CLEAN MACHINE LEARNING CODE

Moussa Taifi

Clean Machine Learning Code

Moussa Taifi

This book is for sale at http://leanpub.com/cleanmachinelearningcode

This version was published on 2020-10-18

Contents

To Christina and Theo for their loving support while I was writing this book.

Chapter 1 - Clean Machine Learning Code Fundamentals

The Future of Machine Learning Code

We need better machine learning engineers. You are reading this book because you want to be a better machine learning engineer, data scientist, data analyst, or data engineer. Maybe you are also managing teams that are responsible for data analytics or machine learning applications. The whole machine learning industry relies on better machine learning engineers.

In a sense, this book is about rehabilitation from the short-sightedness of quarterly business goals. We must recognize that businesses are addicted to quarterly results. This addiction trickles down to all ranks of software engineers. ML engineers are not immune to this business pressure, and results must come at any price. This book attempts to help the reader identify the role of business pressure on their codebases, and on the mindset and attitudes that lead to ML software disasters. This book also attempts to provide mental and technical tools to kick the habit of mindless code patching to fit the next functional feature into the upcoming launch.

This business pressure is especially relevant for ML engineers that move in the blink of an eye from business explorations to Minimum Viable Product (MVP) to production-level support. For months, these ML engineers kept swinging a bat at a pinata, blindfolded in a dark room; They were trying to improve some obscure offline metric by throwing all the techniques, algorithms, and compute resources at their disposal. It is dark, their arms are tired, and there is candy everywhere on the floor, but we can't see it. Then suddenly they are asked to go to production. This transition can happen without any clear leading signals. Product managers or business leaders ask the engineers to put their MVP in production. The lucky ones know how to move to the expansion phase, which requires them to switch their mindset. Now they are asked for maintainability, reliability, and scalability. They do their best to put their MVP in production. Still, it does not take long before the product managers come up to them infuriated at why a similar functionality that took 3 hours to develop, one month ago, is estimated to take a minimum of 4 weeks to implement in production. The ML developers revise their estimates, and so begins the inexorable downward spiral of ever decreasing code quality and project velocity. This humble book aims at smoothing the transition from exploration to expansion with a set of techniques and principles that proved useful in the software engineering world.

This book will look at ML code from many different dimensions. We will look at the small scale concerns like functions. We will explain useful big ideas of components and architecture design. We will examine the inside details of ML code and the outside form of ML applications. By the end of this book, we will have covered a lot of concepts and coding techniques. You will start recognizing

good from bad ML code. You will also learn how to write better ML code and transform smelly, rotten, and disgusting code into fresh and useful code that is easy to change.

Do we need a book about machine learning code? Leading ML tooling providers are bombarding us with managed ML services. They assure us that we are moving away from the old days of coding every single ML application detail. They are saying that we are moving to a new world of AutoML, Automatic Machine Learning Pipeline Generation, and magically managed APIs that will predict your next job, spouse, and favorite book (i.e., like this book). Business folks will be able to speak their minds about what to predict, recommend intuitively, or forecast, and the ML applications will magically appear from thin air.

That is very far from the reality we are experiencing. Here is my prediction. There will be machine learning code, lots of it. Yes, it is reasonable to expect that the level of abstraction will keep rising as the ML industry discovers the right abstractions, libraries, and infrastructure needed. However, humans are not yet able to translate their customer's inspirations and intuitions into working applications without formal, rigorous, and incredibly detailed programs that can be interpreted and executed by our machines.

Bad Machine Learning Code

Bad machine learning code is among us. But does it matter? Why should we care about this "badness." Why can't we focus on delivering feature after feature? In a way, this whole idea of clean machine learning code relies on a fragile assumption: clean machine learning code matters.

Disasters in ML code are starting to appear all over the place. Bad ML code destroys businesses that had bright futures, and tarnish the image of well-respected companies. Machine learning pipelines are software pipelines, after all. They are full of needless complexity and repetition. They are also very prone to thick opacity, rigidity, and viscosity of design. With these issues, ML failures are growing in importance at an unprecedented pace. Here is a panoply:

- Self-driving cars that are hitting pedestrians.
- Gender bias of large scale translation system.
- Job search engines that are racially biased.
- Simple facial masks that are hacking face id systems in smartphones.
- Machine learning-based social news feeds radicalizing people and contorts public opinions.
- ML generated insurance company premiums predicted to be zero dollars and set for the year.
- Smart financial systems that were making bad decisions and losing $456M in a day (e.g., Knight Capital) because of dead code paths.
- Medical malpractice lawsuits are happening because of unsafe and incorrect treatment ML recommendations.
- Policing systems are prey to ML disasters where facial recognition systems target people of color with high accuracy because that's all they know.

On a less dramatic note, here is an example scenario. A capable junior machine learning engineer with two years of experience started working this month for a medical insurance company. He just inherited an insurance prediction pipeline that predicts optimal premiums for members. Management is asking him to change the model type: ""We want you to change the model from a logistic regression model to a gradient boosted trees model because 'offline' the expanded model works better, and we can increase revenue." No problem, he says, and gives the three weeks minimal estimate. Management is pretty happy with that because they got the minimum estimate.

What kind of issues will he face to get the new version of this critical pipeline working? He clones the project from a git source control system dodging the first bullet. He says to himself, "Nice! Version control will surely help with this project!". What else will he find:

- **Challenge 1**: *Viscosity of design*. He looks for a "tests" folder and finds some notebooks lying around to test the existing code manually.
- **Challenge 2**: *Viscosity of the environment*. He finds that the "notebook unit tests" take around 1 hour to complete.
- **Challenge 3**: *Rigidity*. He ventures to change one name in the loss function tests and finds out that he has to change that variable name in 6 different places.
- **Challenge 4**: *Fragility*. He gets one test related to gradient calculations to work and breaks the code that * generates the test data.
- **Challenge 5**: *Immobility*. He tries to reuse a small data function from the gradient calculations tests to fix the data generation tests, but he is not sure what will be the impact of reusing that function.
- **Challenge 6**: *Opacity*. What does that "ovh_v5" variable name stand for anyway?

Is this a familiar scenario for you? Do you regularly get slowed down and dragged to the border of quitting your job because of a codebase is a jungle of tangled leaky abstractions glued by layers of stitching. We regularly cope and endure, while we attempt to decode encodings, and hope for signs and indications to understand this mad-libs-based story. From the outside, it looks like a well-written story. From the inside, it is a hybrid gallimaufry-hodgepodge that the product and engineering managers want us to extend.

The question is, then, why do You do it this way? Here are the usual options. Maybe you were in a rush to finish that feature engineering pipeline, and you were afraid that your boss would be mad at you for using the time to clean up the code. Maybe you were exhausted by writing the "dumb" model validator and just wanted to move on to a new exciting task. Perhaps you looked at all the promises you made during the previous sprint meeting and felt overwhelmed and started hard-coding models hyper-parameters with index-based variables to get over with this task. Maybe you said to yourself that this hand-crafted model architecture is temporary, and we will fix it later.

Any ML engineer with a couple of projects under their belt did this at one point or another. We all leave messes for later. We tell ourselves that we will come back to this and make it right. We all experienced the comfort of seeing the turtles tower we build work and deciding that it was good enough. We can come back later, add those tests, refactor the functions, and do a deep clean up

with some power washing tool to remove the rust, dust, debris, and rubble. On the one hand, you might be thinking, "Don't worry if it doesn't work right. If everything did, you'd be out of a job." - Mosher's Law of Software Engineering. But on the other hand, you vaguely remember the famous saying: "Later equals never."- Le Blanc's law.

The TCO of a Predictive Service Mess

When was the last time messy code impacted your productivity? Were you asked to add a single new categorical feature to an existing ML pipeline only to find that changing anything changes everything?[1]. Over time the mess increases; the entanglement grows. The engineers throw more solutions to the problem. This uncoordinated activity creates more chaos. ML teams that were churning out continuously improving models, now find themselves slower than a tortoise in the sand. Any change requires a full understanding of many other parts of the system. Teams become terrified of changing anything because they don't know the impact of their change. They feel exposed that their code has taken a life of its own. They are frightened by the sheer amount of knowledge needed to extend anything in the system. The system resists change because of its rigidity. The code is so fragile that changing anything can break non-related functionality. The previous owners left, and the naming conventions are so opaque that no one can clearly explain what a specific function does. The cloak of immobility sets in as the team becomes more helpless.

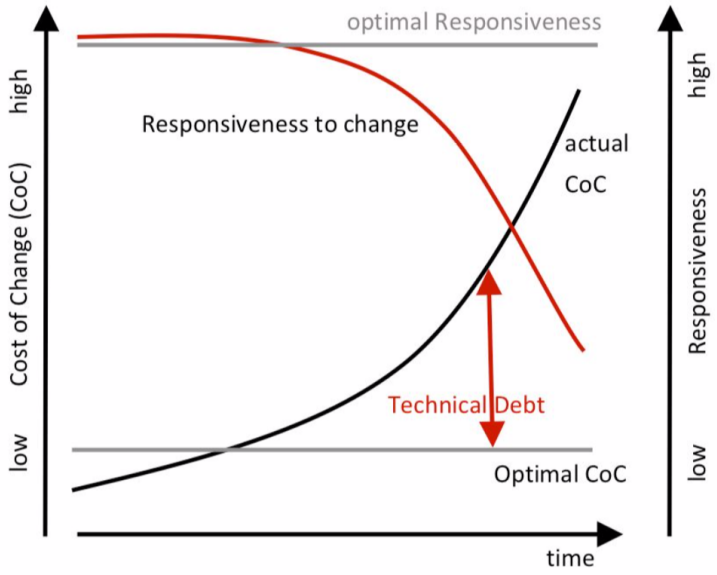


Figure 1. Change in Cost of Change and Responsiveness over time. [6]

Figure 1 shows how the cost of change vs. the responsiveness evolves as time passes. Initially, the productivity/responsiveness of the team is at an all-time high. Management comes up with new functionality, and the ML engineers can explore that path in no time. They come up with a new request: "We need to include the class probability in addition to the existing class prediction to the public API." "No problem, boss, we are on it.". Initially, the cost of a change like this is minimal. "Let

me just change the function result to include a second field in the JSON we return to the client…..". As time passes, similar requests start getting delayed, and management notices that the responsiveness is not what it used to be. So they decide the ML team needs more staff to increase productivity. The new team members get onboarded, and suddenly, the project's cost grows in a step function fashion. Now the new team members are getting familiar with the codebase and learn, "How things are done around here." Besides, due to the increase in head-count, the team is under even more pressure to deliver. More mess builds up in the system.

