's the most promising so far?
    - Are you going to stop pursuing anything that you had planned to test?
    - Are we on track to having a prototype by the scheduled due date?
- What risks have you uncovered so far?
    - Are there challenges with the data that the Data Engineering team needs to be made aware of?
    - Are we going to need a new technology, platform, or tooling that the team isn't familiar with?
    - As of right now, do you feel as though this is a solvable problem for us?

Aside from these direct questions, there really isn't much else to discuss that will do anything other than waste the time of the ML team at this point. These questions are all designed as the vehicle of evaluating whether or not this project is tenable from a personnel, technology, platform, and cost perspective.

Raising a white flag – when admitting defeat is acceptable

Few 'serious ML' people ever like to admit defeat. When it's a junior person, fresh out of PhD program where research and experimentation can last months (or even years), the concept of admitting that the problem is unsolvable is something that will not ever enter their mind.

When developing an ML solution for a company, however, the question of 'possible to solve' for a problem is not one of whether it is possible to solve the problem, rather if it's possible to create a solution in a short enough time span so as not to waste too much money and resources. Eagerness at arriving at a solution can cloud the estimation of capabilities for even the most skilled ML practitioner. It is the wisdom of the experienced, those who have struggled for months on trying to just make something work, or had to maintain an incredibly fragile code base that had so many moving parts that troubleshooting a failure in it could take days, is what brings a temperance to projects. Desire for solving 'all the things' can be suppressed with the knowledge that the solution might not be right for this particular project, the company, or the teams involved in maintaining it.

The earlier this is realized the better. As mentioned earlier, the reluctance that creators have of abandoning their creations only grows as time and energy expenditure towards that creation increases. If you can call a halt to a project early enough (and hopefully recognize the signs that this is something that is not worth pursuing), you will be able to move onto something more worthwhile, instead of wading blindly through solutions that will end up creating little more than frustration, regret, and a complete loss of faith in ML at your company.

Provided that the answers are all positive in this meeting, work should commence in earnest (and hopefully there aren't any further interruptions to the work of the ML team so that they can meet the next deadline and present at the next meeting).

4.2.2 SME review (prototype review / can we solve this?)

By far the most important of the early meetings, the review of the experimentation phase's results is the final stage before a resourcing commit occurs. During this review session, the same questions should be asked as in the preceding meeting, except tailored to the estimations of capability for developing the full solution.

Resourcing commit? We're not building a space shuttle here.

It should come as no surprise that you, dear reader, are likely one of the most expensive humans in your engineering group. Not only is your salary probably quite substantial as a ratio to years of experience compared to, say, a software developer, but the tools that you use are, shall we say, pricey.

ML, by its very nature, requires data. A lot of data. It also requires an awful lot of transistors to make sense of that data to create something that can infer from that data useful things. So much data and so many transistors in most situations that the hardware (and sometimes the software, depending on what you're doing) is far more expensive for you to do your core job that nearly anyone else at your organization who is involved in writing code. While a back-end, front-end, or full-stack developer might need a small mock-up system consisting of a few virtual machines to test out an architecture, most of their work is done with unit-tested mock-up data on their laptops. You, on the other hand, could be dealing with TB's of data that required a 50-node Spark cluster just to execute a single run of a theory.

If you're going to start to work on something, this exceptional cost had better be worth it.

The main focus of this discussion is typically on the mocked-up prototype. For the case of the recommendation engine that we're discussing in this chapter, it may look like a synthetic wire frame of the website with a superimposed block of product image and labels associated with the product being displayed. It is always helpful, for the purposes of these demonstrations, to use real data. If you're showing a demonstration of recommendations to

a group of SME members, then show their data. Show the recommendations for their account (with their permission, of course!) and gauge their responses. Record each positive, but more importantly, each negative impression that they give.

What if it's terrible?

Depending on the project, the models involved, and the general approach to the ML task, the subjective rating of a prototype being 'terrible' can be either trivial to fix (properly tune the model, augment the feature set, etc.), or can be a complete impossibility (there data doesn't exist to augment the additional feature requests, the data isn't granular enough to solve the request, or in order to improve the prediction to the satisfaction of what is being asked for would require a healthy dose of magic since the technology to solve that problem doesn't exist yet).

It's important to quickly distill the reasons for why certain issues that are identified are happening. If the reasons are obvious and widely known as elements that can be modified by the ML team, then simply answer as such. "Don't worry, we'll be able to adjust the predictions so that you don't see multiple pairs of sandals right next to one another" is perfectly fine. But if the problem is of an intensely complex nature, "I really don't want to see Bohemian Maxi dresses next to Grunge shoes" (hopefully you will be able to quickly search what those terms means during the meeting), the response should be either thoughtfully articulated to the person, or recorded for a period of additional research, capped in time and effort to such research. At the next available opportunity the response may be along the lines of either, "we looked into that and since we don't have data that declares what style these shoes are, we would have to build a CNN model, train it to recognize styles, and create the hundreds of thousands of labels needed to identify these styles across our product catalog. That would likely take several years to build.", or "we looked into that and because we have the labels for every product, we can easily group recommendations by style type to give you more flexibility around what sort of product mixing you would like."

Make sure that you know what is and what is not possible before coming into a prototype review session, and if you encounter a request that you're not sure of, use the eight golden words of ML, "I don't know, but I'll go find out."

At the end of the demonstration, the entire team should have the ability to gauge whether the project is worth pursuing. There is no need to discuss technical details of implementation during this, however. Meetings for implementation details should always happen exclusively within the engineering teams with tech leads of each group meeting regularly to discuss issues of integration and status with the assistance of an architect. Having these technical discussions in a larger group of non-technical individuals will only serve to confuse and annoy them; it's best to keep the nerd-talk amongst nerds.

4.2.3 Development Progress Review(s) (is this thing actually going to work?)

These meetings are opportunities to 'right the ship' during development. The teams should be focusing on milestones such as these to show off the current state of the features being developed. It is very useful to use the same wire frame approach that was used in the experimentation review phase, using the same prototype data as well so that a direct comparison between earlier stages can be seen by the entire team. It helps to have a common frame of reference for the SME's to use in order to gauge the subjective quality of the solution in terms that they understand fully.

The first few of these meetings should be reviews of the actual development. While the details should never go into the realm of specific aspects of software development, model

tuning, or technical details of implementation, the overall progress of feature development should be discussed in abstract terms.