Rebuild the ML Pipelines from Scratch

The ML application cost reaches a tipping point. The V1 team wants to rebuild the ML pipeline from scratch using the latest technology and methodologies. Unfortunately, management knows how much was spent on the first version and is reluctant to unlock the funds for this new pipeline.

They eventually give in to the demands of the ML engineers. A new cross-functional capability-wide team is brought up. The selection of the team members is a nightmare. Everyone wants to be on the new V2 team, and management selects the best and brightest to staff this newly formed crack team. The rest of the ML engineers are left maintaining the ruins of the existing pipeline. They usually are jealous of the new V2 team. They imagine that working on a greenfield project is all fun and games. But in reality, the two teams are in a race.

The customers still ask for new features, and the v1 team has to ship bug fixes and new functionality. At the same time, the V2 team is looking for documentation on the design of the V1 pipeline, but they don't find any. The only true reference is the system itself. Not only that, but as they learn about the V1 system, the V1 ML engineers keep changing it under their feet.

The rewrite drags on, the V2 team members change over time and morphs into a V3 team. The V3 team looks at the mess created by the V2 team and wants a new redesign because they don't want to maintain the old V2 "new" design.

If you spend any time around software projects, this story is probably familiar to you. For ML engineering, unfortunately, too few reports of such stories exist because of how young the field and the applications are.

The moral of this story is there are many reasons for accepting the cost of keeping ML code clean. The most central reason is professional survival. Not unlike in gardening, few projects can survive without constant attention. ML practitioners learn the hard way that taking the time to care for the living and growing entities that are ML systems is central. Central to both to their mental sanity and the health of their business and its economic goals.

Ideal vs. Real Machine Learning Workflows

With all the hype around machine learning and data science, customers are confused when they are told about the complexity of ML software. They see the following linear figure that defines a nice marketing message of the CRISP-DM:

Figure 2: Idealized ML workflow [2]

Figure 2 shows one such pleasant abstract process with a few well-defined transitions and clearly defined boundaries between ML/DS tasks. The whole image brings a sense of comfort to the reader about the stability and predictability of any ML project. First, there is the business understanding coupled with the data understanding. Then once we have full knowledge of the business data, we move to the data preparation, which is coupled to the modeling part. We repeat the process to get the right data in the right model. Once ready, the best model gets moved to the evaluation part that that will decide if the model is prepared for the business prediction task. This might need us to refine our business understanding and data understanding. Once we are ready to expose the model, we deploy it in a single, nicely defined box of peaceful bliss. The business is happy, the ML engineers rejoice, and everyone gets promoted!

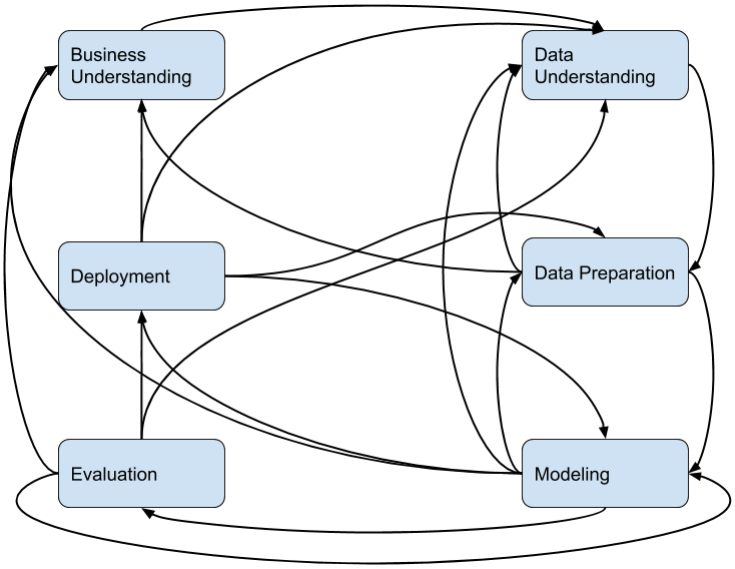


Figure 3: Simplified but Realistic ML tornado workflow

In reality, ML projects, like any software project, are intertwined, domain-specific, engineering-heavy, compromise ridden endeavors. There is no way around that. The marketing and sales process might say otherwise, for good reasons. Still, the reality is that software engineering projects tend to move away from perfect circles and generate tornadoes of dependencies.

This view of ML software can help understand how the code that powers it all will look after a few "iterations." The ML code will reflect the ML software design tornado if left unchecked.

Taking Responsibility for ML Code Rot

Why does working ML code rot? What makes ML code so susceptible to deteriorate and become a nasty mess? We usually look at requirement changes that are not compatible with the current system. We call for the everlasting phrase: "It was not designed to support that functionality." We blame management and leadership for their ignorance of the intricacies of our craft. We get infuriated because of changing schedules and unrealistic timelines. We look at users that change their minds on a whim. All this external blame might be misdirected. The truth is that the fault is most probably coming from us.

I understand that this might get pretty controversial. But we are responsible for guiding the managers, product people, and sales folks. They rely on our judgment to tell them what is possible and what is not realistic. Even when a top-down directive comes crashing from the skies, we should not hold back and be reserved about our opinions on the plans handed to us. The level of responsibility in the hand of an ML engineer is so high that we must speak up. We are just as responsible for ML project failures as any other function on the team.

I hear you say: "Preposterous! I am putting this gross book down!". "If I don't deliver the ML system on time, I'll get kicked out to the curb… I can't stand up to the product manager and tell them that

they are wrong about their estimates … madness!". Most product and engineering managers are reasonable people. They need your honest estimates and opinions. They rely on you to fight back to protect the code with the equal intensity they are fighting to defend their schedules and the promises they made to clients. It is all about checks and balances. They need your unwavering professionalism to help them go fast.

Overfitting to Deadlines

Think about the bias-variance tradeoff. You probably deal with it every now and then when building an ML system. When training ML models, that is a central concept that gets hammered in our heads.

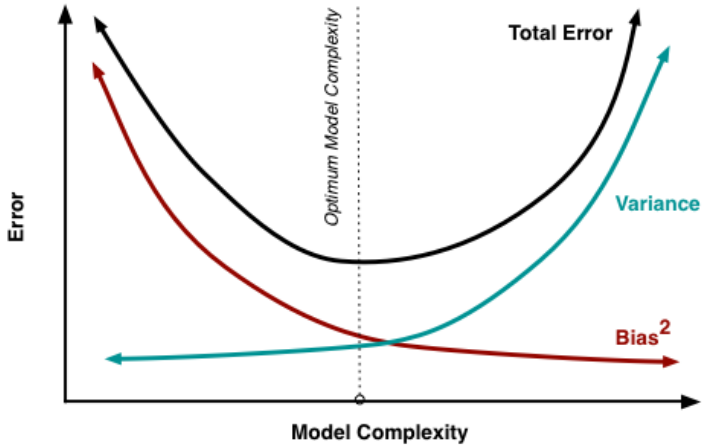


Figure 4 Traditional Bias Variance Tradeoff [3]

High variance leads algorithms to model the random noise of the training data. This usually leads to overfitting. In the software world, this akin to fitting your development style to the deadlines at all costs. Everything takes a back seat compared to the importance of delivering the ML system on time. You get excellent training set accuracy in the short term, but your testing set accuracy tanks dramatically.

High bias, on the other hand, leads algorithms to miss relevant relations between the input features and the predicted output. This usually comes from low complexity models, when we expect, for example, all relationships to be linear. In the software world, this maps to development teams that are stuck with initial assumptions about the software and are not open to change.

Experienced ML practitioners know that systems need to balance their design between underfitting and overfitting to the deadlines. They know that the only way to keep their code manageable is to keep their code clean continuously.

The Art of Feature Engineering Your Code

Feature engineering is still an art form. Coming up with relevant, cheap, and non-correlated features is probably still why ML engineers get paid so well. Experienced ML engineers gather a growing toolbox of feature engineering tricks and techniques that they employ as they see fit. However, when you ask them to explain their reasoning, the conversations usually drifts into, "it worked on a similar problem last year" and "I saw that technique work well in some obscure Kaggle competition write up." They learned the tricks of the trade by trial and error. They built an intuition for what to do when faced with different features.

It is essential to realize that software design is very similar. It is reasonable to ask, "How do I write clean machine learning code?". It is pretty hard to write clean machine learning code when we don't know what it means. Unfortunately, ML engineers write more code than they read. It is similar to asking a screen-writer to improve their character development without having read 10s or 100s of screenplays and novels beforehand.

We are in the same situation; we are asked to write without reading enough. Newcomers to the ML field have a vague sense that some ML code is poorly structured, but they don't have the mental tools to improve the code. They either witnessed full messes or great software. They did not have yet the chance to see live the multitude of transformations, variation, and options that a more experienced ML software writer can perform.

What Is Clean Machine Learning Code?

It should come as no surprise that this book is heavily inspired by the fantastic body of work that Rober Martin provided the software design world. In his book, Clean Code, he recognized that clean code means different things to different people. There are schools of thought, style choices, and philosophies that are very personal to each software professional. How would that translate to machine learning code? Let's see how we can adapt what giants of software design said about software, and attempt to merge it with the goals of machine learning:

- **Robert C. Martin**, the author of the book Clean Code, said, "Remember that code is really the language in which we ultimately express the requirements. We may create languages that are closer to the requirements. We may create tools that help us parse and assemble those requirements into formal structures. But we will never eliminate necessary precision—so there will always be code."
- **Martin Fowler**, the author of the book Refactoring: Improving the Design of Existing Code, famously stated, "Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
- **Michael Feathers**, the author of Working Effectively with Legacy Code, said, "Clean code always looks like it was written by someone who cares."

- **C.A.R. Hoare**, the inventor of quicksort and too many things to mention, said, "The most important property of a program is whether it accomplishes the intention of its user."
- **Donald E. Knuth**, the author of The Art of Computer Programming, said, "The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly."
- **Guido van Rossum**, the author of the Python language, said, "Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressiveness is endangered."