If at a previous meeting, the quality of the predictions was determined to be lacking in some way or another, an update and a demonstration of the fix should be shown to ensure that the problems were actually solved to the satisfaction of the SME group. It is not simply sufficient to claim that "the feature is complete and has been checked into master". Prove it instead. Show them the fix with the same data that they originally identified the problem.

As the project moves further and further along, these meetings should become shorter in duration and more focused on integration aspects of the project. By the time that the final meeting comes along for a recommendation project, the SME group should be looking at an actual demo of the website in a QA environment. The recommendations should be updating as planned through navigation, and validation of functionality on different platforms should be checked. As the complexity grows in these later stages, it can be helpful to push out builds of the QA version of the project to the SME team members so that they can evaluate the solution on their own time, bringing their feedback to the team at a regularly scheduled cadence meeting.

Unforeseen changes – welcome to the world of ML

To say that most ML projects are "complex" is a sad understatement. Some implementations, such as a recommendation engine, can be among the most complex code bases that a company has. Setting aside the modeling, which can be relatively complex, the interrelated rules, conditions, and usages of the predictions can be complex enough to almost guarantee that things will be missed or overlooked even in the most thorough planning phases.

The sometimes fitful, but frequently fungible nature of ML projects means that things will change. Perhaps the data doesn't exist or is too costly to create to solve a particular problem in the framework of what has been built up to that point. With a few changes in approach, the solution can be realized, but it will be at the expense of an increase in complexity or cost for another aspect of the solution. This is, both fortunately and unfortunately (depending on what needs to be changed), a part of ML.

The important thing to realize, understanding that things change, is that when a blocker arises, it should be communicated clearly to everyone that needs to know about the change. Is it something affecting the API contract for the serving layer? Talk to the front-end team; don't call a full team-wide meeting to discuss technical details. Is it something that affects the ability to filter out gender-specific recommendations? That's a big deal (according to the SMEs) and talking through solutions could benefit having every bright mind in the group together to solve the problem and explore alternatives.

When problems arise (and they will), just ensure that you're not doing a 'ninja-solve'. Don't silently hack a solution that seems like it will work and not mention it to anyone. The chances that you are created unforeseen issues later on is incredibly high and the impacts to the solution should be reviewed by the larger team.

4.2.4 MVP review (did you build what we asked for?)

By the time that you're having this meeting, everyone should be both elated and quite burned-out on the project. It's the final phase; the internal engineering reviews have been done, the system is functioning correctly, the integration tests are all passing, latencies have

been tested at large burst-traffic scale, and everyone involved in development is ready for a vacation.

The number of times I've seen teams and companies release a solution to production right at this stage is astounding. Each time that it's happened, they've all regretted it. Successful releases involve a stage after the engineering QA phase is complete in which the solution undergoes user acceptance testing (UAT). This stage is designed to measure the subjective quality of the solution, rather than the objective measures that can be calculated (statistical measures of the prediction quality) or the bias-laden subjective measure of the quality done by the SME's on the team who are, by this point, emotionally invested in the project.

UAT phases are wonderful. Well, that is, they're wonderful for everyone but the ML team. It's at this point in a project when their hard-earned solution finally sees the light of day in the form of feedback from a curated group of people. While all of the other work in the project is effectively measured via the Boolean scale of "works" / "doesn't work", the ML aspect is a sliding scale of quality that is dependent on the interpretations of the end-consumer of the predictions. In the case of something as subjective as the relevancy of recommendations to an end-user, this scale can be remarkably broad. In order to gather relevant data to create adjustments, an effective technique that can be employed is a survey (particularly for a project as subjective as recommendations). Providing feedback based on a controlled test with a number ranking of effective quality can allow for standardization in the analysis of the responses, giving a broad estimation of what additional elements need to be added to the engine or settings that need to be modified.

The critical aspect of this evaluation and metric collection is to ensure that the members of the evaluation of the solution are not in any way vested in the creation of it, nor are they aware of the inner workings of the engine. Having foreknowledge of the functionality of any aspect of the engine may taint the results, and certainly if any of the team members of the project were to be included in the evaluation, the review data would be instantly suspect.

When evaluating UAT results, it is important to use appropriate statistical methodologies to normalize the data. Scores (particularly those on a large numeric scale) need to be normalized within the range of scores that each user provides to account for review bias that most people have (tending to either score maximum or minimum for some, other gravitating around the mean value, and others being overly positive in their review scores). Once normalized, importance for each of the questions and how they impact the overall predictive quality of the model can be assessed, ranked, and a determination of feasibility to implement be conducted. Provided that there is enough time, that the changes are warranted, and the implementation is of a low enough risk to as not to require an additional full round of UAT, these changes may be implemented in order to create the best possible solution upon release.

Should you ever find yourself having made it through a UAT review session without a single issue being found, either you're the luckiest team ever, or the evaluators are completely checked-out. This is actually quite common in smaller companies where nearly everyone is aware of an invested in a solution. It can be quite helpful to bring in outsiders in this case to validate the solution (provided that the project is not something, for instance, akin to a fraud detection model or anything else of extreme sensitivity). Many companies

that are successful in the space of building solutions for external-facing customers typically engage in alpha or beta testing periods of new features for this exact purpose; to elicit high-quality feedback from customers that are invested in their products and platforms. Why not use your most passionate end-users (either internal or external) to give feedback? After all, they're the ones who are going to be using what you're building.

4.2.5 Pre-prod review (We really hope we didn't screw this up)

The end is nigh for the project. The final features have been added from UAT feedback, QA checks have all passed, the solution has been running for over a week without a single issue with it in a stress-testing environment. Metrics are set up for collecting performance, analytics reporting datasets have been created, ready to be populated for measuring the success of the project.

The last thing to do is to 'ship it' to production.

It's best to meet one final time, but not for self-congratulations. This final meeting should be structured as a project-based retrospective and analysis of features. Everyone at this meeting, regardless of area of expertise and level of contribution to the final product should be asking the same thing: "Did we build what we set out to build?". In the effort to answer this question, the original plans should be compared to the final designed solution. Each feature that was in the original design should be gone through and validated that it functions in real-time from within the QA (testing) environment. Do the items get filtered out when switching between pages? If multiple items are added to the cart in succession, do all of those related products get filtered, or just the last one? What if items are removed from the cart – do the products stay removed from the recommendations? What happens if some navigates the site and adds a thousand products to the cart and then removes all of them?