On the other hand, we have giants of machine learning that express their views on ML design needs:

- **Andrew Ng**, the founder of Coursera and Deeplearning.ai, said, "When you become sufficiently expert in the state of the art, you stop picking ideas at random. You are thoughtful in how to select ideas and how to combine ideas. You are thoughtful about when you should be generating many ideas versus pruning down ideas."
- **Jeremy P. Howard**, Founding Researcher at fast.ai "If you get a detail wrong, much of the time it's not going to give you an exception. It will just silently be slightly less good than it otherwise would have been…You just don't know if your company's model is like half as good as it could be because you made a little mistake."
- **Peter Norvig**, Director of research at Google and author of Artificial Intelligence: A Modern Approach, said: "More data beats clever algorithms, but better data beats more data."
- **Andrej Karpathy**, Director of AI at Tesla, said, "Software 2.0 can be written in a much more abstract, human unfriendly language, such as the weights of a neural network. […] our approach is to specify some goal on the behavior of a desirable program (e.g., "satisfy a dataset of input output pairs of examples", or "win a game of Go"), write a rough skeleton of the code (e.g. a neural net architecture), that identifies a subset of program space to search, and use the computational resources at our disposal to search this space for a program that works."
- **Soumith Chintala**, creator and development lead of PyTorch at Facebook A.I, said "Put researchers first: PyTorch strives to make writing models, data loaders, and optimizers as easy and productive as possible. The complexity inherent to machine learning should be handled internally by the PyTorch library and hidden behind intuitive APIs free of side-effects and unexpected performance cliffs."
- **Martin Zinkevich**, the author of the Rules of Machine Learning and lead on TFX, said, "Plan to launch and iterate: Don't expect that the model you are working on now will be the last one that you will launch, or even that you will ever stop launching models. Thus consider whether the complexity you are adding with this launch will slow down future launches."

One striking element of the above quotes is how intertwined and varied the way knowledgeable people report on their software views. There is a sense of caring for the future of the code by expecting "change" to be a central part of the software life-cycle. There is also a clear sense of excitement in the ML community that can lead to exuberant exploration first mode of operation. There is also a general sense of complexity getting out of hand. These experienced folks lived through

many iterations of software projects and realized that complexity is the enemy. They tell us to use more data instead of sophisticated software. They tell us about the impact of complexity on future launches. They talk about removing side-effects and hiding functionality behind simple APIs. They talk about the difficulties of debugging ML code due to silent errors that stem from complexity. They mention that as the ML developer matures in their craft, they have the mental tools to grow and shrink the code's complexity as needed.

In many ways, this book will be forever imperfect. It is a collection of design techniques that were crowdsourced over the years since the 1970s and beyond. The people that came up with the principles, rules, and laws that we will discuss, have been through the pain of building significant software and were kind enough to express their wisdom so that we don't have to go through the same traps.

One word of caution is necessary here. None of the elements in this book are absolutely right. There is no scientific proof that the design techniques in this book are always valid. They are a representation of a school of thought that has good and bad points as all schools of thought have. We hope that it will help you develop a sense of professionalism and craftsmanship by benefiting from our experience. There is ample room for you to disagree with the contents of this book, as you probably will. Some of the materials are radically opposed to the incentives you are trained to seek. We hope you will give this content a look and grab the pieces you need to improve your software as an ML engineer.

Inference vs Training of Source Code

It is estimated that 90% of all the cost of machine learning is at inference time[4]. Generating predictions at inference time far outweighs the cost of producing the trained model. All the optimizations that we perform to improve ML models' training impact at most 10% of the total cost. Since most new projects are geared toward MVP style development, they focus on optimizing the training part and commonly ignore the effort that will be needed in the inference phase. Some less than noble thoughts creep in such as "that will be engineering and operations' problem." and "once I am done with the initial model, I am moving the next project, no need to optimize the inference phase."

I am using this metaphor to relate to reading versus writing software. If you agree with the above inference vs. training cost, you will find that a very similar thing happens on code. We read code a lot more than we write it. You are effectively writing so that other humans can read the code and extend it. We fall into the trap of optimizing the writing of code with no concern for reading it.

Recording screencasts is an excellent way to figure out what happens in that neat Jupyter notebook you are working on. A "writing" session might look something like:

- Open the notebook.
- Scroll down to the last cell where you need to add another aggregation.
- Pause to check if the current dataframe has the correct columns.

- Scroll to the top to where the dataframe was joined with the new index.
- Scroll back down to the last cell.
- Ah! Restore the notebooks cell that was deleted by mistake.
- Get the head of the dataframe.
- Start typing the group by operation.
- Go look up the syntax on the pandas library docs.
- Scroll back up to check that the join was a left join.
- Pause and think.
- Remove the group by that added in the cell.
- Replace it with a simple count that will do the trick this time.

….

Robert Martin, in Clean Code, states it elegantly: “making it easy to read actually makes it easier to write.” You must read the code around the code you are extending. There is not much leeway around that. We must strive to make the code easier to read.

Active Reinforcement Learning for Source Code

Active reinforcement learning is useful when dealing with Sequential Decision Problems. In this setting, the agent is asked to learn and act. It is looking to find an optimal policy to maximize the overall cumulative reward. The core idea is trial and error, but the agent is tuned so that it optimizes the overall “long-term” cumulative reward. Agents are also usually provided with exploration vs. exploitation modes of operation. The exploitation mode helps gather known rewards, and exploration helps the agent discover new ways to increase the total rewards.

These modes are very similar to what we deal with when writing software. We are tasked with writing software to solve a problem faced by the business “right now.” But we are implicitly asked to keep the code base “clean over time.” The same explore exploit strategy can be used for source code. There are times where exploitation and releasing a new functionality is paramount. But there must be time spent exploring and improving the overall code that generates that functionality. Otherwise, the code gets stuck in a local minimum. And we don’t like local minima that much. So the goal is that when you are tasked to change that feature engineering code, pick something small. Fix some naming, break that one hot encoding function into multiple functions, remove a little duplication, and fix that obscure if statement that picks the model to train. Commit back the code just a bit better than you checked it out.

Transfer Learning and the Origins of CMLC

It might change later, but these days, researchers are almost forced to use transfer learning to obtain state-of-the-art results in machine learning. Take NLP tasks; it would be immensely costly to retrain

a GPT, BERT, ELMO, or Transformer from scratch. However, if you need to match the performance of published work, then you have no choice. You have to be downstream of the giants of knowledge compression. You take their model and fine-tune it to fit your use-case. As we get larger and more accurate upstream models, this trend is accelerating.

Similarly, this book is attempting to stand on the shoulders of giants. There is no doubt about it. The awe-inspiring body of work related to software design is breath-taking. Comparatively, software design works that focus on the machine learning field are few and far between. The current 2019 landscape of books and writings focus more on training the current cohort of young, energetic, and immensely ambitious budding ML engineers on the behavior of ML code. The existing training focuses on the behavior of ML code. They focus on how to get started and how to build MVPs. They focus on how to mash together with a set of existing ML tools, and how to duck tape the whole thing with just enough gorilla glue to explore the market for ML products. This is hugely necessary since many of the ideas and inventions will be trashed anyway. This book cares about the next step in the life-cycle of ML software. That moment when there is a hint of a customer that needs your product, and bug fixes and new feature requests start coming in. Recognizing that moment is crucial for the lifetime of the product. In that critical inflection point, the ML engineer should have the mental tools to start training product owners that the effort and time required for exploration and expansion are different. Most importantly, the ML engineer needs to have a plan on how to transition smoothly between the crack-fueled exploration phase, where nothing else matters but product-market fit, and the expansion phase, where stability and maintainability start becoming the core challenge.

This book follows the central philosophy: "To go fast, you must go well" [5]. This fundamental tenet of clean code is the guiding principle of CMLC as well. The challenge is that while CC has guided many cohorts of programmers and still does to this day, the newly minted ML engineers are not aware of what is coming for them. This book aims to act as a reference to train these bright minds on the best practices that the software industry has to offer with examples that machine learning-centric. The concerns we address in this book are geared specifically to the ML crowd, and all the examples target the various dimensions of ML products.

Conclusion

Programmers are akin to studio artists. ML engineers are similar to that as well. They come every day to work and get tasked with creative tasks. Most of them try to balance their artistic needs of organization and elegance, with the business needs of quarterly earnings.

This book is about this art form. But art books usually are only able to share the thought process of fellow artists. This book aims to do the same with a set of programming tricks, principles, techniques, heuristics, disciplines, and methodologies.

Beyond that, it is up to you to take them and include them in your practice. Geoff Colvin put is best in his book, Talent is Overrated: What Really Separates World-Class Performers from Everybody Else, "Deliberate practice requires that one identify certain sharply defined elements of performance that need to be improved, and then work intently on them."

References

[1] Machine learning: The high interest credit card of technical debt. D Sculley, G Holt, D Golovin, E Davydov, T Phillips - 2014 - research.google
[2] https://en.wikipedia.org/wiki/Cross-industry_standard_process_for_data_mining
[3] http://scott.fortmann-roe.com/docs/BiasVariance.html
[4] Amazon Elastic Inference: Reduce Learning Inference Cost https://www.youtube.com/watch?v=hqYjkT0BP1o
[5] Clean Code: A Handbook of Agile Software Craftsmanship. Robert C. Martin. 2008. Prentice Hall PTR, Upper Saddle River, NJ, United States
[6] Clean Code Cheat Sheet: https://www.bbv.ch/images/bbv/pdf/downloads/V2_Clean_Code_-V3.pdf

Chapter 2 - Optimizing Names

Introduction

We find ourselves naming everything, but naming is hard. We need good rules and guidelines for making good names. Deployments, components, classes, ML models, features, transformations, functions, variables, arguments, SQL queries, DB objects, and more, they all need names. Here are some rules for making the best of a hard situation.
Most of the rules in this chapter are gathered from around the thinkers and authors [1] that tried to construct rules for software naming. We compile them here to provide a unified list of concerns and solutions with a focus on ML components. However, as said before, ML code is software after all, so this might help you with all the neighboring concerns to the core ML code that you might face when integrating with larger/external systems.

The Objective Function of Names

The primary objective of the good names is to reveal an "intention." Names are the primary source of information for your readers. They are compressed blocks of meaning, and as the author, you are responsible for this compression. Using the right amount of compression is vital. Adapting the compression as the intent of a variable as it evolves, shows care and compassion for your readers and most importantly, your future self. Will you understand the intention of your names in 3 months when someone asks you to change the code you wrote last quarter?
The names you create should communicate three key concepts:

- Why does it exist?
- What does it do?
- How is it used?

If you need a comment to explain what a name represents, then act now and rename it.

Here the CSV file we are reading ends up in a variable df. This comes from most tutorials online that take a shortcut to speed the onboarding of tutorial takers. We should choose names that reflect the contents of this dataframe.

Bad:

```
df = pd.read_csv('data.csv') # read house prices csv
```

Good:

```
house_prices = pd.read_csv('data.csv')
```

Revealing intent is good for your sanity. You come back one week after writing a function like the following and forget what the function is doing.

The code below is hard to decrypt. Some operations require prior external knowledge about the columns. The function is small, and the formatting is ok. There isn't anything particularly wrong with how it is structured. Why is it hard to decipher? The issue here is that we are assuming that the readers of this function are aware of the external context.

1. What sort of data is in the df dataframe?
2. What is the meaning of the 6th column?
3. Why is the last column used?
4. Why are we overwriting the dataframe df?

Bad:

```
def get_data(path):
    df = pd.read_csv(path)
    df = df.groupby(df.columns[6]).sum()[df.columns[-1]]
    return df
res = get_data('house_prices.csv')
```

Good:

```
def get_house_prices_by_age(filepath=None):
    house_prices_dataframe = pd.read_csv(filepath)
    house_prices_sum_by_age = house_prices_dataframe.groupby('AGE').sum()
    house_median_value_sum_by_age = house_prices_sum_by_age['MEDV']
    return house_median_value_sum_by_age
house_median_value_sum_by_age = get_house_prices_by_age(filepath='house_prices.csv')
house_median_value_sum_by_age.shape
```

The missing information could have been included in the function above using good naming. In this case, we are working with house prices data. We are interested in the sum of the median price of houses per age. We can replace the function call with a name that wouldn't need a comment to explain what the function returns. Then we could rename the dataframe name to reflect what it is and what it represents. Then we can replace the brittle positional grouping and projection keys with meaningful names. Furthermore, we can replace the return value with the actual intention of the function.

Avoid Mislabeled Labels

That target column in your dataset is precious. Be it a numerical value you are regressing on, a class to predict, or something else that is central to your ML task. What if someone came in and reset the values of that column from 0's to 1's without telling you, effectively flipping the classes? Of course, you would be able to debug the issue and fix your ML pipeline. But just why? And why do we do it to actual code?

We sometimes leave false hints that mangle the meaning of your code. For example, `tf`, `pd`, and `np` would be bad variable names because they are the usual abbreviations of common python libraries (e.g. Tensorflow, Pandas, Numpy). Even if we are honestly trying to encode a `processed_data` with `pd`, it would be misinformative to the reader.

Be careful with long variable names as well, `evaluated_data_with_map_global_metrics` would be easily confused with `evaluated_data_with_mrr_global_metrics`. Similar variable names have the disadvantage of confusing the reader and our IDEs as well. Because IDEs have magical code completion capabilities, differences between variables should be as close to the beginning of the name as possible. It would be easier for the IDE to autocomplete `map_evaluated_data_metrics` vs. `mrr_evaluated_data_metrics` because the second keystroke would already disqualify one of the two names.

Also, do yourself a favor and choose code editor fonts that are programmer centric. Nobody wants to spend time distinguishing between the following:
`ml_pipeline` and `m1_pipeline`
`feature_01` and `feature_Ol`

Avoid Noisy Labels

Data pipelines quickly get complicated, and the concepts' names start colliding in our code. For example, this surfaces when we have two names, one that comes from a library and one that we are adding to our application business logic. Programmers are inclined to change a single letter to keep their chosen name while not clashing with the existing names; DataFrame, a pandas class name vs. Dataframe, an application class name.
Even internally within the application, the same names keep reappearing. How many `df` have you seen in pandas centric code? The quick fix that starts showing up is adding numerical suffixes, `df1`, `df2`, `df3`, `_df`, `_df1`, and so on. These names don't say what they are meant to represent. Check out the following:

Bad:

```
def lower_case_column_names(columns1):
    columns2 = [str.lower(column) for column in columns1]
    return columns2
lower_case_column_names(["AA","BB","CC"])
```

Good:

```
def lower_case_column_names(input_columns):
    output_columns = [str.lower(column) for column in input_columns]
    return output_columns
lower_case_column_names(["AA","BB","CC"])
```

Adding noise to your names does not help anyone:
No `table` in your DB tables or in your variables that refer to tables.
`customer_table` vs. `customers`
No `dataframe` in your dataframes variables.
`invoices_dataframes` vs. `invoices`
No `a`, `the`, `an` words in your names
`aModel` vs. `Model`
No `object` in variables
`feature_object` vs. `feature`

Adding the datatype to the variable in python can be attractive to keep track of the type of variables. However, with the advent of a mature optional typing system, you can now use that convention in your code. Not only you'll be able to determine the type of an object, but the IDEs will be able to give you full autocomplete for methods and attributes available:

OK:

```
def transform_features(raw_features_dataframe)
```

Better:

```
def transform_features(raw_features: DataFrame)
```

Function names would benefit as well. Adding noise words to avoid name clashing can be infuriating for your fellow developers. In the example below, how are we supposed to know the difference between info, details, and information?

```
def get_model_info():
    pass
def get_model_details():
    pass
def get_model_information():
    pass
```

Make Siri Say it

Did you ever get a text message and you asked Siri or Google Assistant to read you the message. What happened when you got a message from your friend that says “kk, brb” or “lol, just fyi tmrw”. You get the point; a good baseline is to put your names in a “how to pronounce it” app and see what comes up. Usually, the default behavior for these machines is to spell the word when it is not a known word. A similar process will happen in the mind of your fellow programmers. It is easier to read and remember pronounceable names, with the added benefit that people can have intelligent conversations about them. Check out this example:

Bad:

```
class FTran6:
    def __init__(self, i_fs, fs_s, t_strt):
        self.i_fs = i_fs
        self.fs_s = fs_s
        self.t_strt = t_strt
```

Good:

```
class FeatureTransformer:
    def __init__(self,
                 input_features,
                 features_schema,
                 transformer_strategy):
        self.input_features = input_features
        self.features_schema = features_schema
        self.transformer_strategy = transformer_strategy
```

How do we call that `i_fs`, `fs_s`, and `t_strt`? It used to be called “The Great Vowel Shortage.” Fortunately, we found new sources of vowels on the asteroid belt, and all is back to normal. This can be easily resolved by using the full names of the object we are manipulating. And the next time you are discussing a new `transformer_strategy`, you’ll thank yourself for choosing a more pronounceable name.

Make it Greppable

Magic numbers are handy. If you are prototyping something and add a couple of number literals here and there to get the code to work, that's fair. But don't leave them there once they work.

Imagine how hard it will be in the code below to change the embedding max size if we need to track the number 30 around the code? In SQL, once a query changes, how will we know where to search for those positional fields? And what happens when we realize that we need to support leap years in our code, and the numbers 365/366 are sprinkled around our codebase? These numbers will most likely be included in other parts of the code, either on their own or as part of other longer or shorter number literals.

Bad:

```
#Magic Numbers
generate_embeddings(input_features,30)
-----
for i in range(52):
    ....
...
query = "SELECT .... GROUPY 1, 5 ORDERBY 7"
```

Good:

```
embedding_max_size = 30
generate_embeddings(input_features, embedding_max_size)
-----
num_days_in_year = 365
for i in range(num_weeks_in_year):
    ....
-----
query = "SELECT .... GROUPY age, income ORDER BY prediction_accuracy"
```

Avoid Name Embeddings

An **embedding** is a relatively low-dimensional space into which you can translate high-dimensional vectors. Embeddings make it easier to do machine learning on large inputs like sparse vectors representing words. They are a wonderful tool to deal with all sorts of data in our ML pipelines. But in our code naming conventions? Not so much.

Encoding extra knowledge in your names might sound like a good idea, but it should be avoided when possible. Adding the datatype to a variable should be replaced with typing information, `features: DataFrame` vs. `features_df`. Abstract classes and interfaces should avoid the

`IModelTrainer` pattern for upstream concepts and avoid the `ModelTrainerImpl` for the downstream concepts. Finding good names is hard, but worth it.

Avoid Semantic Name Maps

Finding your way through new code with a dictionary that has a semantic mapping of variables to what they really mean is exhausting. Reduce friction between reading and understanding the responsibility of each name. Check out the following:

Bad:

```
f_list = ('city', 'county', 'loan_amount')
....
#20 lines of code later
....
for d in f_list: # What is f_list here?
    k = transform(f)
    store(k)
```

Good:

```
feature_names = ('city', 'county', 'loan_amount')
....
#20 lines of code
....
for feature in feature_names:
    transformed_feature_name = transform(feature)
    store(transformed_feature_name)
```

Remember that we read code a lot more than we write. So be nice to your readers and make your names self-explanatory so that they don't have to carry a dictionary with them to decode your source code.

Part-of-Speech Tagging

POS tagging is becoming so central to NLP application that we forget the origins of the word. POS is Part Of Speech. While we can use many kinds of POS algorithms out of the box with the excellent NLTK and Spacy libraries (circa 2020), one forgets how to use POS in our code. The rules are simpler than expected:

- **Classes** use nouns to represent objects: `DataLoader`, `DataValidator`, `FeatureTransformer`, `ModelBuilder`, `ModelValidator`, `ModelPusher`.

- **Enums** are usually adjectives: `CATEGORICAL`, `ORDINAL`, `NUMERICAL`, `RED`, `SUCCESS`.
- **Functions** use verbs to represent actions: `get_features`, `train_model`, `fit`, `predict`, `publish_metrics`, `save_to_db`.
  - One exception here is when a function returns a boolean. This type of methods and functions should be in a yes-no question form. Use the word `is` to represent a question: `is_trainable`, `is_validated`, `is_completed`, `evaluation_is_ready`, `metrics_are_published`.
- **SQL tables* are plural nouns to represent the entities that are stored in a single row: `accounts`, `currencies`, `users`, `loans`, `adjusted_quarterly_returns`, `trained_models`, `artifacts`.

CumSum vs. CummulativeSum

I will not say much here. Keep it clean and professional. Not only will you keep the respect of your coworkers, but HR compliance will not track you down for bad conduct.
Also, remember that GIT remembers forever.

Naming Consistency

Naming consistency helps unify identical concepts under a single name. Take the following example:

Bad:

```
model.train()
model.optimize()
model.fit()

data.fetch()
data.retrieve()
data.get()
```

Good:

```
model.fit()
data.get()
```

Pick a single word per concept, and use it everywhere it fits in.

Avoid Paronomasia

par·o·no·ma·sia /ˌperənōˈmāZH(ē)ə/ noun: paronomasia; plural noun: paronomasias. A play on words; a pun.