Hopefully all of these scenarios have been tested long before this point, but it's an important exercise to engage in with the entire team to ensure that the functionality is conclusively confirmed to be implemented correctly. After this point, there's no going back; once it's released to production, it's in the hands of the customer, for better or for worse. We'll get into how to handle issues in production in later chapters, but for now, think of the damage to the reputation of the project if something that is fundamentally broken is released. It's this last pre-production meeting where concerns and last-minute fixes can be planned before the irrevocable production release.

You can have the congratulatory party later – 2 months later, after sales and engagement lifts have been confirmed through statistical analysis.

4.3 Setting limits on your experimentation

We've gone through the exhausting slog of preparing everything that we can up until this point for the recommendation engine project. Meetings have been attended, concerns and risks voiced, plans for design have been conducted, and, based on the research phase, we have a clear set of models to try out. It's finally time to play some jazz, get creative, and see if we can make something that's not total garbage.

Before we get too excited though, it's important to realize that, with all other aspects of ML project work, we should be doing things in moderation and with a thoughtful purpose

behind what we're doing. This applies more so to the experimentation phase than any other aspect of the project – primarily because this is one of the completely siloed off phases of the project. This is the "nerds going to the lab to do nerd stuff" phase in which theories are scratched together, code is written, code is deleted, and written again. If you're not careful with experimentation, you can spend so much time doing it that you run the risk of compromising the entire project.

What might we do with this personalized recommendation engine if we had all of the time and resources in the world? Would we research the latest white papers (vague though they may be) and try to implement a completely novel solution (likely taking more than a year to implement)? Would we think about building a broad-ensemble of recommendation models to cover all of the ideas that we have (let's do a collaborative filtering model for each of our customer cohorts based on their customer lifetime value scores for propensity and their general product-group affinity, then merge that with an FP-Growth market basket model to populate sparse predictions for certain users, et al.)? Perhaps we would build a graph-embedding to a deep learning model that would find relationships in product and user behavior that could potentially create the most sophisticated and accurate predictions that are achievable. All of these are neat ideas and could potentially be worthwhile if the entire purpose of our company was to recommend items to humans. However, these are all *very expensive to develop* in the currency that most companies are most strapped for: time.

We need to understand that time is a finite resource, as is the patience of the business unit that requested the solution. As we discussed before in Chapter 3, section 3.2.2, the scoping of the experimentation is tied directly to the resources available: how many Data Scientists there are on the team, what the number of options that we are going to attempt to compare against one another, and, most critically, the time that we have to complete this in. The final limitations that we need to control for, knowing that there are limited constraints on time and developers, is that there is only so much that can be built within an MVP phase.

It's tempting to want to fully build out a solution that you have in your head and see it work exactly as you've designed it. This works great for internal tools that are helping your own productivity or projects that are internal to the ML team. But pretty much every other thing that an ML Engineer or Data Scientist is going to work on in their careers has a customer aspect to it, be it an internal or external one. This will mean that you have someone else depending on your work to solve a problem. They will have a nuanced understanding of the needs of the solution that might not align with your assumptions.

Not only is it, as we have mentioned earlier, incredibly important to include them in the process of aligning the project to the goals, but it's potentially very dangerous to fully build out a tightly coupled and complex solution without getting their input on the validity of what you're building to the issue of solving the problem.

The way of solving this issue of involving the SMEs in the process is to set boundaries around prototypes that you'll be testing.

4.3.1 Set a time limit

Perhaps one of the easiest ways to stall or cancel a project is by spending too much time and effort into the initial prototype. This can happen for any number of reasons, but most of them, I've found, are due to poor communication amongst a team, incorrect assumptions by

non-ML team members about how the ML process works (refinement through testing with healthy doses of trial, error, and reworking mixed in), or an inexperienced ML team assuming that they need to have a 'perfect solution' before anyone sees their prototypes.

The best way to prevent this confusion and complete wasting of time is to set limits on the time allotted for experimentation surrounding vetting of ideas to try, which, by its very nature, will eliminate the volume of code that is written at this stage. It should be very clear to all members of the project team that the vast majority of the ideas expressed during the planning stages are not going to be implemented for the vetting phase; rather, in order to make the crucial decision about which implementation to go with, the bare minimum amount of the project should be tested. Below, in figure 4.9, is shown the most minimalistic amount of implementation that needs to be done to achieve the goals of the experimentation phase. Any additional work, at this time, does not serve the need at the moment: to decide on an algorithm that will work well at scale, at cost, and meets objective and subjective quality standards.

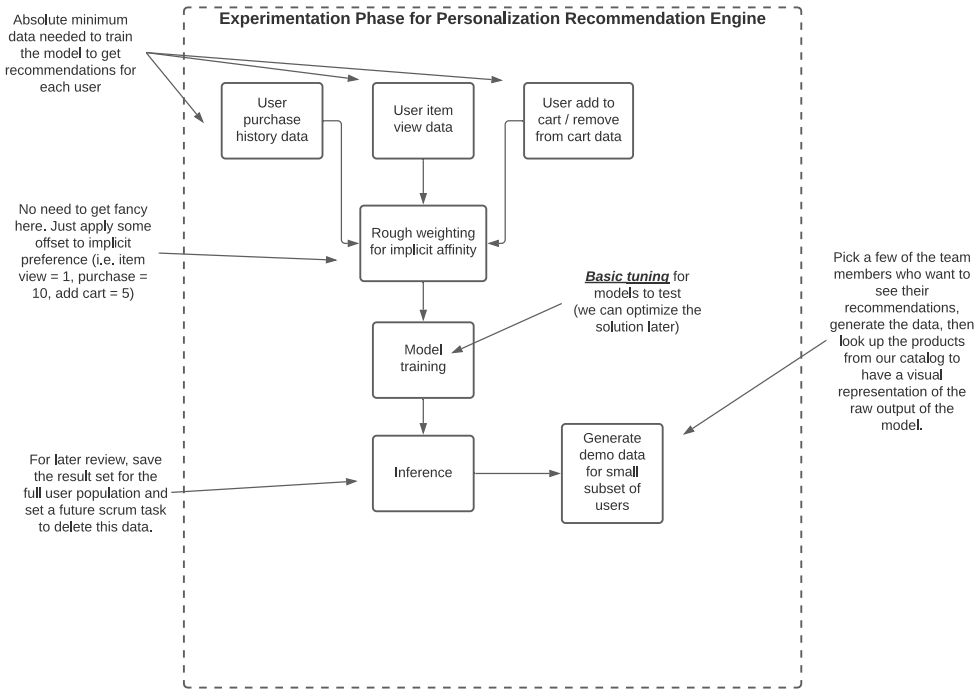


Figure 4.9 The experimental phase scope of work for the ML team. In order to perform the minimum required work to meet the goals of comparing each of the different models available, the only elements required are to acquire the affinity data, to train the model, and run inference on the testing subset of users. Work beyond this is simply wasted effort (due to the fact that additional work done on the non-selected model implementations is going to be completely thrown away).