As authors, we need to optimize for clarity. We want our code to be skimmable. When puns are included in code, the reader suffers. This is the opposite of what we aim for. There should not be any inside jokes in the code, no matter how attractive it would be. “[Code] is like a joke. if you have to explain it, it is not that good.”[2]

Use Technical Names

Fellow ML engineers are the ones who will read your code. There is no need to use Domain names for everything in your code. It is preferable to use machine learning and statistics based names instead of mapping directly to a customer concern. Just because the business calls something “Damage Rate” does not mean that you shouldn’t use the name “NormalizedErrorRate.” Technical names are fine for most backend things and, when selected correctly, usually map to a mathematical or statistical concept that can increase the consistency of your codebase.

Use Domain Names

As you get closer to the customer, start transitioning to more domain names that will be better understood by the customer. This guideline is especially useful for customer-facing functionality and errors in case the customer gets unexpected stack-traces with relevant names. Support and bug fixes will benefit from the fact that the customer will translate their error to a middle-ground language that will require fewer round-trips of translation. Say the customer is complaining about seeing error related to `CustomerChurnProbability`. If they hit a problem with your software, then that name makes sense to them and will direct your ML engineers to the problem immediately.

Use Clustering for Context

Using the right number of clusters in any unsupervised algorithm is more art than science. This lack of supervision is the same when writing code. The author needs to decide how to add context to variable names. When faced with words that need extra context such as channel, IO, artifact, component, name, status, and the like, the author needs to inform the reader about the context in which these words are useful for the application. This is done by grouping the relevant words in modules, classes, and functions. Prefixes and suffixes should only be used as a last resort.

In the example below, we see how duplicate context adds noise to the attributes:

Bad:

```python
class DataGenerator:
    data_generator_name: str
    data_generator_source: str
    data_generator_format: str

data_generator = DataGenerator()
data_generator.data_generator_name
data_generator.data_generator_source
data_generator.data_generator_format
```

Good:

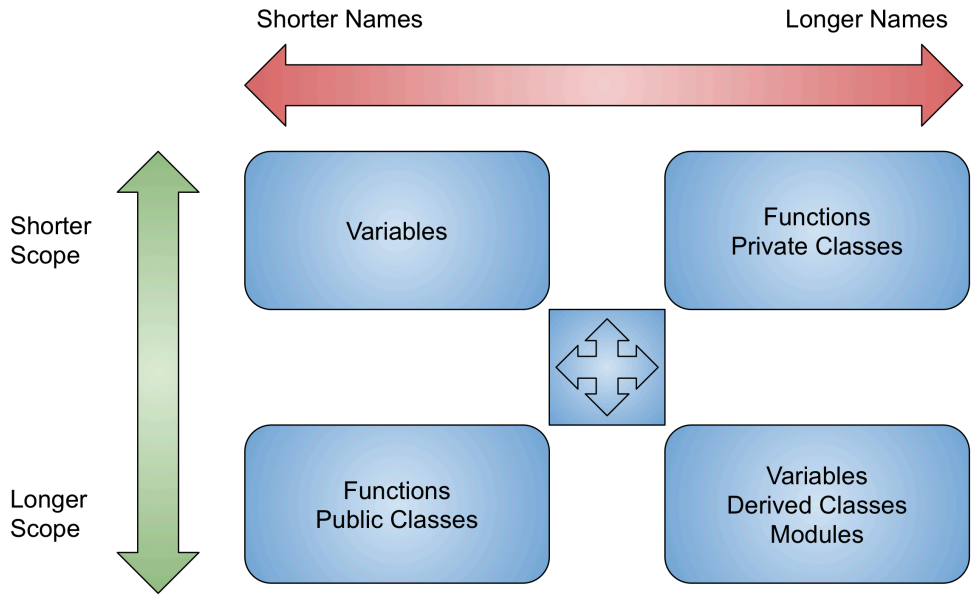
```python
class DataGenerator:
    name: str
    source: str
    format: str

data_gen = DataGenerator()
data_gen.name
data_gen.source
data_gen.format
```

The Scope Length Guidelines

There is a neat relationship between the scope of a name and the length of a name [3]. The same thing applies to your ML code. Scope length is the distance in lines of code, source files, module depth, or how many other places of your code need to know or use a name defined elsewhere.

The guidelines are as follows:



1. Variables:
   a. The longer the scope, the longer the name.
   b. The shorter the scope, the shorter the name.
2. Functions:
   a. The longer the scope, the shorter the name.
   b. The shorter the scope, the longer the name.
3. Classes:
   a. Public classes with long scopes, get shorter names.
   b. Private classes, usually have short scopes, get longer names.
   c. Derived classes, usually have long scopes, but get longer names because of the composed names
4. Modules:
   a. Modules usually have long scopes, get long file names, but short aliases.

Variables

Scope length guidelines 1.a and 1.a have the nice property of allowing for short names for variables that are in short scopes. The best example is an iterator index in a for loop. The variables, i, j, k, are usually fine for short functions, with loops and indexes. See the following example of standard non-optimized matrix multiplication:

```
def mat_mul(X, Y):
    # iterate through rows of X
    for i in range(len(X)):
    # iterate through columns of Y
    for j in range(len(Y[0])):
        # iterate through rows of Y
        for k in range(len(Y)):
            result[i][j] += X[i][k] * Y[k][j]
    return result
```

In the above example, the matrix multiplication is not difficult to follow because we don't have to keep track of the scope of the i, j, k indices. Besides, the matrices X and Y, note the upper case for matrices, also don't have a long scope, so short names are fine. Compare this with a longer function that has the mat_mul function embedded in its core:

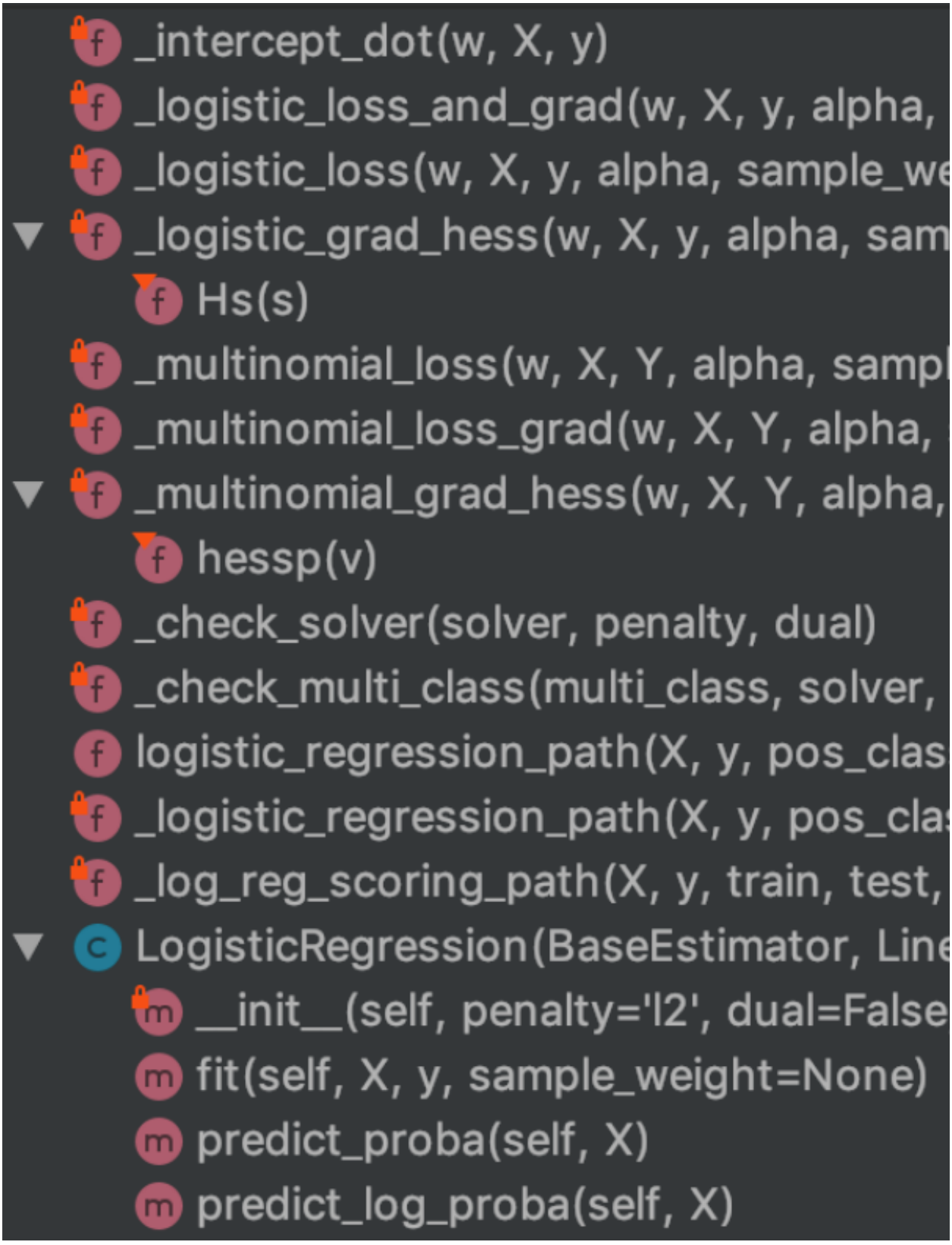
```
def mat_mul_with_many_other_responsibilities():
    # iterate through rows of X
    for i in range(len(X)):
        # iterate through columns of Y
        for j in range(len(Y[0])):
            # iterate through rows of Y
            for k in range(len(Y)):
                result[i][j] += X[i][k] * Y[k][j]
    # 10 lines for filtering_non_zero_elements
    #.
    #.
    #.
    #. complicated_math_calculation
    #.
    #.
    #. update the X and Y variables in place
    #.
    #.
    #. update the result variable
    # 20 lines for clipping_the_elements_with_an_upper_bound_calculated_per_row
    #.
    #.
    #.
    #.
    #.
    #.
    #. update the X and Y variables in place
    #.
```

```
29      #.
30      #.
31      #. complicated_statistics_calculation
32      #.
33      #.
34      #.
35      #.
36      #.
37      #.
38      #.
39      #.
40      #.update the result variable
41      return result
```

We will talk about the oversized length of this function. But imagine having to edit this function. How do you feel about that X, Y, and the results variables? Indeed their scope was significantly expanded. Now we have to jump up and down the file to get to the most recent meaning of those variables. For variables, the guideline is: the longer the scope, the longer the name.