In comparison, figure 4.10 shows a simplified view of what some of the core features might be based on the initial plan from the planning meeting.

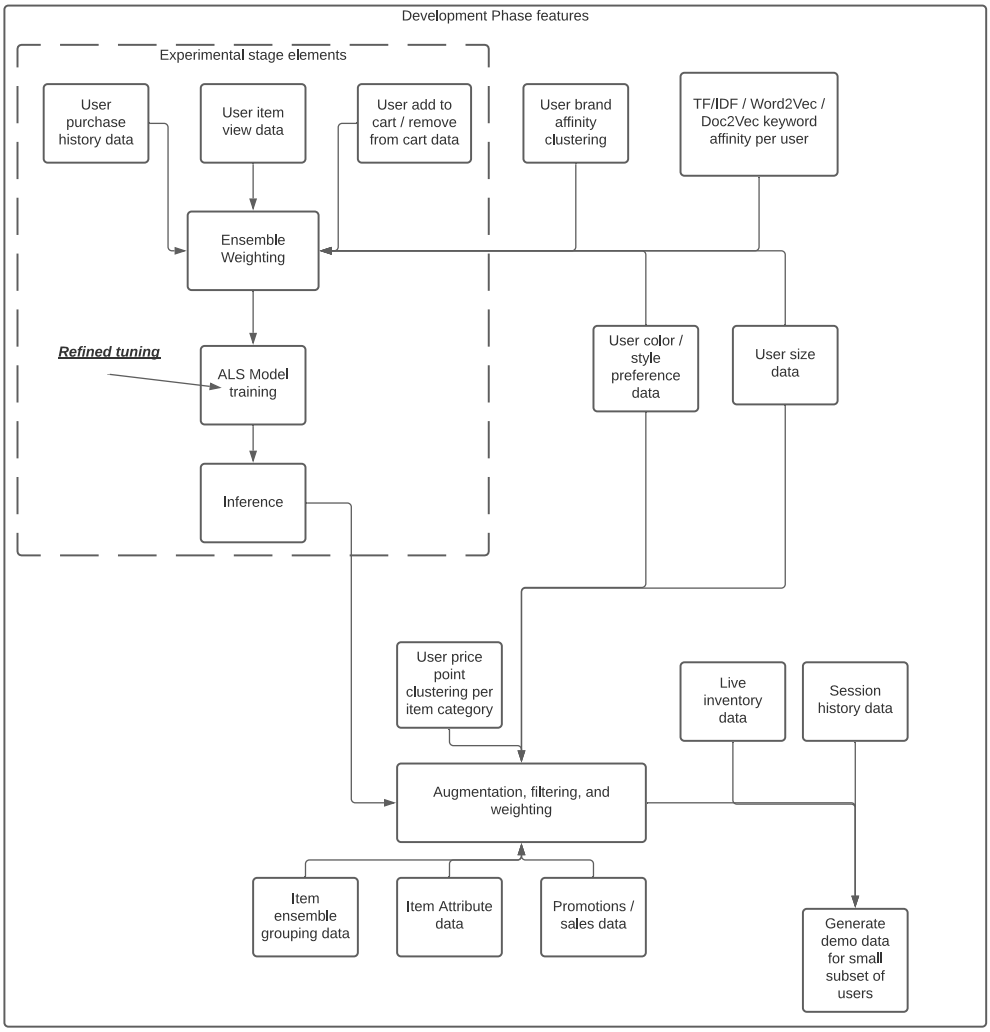


Figure 4.10 An example of the features and their interrelationships for the project as of the planning phase meetings. This diagram is more applicable to a roadmap of features to be built throughout the sprints of the development phase. Exerting any effort outside of the experimental stage element box in the upper left is wasted effort until the model type and refined feature ideas can be generated after seeing the initial experimental results.

Based on the comparison of these two figures, figure 4.9 and figure 4.10, it should be easy to imagine the increasing scope of work that is involved in the transition from the first plan to the second. There are entirely new models that need to be built, a great deal of dynamic run-specific aggregations and filtering that need to be done, custom weighting to be incorporated, and potentially dozens of additional data sets to be generated. None of these elements solves the core problem at the boundary of experimentation: which model should we go with for developing?

Limiting the time to make this decision will prevent (or, at least minimize) the natural tendency of most ML practitioners to want to build a solution, regardless of plans that have been laid out. Sometimes forcing less work to get done is a good thing, for the cause of reducing churn and making sure the right elements are being worked on.

A note on experimental code quality

Experimental code should be a little 'janky'. It should be scripted, commented-out, ugly, and nigh-untestable. It should be a script, filled with charts, graphs, print statements, and all manner of bad coding practices.

It's an experiment, after all. If you're following a tight timeline to get an experimental decision-of-action to be made, you likely won't have time to be creating classes, methods, interfaces, enumerators, factory builder patterns, couriering configurations, etc. You're going to be using high-level APIs, declarative scripting, and a static data set.

Don't worry about the state of the code at the end of experimentation. It should serve as a reference for development efforts in which proper coding is done (and under no circumstances should experimental code be expanded upon for the final solution), wherein the team is building maintainable software, using standard software development practices.

But for this stage, and only this stage, it's usually ok to write some pretty horrible-looking scripts.

4.3.2 Can you put this into production? Would you want to maintain it?

While the primary purpose of an experimentation phase, to the larger team, is to make a decision on the predictive capabilities of a model's implementation, one of the chief purposes internally, amongst the ML team, is to determine if the solution is tenable for the team.

The ML team lead, architect, or senior ML person on the team should be taking a very close look at what is going to be involved with this project, asking some difficult questions, while producing some very honest answers to them. Some of the most important ones being:

- How long is this solution going to take to build?
- How complex is this code base going to be?
- How expensive is this going to be to train based on the schedule it needs to be retrained at?
- How much skill is there on my team to be able to maintain this solution? Does everyone know this algorithm / language / platform?
- How quickly will we be able to modify this solution should something dramatically change with the data that it's training or inferring on?
- Has anyone else reported success with using this methodology / platform / language / API? Are we re-inventing the wheel here, or are we building a square wheel?
- How much additional work is the team going to have to do to make this solution work

while meeting all of the other feature goals?
- Is this going to be extensible? When the inevitable version 2.0 of this is requested, will we be able to enhance this solution easily?
- Is this testable?
- Is this auditable?