Functions

Functions have opposite guidelines. When we have a function that gets called many times, and in multiple parts of the program, we want a short name for it. On the other hand, if a function has a short usage scope, then we want a longer self-documenting name. In the following listing we get a consistent application of this guideline:

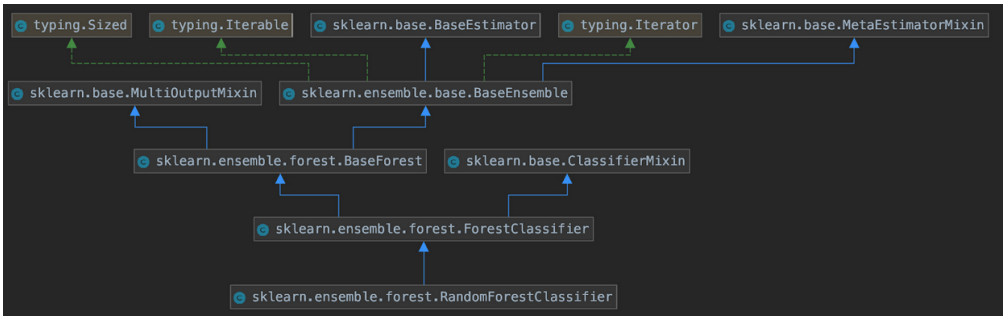

_intercept_dot(w, X, y)
_logistic_loss_and_grad(w, X, y, alpha,
_logistic_loss(w, X, y, alpha, sample_we
_logistic_grad_hess(w, X, y, alpha, sam
Hs(s)
_multinomial_loss(w, X, Y, alpha, sampl
_multinomial_loss_grad(w, X, Y, alpha,
_multinomial_grad_hess(w, X, Y, alpha,
hessp(v)
_check_solver(solver, penalty, dual)
_check_multi_class(multi_class, solver,
logistic_regression_path(X, y, pos_clas
_logistic_regression_path(X, y, pos_cla
_log_reg_scoring_path(X, y, train, test,
LogisticRegression(BaseEstimator, Line
__init__(self, penalty='l2', dual=False
fit(self, X, y, sample_weight=None)
predict_proba(self, X)
predict_log_proba(self, X)

Functions that are publicly available such as fit, predict_proba, predict_log_proba, will be used all over the place wherever we need to train a logistic classifier or produce some predictions. Long scopes need short function names. On the other hand, _log_reg_scoring_path and _multinomial_grad_hess, are probably only going to be used inside this class. It will have a relatively short scope, which warrants the longer self-explanatory names.

Classes

Public classes that will be used all over the place should have short names. A nice compact name allows for denoising the code. For private classes that only live in a short context, they will be better served with a longer, more expressive name. One last exception is derived classes, they usually are public classes with large scopes, but they typically have adjectives that specialize an abstract class or an interface. In the case of derived classes, shorten the name, the base classes themselves should have short names. That way, the derived classes can stay within reasonable bounds. Besides, using a more concise adjective to minimize the size of a derived class name is the right thing to do.

An excellent example of this can be found in the RandomForest implementation in scikit-learn. The figure below shows that pattern. The public derived-class `RandomForestClassifier` has to be long because of its derived nature. It inherits a set of behavior from `ForestClassifier` that is a shorter name. Then moving up the inheritance hierarchy, we see that parent class names `BaseEnsemble` and `BaseEstimator` follow the guidelines of short class names for classes with a large scope, since every ensemble and every estimator will inherit from these base classes in scikit-learn:



Modules

In Python, modules constitute namespaces that group of classes and functions that have long scopes. Their naming is less dynamic than the other types and usually only gets involved during the import phase. The conventional method is to give module/filenames, longer names. This is balanced with shorter aliases when the usage of the model is sprinkled over a broader scope. Here are some examples of this guideline:

```python
# Famous packages do not need much introductions
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

# Same thing for rising stars
import tensorflow as tf
import tensorflow_data_validation as tfdv
stats = tfdv.generate_statistics_from_tfrecord(data_location=path)

# Another rising star module naming strategy
import torch
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

###################################
## Application-level imports naming

# Valid direct module to function access
import quarterly_budget_calculator as qbc
def main():
    qbc.get_quarterly_ebitda()

# Valid full long name imports
from numerical_feature_extraction import NumericalFeatureHasher
def transform_numerical_feature(input_feature):
    nfh = NumericalFeatureHasher()
    result = nfh.transform_feature(input_feature)
    return result
```

Conclusion

Sometimes naming feels like a philosophical introspection. Is a Dataloader really a DataLoader, is a model really a model, is a feature_transform really a feature_transform? The benefit/curse of widespread ML education is that we get a slew of ML tutorials, Kaggle solutions, and data processing libraries examples that do not match the maintainability needs of our industrial ML code. This creates a shared cultural background that encourages quick fixes and short-sighted gains. I hope that this chapter, and this book, will contribute to shifting the balance in the other direction. Good

naming is priceless.

Traditional software developers have learned over the years to use their IDEs renaming capabilities to their fullest. This is yet to trickle down to Jupyter notebooks, where safe refactoring is a hit and miss experience. This creates fear in the ML engineers that need to refactor and rename variables in Jupyter notebooks. For that, here is my painful advice: Once a notebook grows too large (two full mouse pad swipes), extract its contents to a python module, and use an IDE to perform safe renaming. This extraction will cost you time and effort, that is true, and since you probably don't have notebook level tests, bugs will be introduced. But notebooks were not meant for building large complex applications. They have an essential role in the exploration phase, but as the code expands, new tools need to be embraced. Naming will definitely benefit immediately because the IDE will provide you with safe renaming tools and will be a constant positive nagging presence that will push your code in the right direction: Readability.

References

[1] Clean Code: A Handbook of Agile Software Craftsmanship. Robert C. Martin. 2008. Prentice Hall PTR, Upper Saddle River, NJ, United States
[2] Anonymous meme
[3] https://cleancoders.com/. Write Code that Sings. Accessed 2020-06.

Chapter 3 - Optimizing Functions

Machine Learning (ML), Data Science (DS), and Data Analytics (DA) software projects spend their infancy in Jupyter notebooks, Python REPLs, SQL query prompts, and your favorite interactive tool of the day. Usually, ML engineers and data scientists, not unlike software engineers, are given vague feelings, emotions, and thoughts from customers, clients, and managers and are asked to translate these thoughts into code. For ML and DS workloads, they start by interrogating the data to refine the requirements. A query here, a plot there, a data munging operations, a feature transform, and so on.

This inevitably starts accumulating in the history of your REPL, and the number of cells of your notebook starts growing and growing. Scrolling to the right place when the current dataframe was last updated becomes detective work. Some thoughts start bubbling up and taking up longer and longer to explain: "Was it before or after the groupby happened?" "Did I already join this with the metadata that I loaded from the CSV finance sent me by email?" "Is this the first pass of the feature hashing, or did it already happen?"

Then something magical happens, we start combining blocks of code in the first code container we know: Functions. To deal with the complexity, we merge a whole bunch of notebook cells, we indent everything right, we add a function header say, "run_pipeline," and there you go. The notebook was saved from rotting, in one single act of design. We run the notebook cell to define the function, and magically now we have a function we can reuse everywhere in the notebook without scrolling up and down and without having to hold all the details of the function in our head. Time for lunch.

Small is Beautiful

A classic concept from the Clean Code book is: "The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that."
The same concept is reiterated in many different ways over the past 50 years. Sandi Metz, in her talks [1], refers to this principle as: "Make smaller things." I was personally guilty of being the author of functions with a single main function that spanned 300-500 lines of code, that worked, but how hard it was to change! I also, under the pressures of deadlines, cranked out a single ML pipeline function that tried to do everything, with a maintenance price tag of 1000 lines of code. The thing is that, with today's tooling and technology, it has never been easier to do that. We give those fancy retina displays, and multiple screens with incredible, for now, 8K resolutions to developers, and they are magically able to fit whole novels in a single function. They can see the entire application in one big map, which relaxes their fears of getting lost. No indirections, no function calling other functions, no abstractions, no configuration to manage, just one single long procedure. This river of thought takes the reader from the start of the application process, all the way to the end of the application.

The problem is really other people, we live with them, we work with them, and the organization method that works for one developer, might not work for another. Just because a co-worker asked about your weekend does not mean that they are interested in knowing what happened the week before, and what are your plans for the week ahead.

One of the core principles of clean machine learning code is to avoid noise. If a developer needs to modify a specific feature engineering functionality to support some new data that came in last week and broke the primary feature transformation job, then they should not have to understand everything about the rest of the pipeline. If they have to change a single well-contained function, then the bug fix will be implemented faster.

3, 4, maybe 5 lines max!

I already hear you say, "3, 4, maybe 5 lines max?! What a silly rule?! There will always be functions that need longer than 5 lines! This is so arbitrary!" Yes, I am happy that this subtitle did its job. The number of lines in a function is up to the developer, but the need to make the number of lines as small as possible is critical. So my advice is to aim for 3 lines. 4 and 5 liners are ok. Above that number, there is something special about this function. Either it is doing more than one thing, or it represents a specific algorithm that is best kept as a unit.