There have been innumerable times in my career where I've either been the one building these prototypes or been the one asking these questions while reviewing someone else's prototype. Although an ML practitioner's first reaction to seeing results is frequently "let's go with the one that has the best results", many times the 'best one' is the one that ends up being either nigh-impossible to fully implement or is a nightmare to maintain. It is of paramount importance to weigh these 'future thinking' questions about maintainability and extensibility, whether it is regarding the algorithm in use, the API that calls the algorithm, or the very platform that it's running on. If you take the time to properly evaluate the production-specific concerns of an implementation, instead of simply the predictive power of the model's prototype, it can mean the difference between a successful solution and vaporware.

4.3.3 TDD vs RDD vs PDD vs CDD for ML projects

It seems as though there are an infinite array of methodologies to choose from when thinking about how to develop software. From waterfall, to the agile revolution (and all of the myriad flavors of that), there are benefits and drawbacks to each. We're not going to be discussing the finer points of which development approach might be best for particular projects or teams. There have been absolutely fantastic books published that are great resources for exploring these topics in depth (Becoming Agile and Test Driven are notable ones), which I highly recommend reading to improve the development processes for ML projects. What is worth discussing here, however, is four general approaches to ML development (one being a successful methodology, the others being cautionary tales).

TEST DRIVEN DEVELOPMENT (TDD) OR FEATURE DRIVE DEVELOPMENT (FDD)

While pure TDD is incredibly challenging (if not impossible) to achieve for ML projects (mostly due to the non-deterministic nature of models themselves), and a pure FDD approach can cause significant rework during the duration of the project, most successful approaches to ML project work embrace aspects of both of these development styles. Keeping work incremental, adaptable to change, and focused on modular code that is not only testable but focused entirely on required features to meet the project guidelines is a proven approach that helps to deliver the project on time while also creating a maintainable and extensible solution.

These Agile approaches will need to be borrowed from and adapted in order to see an effective development strategy that works for not only the development team, but also for an organization's general software development practices, as well as the specific design needs for different project work can dictate slight different approaches to how a specific project is implemented.

Why would I want to use different development philosophies?

When discussing ML as a broad topic, one runs the risk of over-simplifying an incredibly complex and dynamic discipline. Since ML is used for such a wide breadth of use cases (as well as having such a broad set of skills, tools, platforms, and languages), the magnitude of difference in complexity amongst various projects is truly astounding.

For a project as simple as "we would like to predict customer churn", a TDD-heavy approach can be a successful way of developing a solution. A model and inference pipeline for implementations of churn prediction models is typically rather simple (the vast majority of the complexity is in the data engineering portion), and as such, modularizing code and building the code base in a way that each component of the data acquisition phase can be independently tested can be very beneficial to an efficient implementation cycle and an easier to maintain final product.

On the other hand, a project that is as complex as, say, an ensemble recommender engine that uses real-time prediction serving, has hundreds of logic-based reordering features, employs the predictions from several models, and has a large multi-discipline team working on it could greatly benefit from using the testability components of TDD, but throughout the project use the principles of FDD to ensure that only the most critical components are developed as needed can help to reduce feature sprawl.

Each project is unique and the team lead or architect in charge of the implementation from a development perspective should plan accordingly with how they would like to control the creation of the solution. With the proper balance in place of best practices from these proven standards of development, a project can hit its required feature-complete state in a position of being at the lowest-risk-to-failure point so that the solution is stable and maintainable while in production.

PRAYER DRIVEN DEVELOPMENT (PDD)

At one point, all ML projects were this. In many organizations that are new to ML development, it still is. Before the days of well-documented high-level APIs to make modeling work easier, everything was a painful exercise in hoping that what was being scratched and cobbled together would work at least well enough that the model wouldn't detonate in production. That hoping (and praying) for things to 'just work please' isn't what I'm referring to here, though.

What I'm facetiously alluding to with this title is rather the act of frantically scanning for clues about how to solve a particular problem by following either bad advice on internet forums, or following along with someone (who likely doesn't have much more actual experience than the searcher) who has posted a blog covering a technology or application of ML that seems somewhat relevant to the problem that they're trying to solve, only to find out, months later, that the magical solution that they were hoping for was nothing more than fluff.

Prayer driven ML development is the process of seeking out truth to problems that one doesn't know how to solve into the figurative hands of some all-knowing person who has solved it before, all in the goal of eliminating the odious task of doing proper research and evaluation of technical approaches to the problem. Taking such an easy road rarely ends well; with broken code bases, wasted effort ("I did what they did – why doesn't this work?!!") and, in the most extreme cases, project abandonment, this is a problem and development anti-pattern that is growing in magnitude and severity in recent years.

The most common effects that I see happen from this approach of ML 'copy-culture' are either that people who embrace this mentality either want to use a single tool for every

problem (yes, XGBoost is a solid algorithm. No, it's not applicable to every supervised learning task) or that they want to try only the latest and greatest fad ("I think we should use TensorFlow and Keras to predict customer churn").

> ***If all you know is XGBoost, then everything looks like a gradient boosting problem.***

Limiting one's self in this manner (not doing research, not learning or testing alternate approaches, and restricting experimentation or development to a limited set of tools), the solution will reflect these limitations and self-imposed boundaries. In many cases, latching onto a single tool or a new fad and forcing it onto every problem creates sub-optimal solutions or, more disastrously, forces one to write far more lines of unnecessarily complex code in order to fit a square peg into a round hole.

A good way of detecting if the team (or yourself) is on the path of PDD is to see what is planned for a prototyping phase for a project. How many models are being tested? How many frameworks are being vetted? If the answer to either of these is, "one", and no one on the team has solved the particular problem several times before, then you're doing PDD. And you should stop.

CHAOS DRIVEN DEVELOPMENT (CDD)

Also known as 'cowboy development', CDD is a process of skipping the experimentation and prototyping phases altogether. Espoused as "good programming processes" and "pure Agile development", the approach of building ML in an as-needed basis during project work is fraught with peril. As modification requests and new feature demands arise through the process of developing a solution, the sheer volume of rework (sometimes from scratch) slows the project to a crawl. By the end (if it makes it that far), the fragile state of the ML team's sanity will entirely prevent any future improvements or changes to the code due to the spaghetti nature of the implementation.