Let's take a look at an example feature engineering function. This is for the titanic dataset [2] where the goal is to predict who died in the Titanic disaster in 1912. The dataset is a tabular dataset. The purpose of this function is to generate new features to improve the quality of our predictions. The listing below shows the feature engineering functionality:

```
def get_datasets(use_cut_points=True,
                 use_quantiles=True,
                 extract_titles=True,
                 generate_dummies=True,
                 dummies_columns=None):
    application_path = (pathlib.Path(__file__)\
                                .parent\
                                .parent\
                                .parent\
                                .absolute())

    train = pd.read_csv(application_path /
                        "katas/data/titanic/train.csv")

    test = pd.read_csv(application_path /
                       "katas/data/titanic/test.csv")

    if use_cut_points and not use_quantiles:
```

```
        cut_points = [-1, 0, 5, 12, 18, 35, 60, 100]
        label_names = ['Missing', 'Infant', 'Child',
                       'Teenager', 'Young Adult', 'Adult',
                       'Senior']

        train["Age"] = train["Age"].fillna(-0.5)
        train["Age_categories"] = pd.cut(train["Age"],
                                         cut_points,
                                         labels=label_names)
        test["Age"] = test["Age"].fillna(-0.5)
        test["Age_categories"] = pd.cut(test["Age"],
                                        cut_points,
                                        labels=label_names)

    elif use_quantiles and not use_cut_points:
        train['Age'] = train.Age.fillna(train.Age.median())
        test['Age'] = test.Age.fillna(test.Age.median())
        train['Age_categories'] = pd.qcut(train.Age,
                                          q=4,
                                          labels=False)
        test['Age_categories'] = pd.qcut(test.Age,
                                         q=4,
                                         labels=False)

    if extract_titles:
        train['Title'] = train.Name.apply(
                            lambda x: re.search(' ([A-Z][a-z]+)\.',
                                                x).group(1))
        test['Title'] = test.Name.apply(
                            lambda x: re.search(' ([A-Z][a-z]+)\.',
                                                x).group(1))

        train['Title'] = train['Title'].replace(
                                        {'Mlle': 'Miss',
                                         'Mme': 'Mrs',
                                         'Ms': 'Miss'})

        train['Title'] = train['Title'].replace(['Don',
                                                 'Dona',
                                                 'Rev',
                                                 'Dr',
                                                 'Major',
                                                 'Lady',
```

```
                                                'Sir',
                                                'Col',
                                                'Capt',
                                                'Countess',                     \

                                                'Jonkheer'],                    \

                                                'Special')
        test['Title'] = test['Title'].replace({'Mlle': 'Miss',
                                               'Mme': 'Mrs',
                                               'Ms': 'Miss'})
        train['Title'] = train['Title'].replace(['Don',
                                                 'Dona',
                                                 'Rev',
                                                 'Dr',
                                                 'Major',
                                                 'Lady',
                                                 'Sir',
                                                 'Col',
                                                 'Capt',
                                                 'Countess',                    \

                                                 'Jonkheer'],                   \

                                                 'Special')

    if generate_dummies and dummies_columns:
        for column_name in dummies_columns:
            if column_name == 'Age_categories' \
               and not use_cut_points \
               and not use_quantiles:
                continue
            if column_name == 'Title' \
               and not extract_titles:
                continue
            else:
               dummies = pd.get_dummies(train[column_name],
                                        prefix=column_name)
                train = pd.concat([train, dummies], axis=1)
                dummies = pd.get_dummies(test[column_name],
                                         prefix=column_name)
                test = pd.concat([test, dummies], axis=1)
    return train, test
```

How long do you think it would take you to understand what the function does? Could you do it in 3 mins? There are some interesting things going on in there. Some booleans are passed that impact the code path in multiple locations. There are some combinations of booleans that make the code have a specific behavior, all controlled with a sprinkle of if statements. Also, by the way, who knows if there are bugs hiding in that code?

By applying some basic refactoring and renaming, we can transform that complicated function into a bit more readable code. Try to read the following and see how you feel about the "intent" of that function. It still needs quite a bit of work to make it fully readable, but I think you get the point. The function is short, delegates further transformations to other smaller functions, and the transformation pipeline is much more composable.

```
def transform_df(input_df, use_cut_points, use_quantiles,
                 extract_titles, generate_dummies, dummies_columns):
   df = input_df.copy()
   df = fill_age_column(df, use_cut_points, use_quantiles)
   df = append_age_categories(df, use_cut_points, use_quantiles)
   df = append_titles(df, extract_titles)
   df = append_one_hot_encoded_columns(df, use_cut_points,
                                       use_quantiles,
                                       extract_titles,
                                       generate_dummies,
                                       dummies_columns)
   return df
```

Hierarchical functions

Sometimes we include many if, else, while, for in our functions to handle all sorts of unrelated concerns. In Python, there is a clear benefit of the mandatory indentation: It makes the levels of decision making apparent. There are two tricks I found in the literature that help detect these functions. The "squint test" [3] and the "landscape test" [4]. Check out this function from scikit-learn that could use some care and attention:

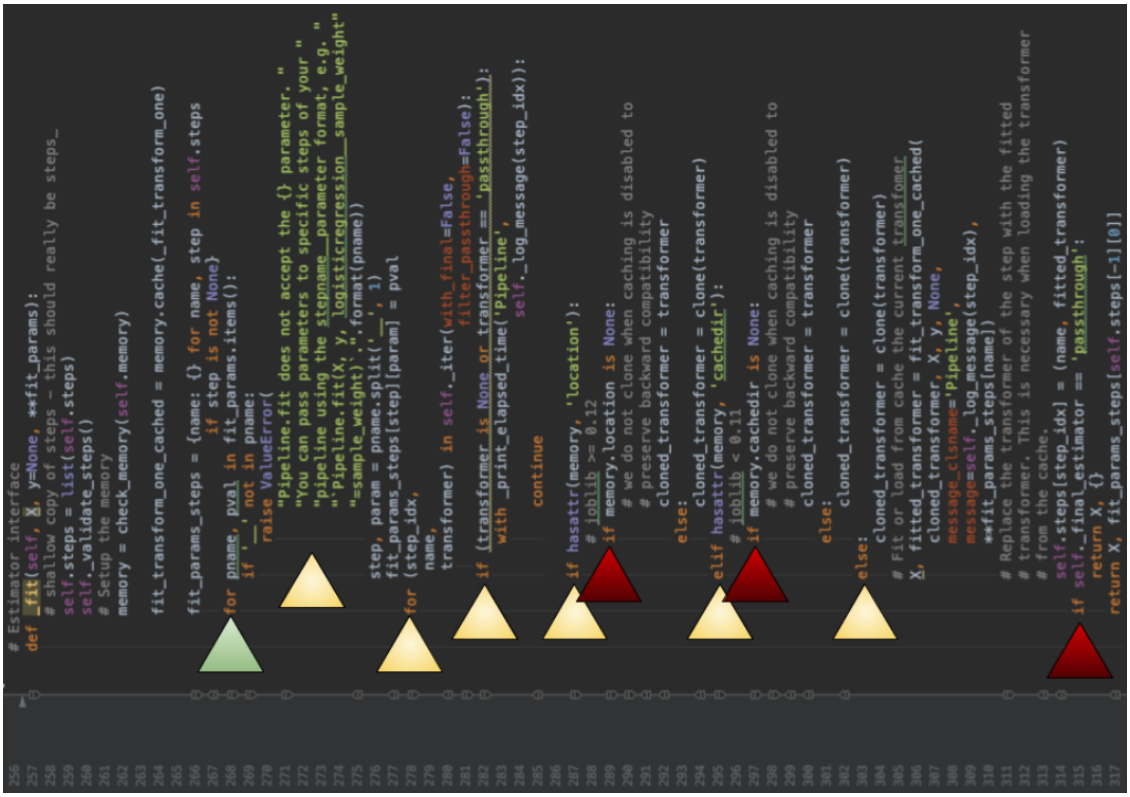
```
    # Estimator interface
    def _fit(self, X, y=None, **fit_params):
        # shallow copy of steps - this should really be steps_
        self.steps = list(self.steps)
        self._validate_steps()
        # Setup the memory
        memory = check_memory(self.memory)

        fit_transform_one_cached = memory.cache(_fit_transform_one)

        fit_params_steps = {name: {} for name, step in self.steps
                            if step is not None}
        for pname, pval in fit_params.items():
            if '__' not in pname:
                raise ValueError(
                    "Pipeline.fit does not accept the {} parameter. "
                    "You can pass parameters to specific steps of your "
                    "pipeline using the stepname__parameter format, e.g. "
                    "`Pipeline.fit(X, y, logisticregression__sample_weight"
                    "=sample_weight)`.".format(pname))
            step, param = pname.split('__', 1)
            fit_params_steps[step][param] = pval
        for (step_idx,
             name,
             transformer) in self._iter(with_final=False,
                                        filter_passthrough=False):
            if (transformer is None or transformer == 'passthrough'):
                with _print_elapsed_time('Pipeline',
                                         self._log_message(step_idx)):
                    continue

            if hasattr(memory, 'location'):
                # joblib >= 0.12
                if memory.location is None:
                    # we do not clone when caching is disabled to
                    # preserve backward compatibility
                    cloned_transformer = transformer
                else:
                    cloned_transformer = clone(transformer)
            elif hasattr(memory, 'cachedir'):
                # joblib < 0.11
                if memory.cachedir is None:
                    # we do not clone when caching is disabled to
                    # preserve backward compatibility
                    cloned_transformer = transformer
                else:
                    cloned_transformer = clone(transformer)
            else:
                cloned_transformer = clone(transformer)
            # Fit or load from cache the current transformer
            X, fitted_transformer = fit_transform_one_cached(
                cloned_transformer, X, y, None,
                message_clsname='Pipeline',
                message=self._log_message(step_idx),
                **fit_params_steps[name])
            # Replace the transformer of the step with the fitted
            # transformer. This is necessary when loading the transformer
            # from the cache.
            self.steps[step_idx] = (name, fitted_transformer)
        if self._final_estimator == 'passthrough':
            return X, {}
        return X, fit_params_steps[self.steps[-1][0]]
```

This is a function called `_fit` in the `pipeline.py` module. In the figure above, we have the squint test that helps understand the structure without knowing what this function is all about. The key is to **zoom out** and look at the levels of indentation. We can spot two for-loops, and the second for-loop has 2 different levels of conditionals with a continue in the middle.



The second test is the “landscape test” that flips the code on its side. Reading from left to right, we see the function going into multiple levels of for and if-elif-else blocks. Here we have the first for loop is ok to deal with. But the content of the loop is directly inlined. Here we are raising an exception, which could be considered fine, but extracting that in a function like _check_fit_params could be a thing. Then we notice that we are starting a second loop altogether, and that’s where the fun begins. We ask the reader to understand that we have two levels of if-elif-else embedded in the code. One of the ifs even has a continue keyword, so we need to watch out for that. Then, after all this hiking up and down these hills, we end up with a conditional that somehow invalidates all the work that was done above it.

These two visual tests are easy diagnostics to know if the function you are dealing with is doing too much. So go ahead and use that swivel in your amazing 8k screen to move code to landscape mode to see the hills and valleys in your functions. Or take a screenshot and rotate the image with an image editor.

Single Objective Function

Andrew NG, the renowned AI researcher, had an excellent course released in his Deeplearning.ai days, where he talked about how to structure a machine learning project [5]. Among the other gems in that course, he mentions that right after the problem understanding and project definition, the first thing that an ML researcher must do is to define a single numeric metric that will be used to

compare all the attempts at solving the problems. In that context, he meant choosing between the multitudes of ML metrics that exist, accuracy, precision, recall, f1-score, MAP, MRR, etc. But he made a point to dedicate a whole section of his course to this idea. To measure progress, there must be one Single Object Function that we are trying to optimize.