RESUME DRIVEN DEVELOPMENT (RDD)

By far the most detrimental development practices, designing an over-engineered 'show-off' implementation to a problem is one of the primary leading causes of projects being abandoned after they are in production.

RDD implementations are generally focused around a few key characteristics:

- A novel algorithm is involved
    - Unless it's warranted due to the unique nature of the problem
    - Unless multiple experienced ML experts agree that there is not an alternative solution
- A new framework for executing the project's job (with features that serve no purpose in solving the problem) is involved
    - There's not really an excuse for this nowadays
- A blog post (or series of blog posts) about the solution are being written during development

    - This should raise a healthy suspicion amongst the team
    - There will be time to self-congratulate after the project is released to production, has been verified to be stable for a month, and impact metrics have been validated.
- An overwhelming amount of the code is devoted to the ML algorithm as opposed to feature engineering or validation
    - For the vast majority of ML solutions, the ratio of feature engineering code to model code should always be > 4x.
- An abnormal level of discussion in status meetings about the model, rather than the problem to be solved
    - We're here to solve a business problem, aren't we?

This isn't to say that novel algorithm development or incredibly in-depth and complex solutions aren't called for. They most certainly can be. But they should only be pursued if all other options have been exhausted. In the case of the example that we've been reviewing throughout this chapter, if someone were to go from a position of having nothing in place at all to proposing a unique solution that has never been built before, objections should be raised. This development practice and the motivations behind it are not only toxic to the team that will have to support the solution, but will poison the well of the project and almost guarantee that it will take longer, be more expensive, and do nothing apart from pad the developer's resume.

4.4 Business rules chaos

As part of our recommendation engine that we've been building throughout this chapter (or, at least, speaking of the process of building), a great many features crept up that were implemented that augmented the results of the model. Some of these were to solve particular use cases for the end-result (collection aggregations to serve the different parts of the site and app for visualization purposes, for instance), while others were designed for contractual obligations to vendors. The most critical ones were to protect the users from offense or to filter inappropriate content. All of these additional nuances to ML I like to refer to as "business rules chaos". They're the specific restrictions and controls that are both incredibly important but are also frequently the most challenging aspects of a project to implement correctly. However, failing to plan for them accordingly (or failing to implement them entirely) is an almost guarantee that your project will be shelved before it hits its expected release date.

4.4.1 Embracing chaos by planning for it

Let's pretend for a moment that the ML team that has been working on the MVP for the recommendation engine didn't realize that the company sold 'sensitive products'. This can be quite understandable, since most e-commerce companies sell a lot of products and the ML team are not product specialists. They may be users of the site (that sweet, sweet employee discount!), but they certainly aren't likely to be intimately familiar with everything that's

sold. Since they weren't familiar that there were items that could potentially be offensive to be a part of a recommendations, they failed to identify these items and filter them out of their result sets.

There's nothing wrong with missing a detail like this. In my experience, details like this always come up in complex ML solutions. The only way to plan for it is to expect things like this to come up; to architect the code base in such a way that there are proverbial 'levers and knobs' (functions or methods that can be applied or modified through passed-in configurations) such that implementing a new restriction doesn't require a full code rewrite or weeks of adjustments to the code base to implement.

When in the process of developing a solution, a lot of ML practitioners tend to think mostly about the quality of the model's predictive power above all other things. Countless hours of experimentation, tuning, validating, and reworking of the solution is done in the pursuit of attaining a mathematically optimal solution that will solve the problem best in terms of validation metrics. Because of this, it can be more than slightly irritating to find out that, after having spent so much time and energy in building an ideal system that there are additional constraints that need to be placed onto the model's predictions.

These constraints exist in almost all systems (either initially or eventually if the solution is in production for long enough) that have predictive ML at their core. It may be that there are legal reasons to filter or adjust the results in a financial system. There could, perhaps, be content restrictions on a recommendation system based on preventing a customer from taking offense by a prediction (trust me, you don't want to explain to anyone why a minor was recommended an adult-oriented product). Whether it be for financial, legal, ethical, or just plain old common-sense reasons, inevitably something is going to have to change with the raw predictions from most ML implementations.

It's definitely a best practice to understand what these restrictions are going to be before you start spending too much time in development of a solution. Knowing restrictions ahead of time can influence the overall architecture of the solution, the feature engineering, and allowing for controlling the method in which an ML model learns the vector. It can save the team countless hours of adjustment and eliminate costly-to-run and difficult-to-read code bases that are filled with never-ending chains of if/elif/else statements to handle post-hoc corrections to the output of the model.

For the recommendation engine project that we've been discussing, there are likely a lot of rules that need to be added to a raw predictive output from an ALS model. As an exercise, let's revisit the earlier development phase work component diagram. Figure 4.11 below shows the elements of the planned solution that are specifically intended to enforce constraints on the output of the recommendations. Some are absolutely required (contract requirement elements, as well as filters that are intended to cull out products that are inappropriate for certain users), while others are ideas that the project team suspect are going to be heavily influential in getting the user to engage with the recommendation.

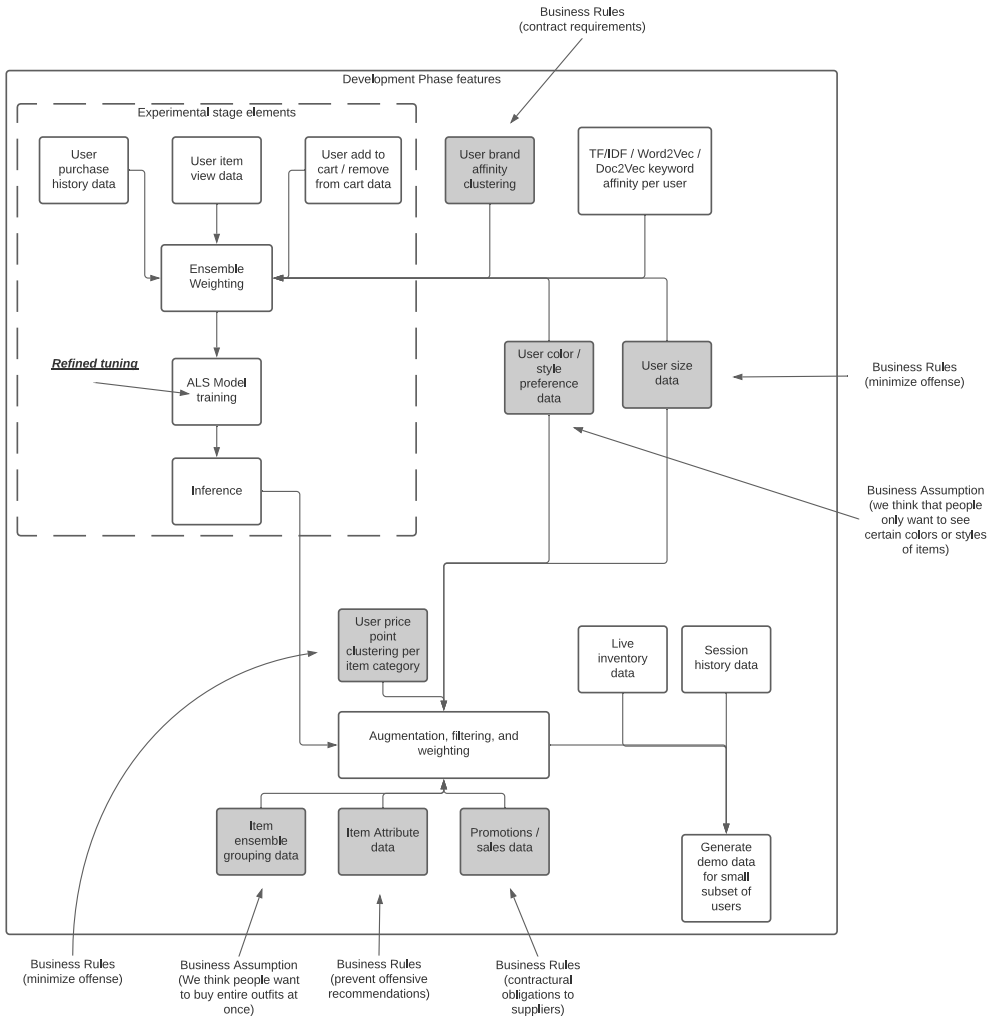


Figure 4.11 Identification of business contextual requirements for the recommendation engine project. While some are absolutely required features to be added, others may be optional (or eligible for testing through AB comparison). Knowing these requirements beforehand can inform the best places to bring them into the solution and to ensure that the required amount of allotted time and scoping can be provided to ensure that they are completed before UAT testing phases.

Figure 4.11 shows the locations, but more importantly, the type of business restriction to the model. In the planning phases, after experimentation and before full development begins, it is worthwhile to identify and to classify each of these features. For the absolutely required aspects (shown above as "Business Rules"), these must be planned within the scope

of the work and built as an integral part of the modeling process. Whether they be constructed in such a way to be tune-able (through weights, conditional logic, or Boolean switches) aspects of the solution or not is up to the needs of the team, but they should be considered as essential features that are not up for being seen as optional or testable features. The remaining aspects of rules (marked in figure 4.11 as "Business Assumption") can be handled in a number of different ways. Firstly, they could be prioritized as testable features (wherein configurations will be built that allow for AB testing different ideas for fine-tuning the solution), or they can be seen as 'future work' that is not part of the initial MVP release of the engine, simple implemented as placeholders within the engine that can be modified with ease at a later time.

4.4.2 Human in the loop design

Whichever approach works best for the team (and particularly the ML developers that are working on the engine), the important fact to keep in mind is that these sorts of restrictions to the model output should be identified early and allowances be made for them to be mutable for the purposes of changing their behavior, if warranted. The last thing that you want to build for these requirements, though, is hard-coded values in the source code that would require a modification to the source code in order to test. It's best to approach these items in a way that you can empower the SME's to modify the performance, to rapidly change the behavior of the system without having to take it down for a lengthy release period, and ensure that there are controls established that restrict the ability to modify these without going through appropriate validation procedures.

4.4.3 What's your backup plan?

What happens when there's a new customer? What happens with recommendations for a customer that has returned after having not visited your site in more than a year? What about for a customer who has only viewed one product and is returning to the site the next day?

Planning for sparse data isn't a concern just for recommendation engines, but it certainly impacts their performance more so than other applications of ML. In any project, there should be thought provided and a plan made for a 'fall back plan'. This safety mode can be as complex as using registration information, IP geolocation tracking to pull aggregated popular products from the region that the person is logging in from (hopefully they're not using a VPN) or can be as simple as generic popularity rankings from all users. Whichever methodology is chosen, it's important to have a safe set of generic data to fall back to if personalization data sets are not available for the user.

This general concept applies to many use cases, not just recommendation engines. If you're running predictions for some use case where you don't have enough data to fully populate the feature vector, this could be a similar issue to having a recommendation engine cold start problem. There are a number of ways to handle this issue, but at the stage of planning, it's important to realize that this is going to be a problem and that there should be form of fallback in place in order to produce some level of information to a service expecting data to be returned.

4.5 How to talk about results

Explaining how ML algorithms work to a layperson is challenging. Analogies, thought-experiment-based examples, and comprehensible diagrams to accompany them are difficult at the best of times (when someone is asking for the sake of genuine curiosity). When the questions are poised by members of a cross-functional team that are trying to get a project released, it can be even more challenging and mentally taxing (since they have a basis for expectation regarding what they want the 'black box' to do). When those same team members are finding fault with the results or quality of the predictions, aggravated at the subjectively poor results, this adventure into describing the functionality and capabilities of the algorithms that have been chosen can be remarkably stressful.

In any project's development, be it at the early stages of planning, during prototype demonstrations, or even at the conclusion of the development phase while the solution is undergoing UAT assessment, these questions will invariably come up. The questions below are specific to the example recommendation engine that we've been discussing, but I can assure you that alternative forms of these questions can be applied to any ML project; from a fraud prediction model, to a threat detection video classification model.

"Why does it think that I would like that? I would never pick something like that for myself!"

"Why is it recommending umbrellas? That customer lives in the desert. What is it thinking?!"

"Why does it think that this customer would like t-shirts? They only buy haute couture."

The flippant answer to all of these questions is simple: "It doesn't think. The algorithm only 'knows' what we 'taught' it." (pro tip: if you're going to use this line, don't, for the sake of further tenure in your position, place emphasis on those quoted elements when delivering this line. On second thought, don't talk to colleagues like this, even if you're annoyed at having to explain this concept for the 491st time during the project) The acceptable answer, conveyed in a patient and understanding tone, is one of simple honesty: "We don't have the data to answer that question." It's best to exhaust all possibilities of feature engineering creativity before claiming that, but assuming that you have, it's really the only actual answer that is worth giving.

What has been successful for me, in the past (and present) is to explain this issue and the root cause of it through articulating the concept of cause and effect; but in a way that is relating to the ML aspect of the problem.

In figure 4.12 below, a helpful visualization for explaining the concept of what ML can, but also, more importantly, what it cannot do is shown.

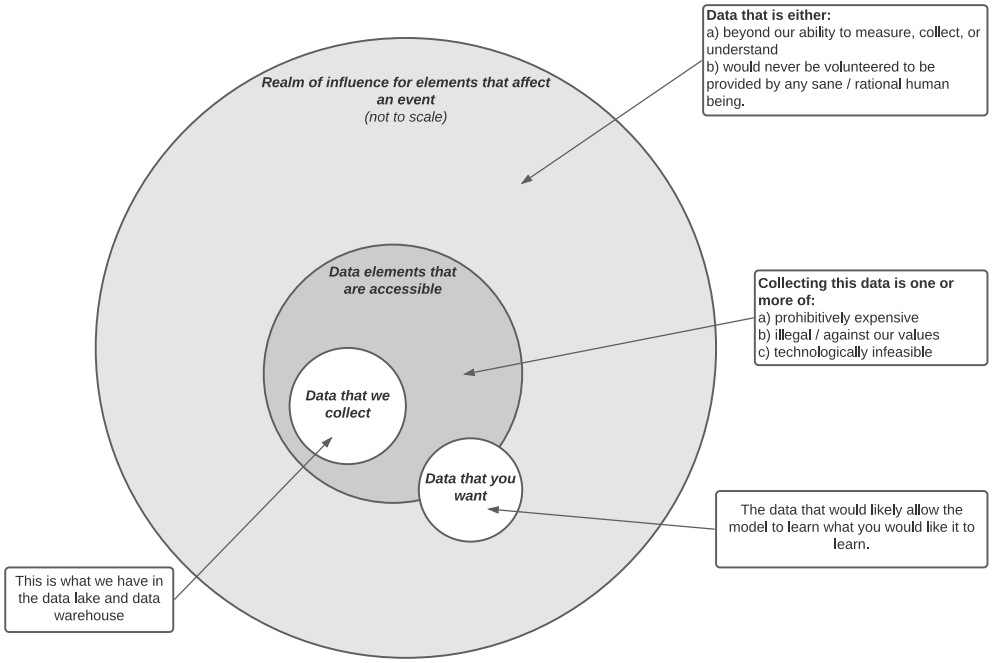


Fig 4.12 Realms of data for ML. A useful visualization to teach non-ML people about the limitations of algorithms with respect to the data that is collected about entities. If the desire for a particular output from a model is outside the realm of data that can be collected, there is little chance that even the most sophisticated algorithm or implementation of one will be able to discern the relationships desired.

As figure 4.12 shows, the data that the person in the review meeting is asking for is simple beyond the capability to acquire. Perhaps the data that would inform someone's subjective preference for a pair of socks is of such a personal nature that there is simply no way to infer this or collect this information. Perhaps, in order to have the model draw the conclusion that is being asked for, the data to be collected would be so complex, expensive to store, or challenging to collect that it's simply not within the budget of the company to do so.

When an SME at the meeting asks, "why didn't this group of people add these items to their cart if the model predicted that these were so relevant for them?", there is absolutely no way you can answer that. Instead of dismissing this line of questioning, which will invariably lead to irritation and frustration from the asker, simply posit a few questions of your own while explaining the view of reality that the model can 'see'. Perhaps the user was shopping for someone else? Perhaps they're looking for something new that they were inspired by from an event that we can't see in the form of data. Perhaps they simply just weren't in the mood. There is a staggering infiniteness to the latent factors that can influence the behavior of events in the 'real world' and even were you to collect all of the knowable information and metrics about the observable universe, you would still not be able to predict,

reliably, what is going to happen, where it's going to happen, and why it will or will not happen as such. It's understandable for that SME to want to know why the model behaved a certain way and the expected outcome (the user giving us money for our goods) didn't happen; as humans, we strive for explainable order.

> ***Relax. We, like our models, can't be perfect.***
> ***The world's a pretty chaotic place. We can only hope to guess right at what's going to happen more than we guess wrong.***

Explaining limitations in this way (that we can't predict something that we don't have information to train on) can help, particularly at the outset of the project, to dispel assumed unrealistic capabilities of ML to laypeople. Having these discussions within the context of the project and how the data involved relates to the business can be a great tool in eliminating disappointment and frustration later on as the project moves forward through milestones of demos and reviews.

Explaining expectations clearly, in plain-speak, particularly to the project leader, can be the difference between an acceptable risk that can be worked around in creative ways and a complete halt to the project and abandonment due to the solution not doing what the business leader had in their mind. As so many wise people have said throughout the history of business, "it's always best to under-promise and over-deliver."

Resist Plato

Although at times it can feel as though, when speaking about things such as 'models', 'data', 'machine learning', 'algorithms', et al. that you are living through the Allegory of the Cave. Resist the urge to think of yourself or your team as the members of the cave dwellers who have 'stepped into the light' and are merely returning to the cave to show the miraculous imagery of full color and the 'real world'. You may know more about ML than the uninitiated, but taking the stance of being 'the enlightened' and adopting a superior tone when explaining concepts to other team members will only breed derision and anger, just as with the returning group who attempted to drag the others to the light.

You will always have more success explaining concepts in familiar terms to your audience and approaching complex topics through allegory and examples rather than defaulting to exclusionary dialogue that will not be fully understood by others on the team who are not familiar with the inner workings of your profession.

4.6 Summary

- Including consistent, targeted, and relevant communication with SMEs and business sponsors of a project will dramatically reduce the chances of project cancellation and help to ensure that you're building a solution that will actually get used.
- Inclusive communication to business owners (keeping the technical details out of meetings and presentations) will allow SMEs to contribute and will typically make for a more successful solution, reducing the risk of project abandonment, scope creep and non-productive interactions amongst the cross-functional team.
- Planning and discussing business-specific restrictions about what can and cannot be

allowed to come out of a predictive solution can prevent shoe-horned and fragile code bases.

- Discussing limitations of the technology (and the data) in a clear and concise way early enough can help set expectations of what is and is not possible with ML, leaving the project owner with few surprises during delivery.