This applies to functions as well. This has been said in many different ways, but my favorite quote is [6]:

"FUNCTIONS SHOULD DO ONE THING.
THEY SHOULD DO IT WELL.
THEY SHOULD DO IT ONLY."

This is central to function design, but it is met with the usual question: what is "one thing"?

The previous titanic example we have the `get_dataset` function that does many things at once:

- Interacts with the file system to read data
- Performs multiple transformations:
  * Age transform.
  * Age categories generation through bucketization or quantization.
  * Titles generation with complex regex.
  * One hot encoded column generation from multiple transformed columns.
- Performs the same transformations one two different datasets.

The same function `get_dataset` operates at multiple different levels of abstraction: IO interaction, feature transformation based on existing and/or calculated columns, and ML train/test data management. All that in a single large function that is hard to read and obviously hard to maintain and change.

An important reason for using functions that are small and that do one thing, at a single level of abstraction, is to decompose a larger goal into a set of smaller tasks that can be achieved at the next level of abstraction.

By looking at the refactored version, `transform_df`, we see that we are we were able to extract the subsequent functions that compose the larger goal. This is an important rule in functional decomposition. As said in the CC book, we want to keep extracting functions from the original function until we are not able to extract new functions anymore. The stopping point of your extraction, should we when we get functions with names that do nothing more than restating the implementation of the extracted function.

Bagging and Function Ensembles

In ML, bagging is a useful variance reduction technique. We group a bunch of decision trees to produce a single ensemble by bootstrap sampling, and we expect that it will reduce the variance of our ML model.

However, in functions, we want to do the opposite. We want a single function to do a single thing. There is no need to have subsections and inner structure inside your functions to provide extra functionality. Each function should have a single responsibility and a single "thing" it is trying to perform. We can ensemble functions to do something useful later, but each function should operate as a unit with minimal inner structure and minimal knowledge about the outside world.

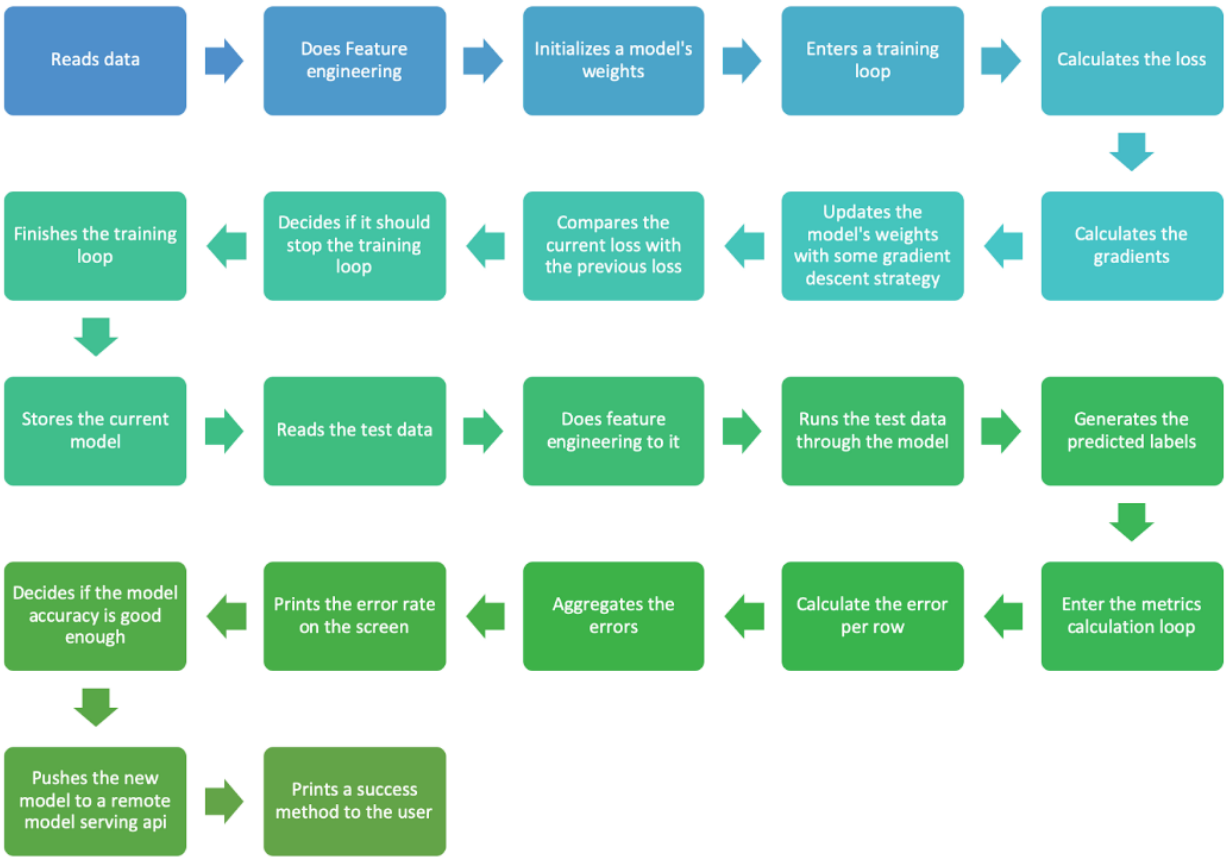
Single Abstraction Level

An easy way to check if a function is doing one thing is to look at the individual statements. You want to aim at having all the statements operating at the same abstraction level. If you notice that your functions are handling 2, 3, or more abstraction levels, then you can be sure that this function is doing more than "one thing."

From Input Data to Gradient Calculation and Back

Imaging a single function that reads data, does feature engineering, initializes a model's weights, enter a training loop, calculates the loss, calculates the gradients, updates the model's weights with some gradient descent strategy, compares the current loss with the previous loss, decides if it should stop the training loop, finishes the training loop, stores the current model, reads the test data, does feature engineering to it, runs the test data through the model, generates the predicted labels, enter the metrics calculation loop, calculate the error per row, aggregates the errors, prints the error rate on the screen, decides if the model accuracy is good enough, pushes the new model to a remote model serving API, and finally prints a success method to the user.

Example of multiple abstraction levels in one function



Just by reading the above, you are probably getting a gut feeling that we are not operating at the same level of abstraction for all these operations. For example, we are mixing accessing raw data, a low-level concern that interacts with an io filesystem, with feature transformation functionality, that is purely business logic. Then we are mixing loss calculation, which is a generic concern, with low-level details of precise gradient calculations with specific model parameter updates. Then we are intermixing training, testing, and model publishing concerns, which is again mixing internal business logic with external interactions with a remote API. I believe that should give you an idea about how to determine if your function is doing "one thing." Each statement in your functions should operate at the same level of abstraction. If you see mixing, then extract the low-level details of the next layer of abstraction, in a separate callable function that can be composed and combined with other functions at the next abstraction layer up.

Function Arguments

I am personally a fan of functions that do not require more than 3 arguments. 0 arguments is ideal. 1 to 2 arguments are fine. Three arguments are ok if args have good names and are not all booleans. Beyond that, I get confused about the role of each parameter. I suspect I am not the only one in this situation.

Stop at Triadic, Run From Polyadic

Testing functions that take multiple arguments gets complicated. We don't like complications. We need to ensure that all the combinations of arguments are working properly. With 0 arguments, we don't have to worry too much about the behavior of the function due to different inputs. With one argument, monadic functions are still clearly defined. With two arguments, dyadic functions start to be harder to check with all its combinations. With three/triadic or more/polyadic, we need to spell out all the combinations of arguments that would change the behavior of the function. This is very much like a linear model with many regressors. It is a lot easier to explain the behavior of a model with 2 predictive features than one with 10 features.

Output arguments

Output arguments should be avoided. It can be tempting to pass in a dataframe to a function and make the function change the dataframe in place. This pattern is common in data pipelines when we want to pass a dataframe, or some other data object through a series of transformations. We will discuss data transformation strategies in detail in the section "Have No Collateral Damage." However, there are clearer infractions of this for non-transformational operations. For example, a function called push_model(model) that supposedly pushes the model to some serving layer, should not modify the model's attributes. We will talk about this issue in more detail in the "CQS: Command-Query Separation" section.

```
def push_model(model, serving_endpoint):
    model_json = model.to_json()
    serving_endpoint.post(model_json)
    model.uploaded=True
    return model
```

Binary Args

Binary args are booleans we pass into functions to modify their internal behavior. This is common in feature engineering pipeline where, for example, we need to enable the one-hot encoding, disable generating the embeddings, enable the "remove stop words", enable to approximate count method for a Dict vectorizer, disable the probability estimation, and finally enable using the mean but not the standard deviation when scaling the numerical columns.

This configuration pattern is all too common and increases the number of cases that need to be tested. The CC school of thought advises against booleans because it immediately complicates the signature and the logic inside the functions, leading to a "more than one thing" situation. Using a flag in a function points to the fact that the function is doing more than one thing.

However, for feature engineering and ML pipelines using booleans with some level of discipline can be worked into your code. One or Two booleans that configure some feature engineering step is not too bad as long as they are independent.

In the example below, we have a feature transform that does some scaling operations that depend on the mean and/or the standard deviation. In the complex scenario, feature_transform_simple, we have an interlocked if/elif/else condition, which makes it hard to reason about the control flow of the feature transformation. In the nicer example, feature_transform_simple, we still accept two flags, but now the pipeline is laid out as a sequence of independent operations. The more straightforward second function is easier to reason about, with the added benefit that we are operating on separate columns altogether. The top-level function, feature_transform_simple, can be thought of as "doing one thing," which is simply orchestrating the pipeline and delegating the work to functions at the next abstraction level.

```
# Bad
def feature_transform_complex(input_df,
                              with_mean=True,
                              with_std=False):
    if with_mean and with_std:
        df = scale_with_mean_and_std(df)
    elif with_mean :
        df = scale_with_mean(df)
    elif with_std:
        df = scale_with_std(df)
    else:
        continue
    return df

# OK
def feature_transform_simple(input_df,
                             with_one_hot_encoding=True,
                             with_remove_stop_words=False):
    if with_one_hot_encoding:
        df = gen_one_hot_encoding(df, column="city")
    if with_remove_stop_word:
        df = remove_stop_words(df, column="text_comment")
    return df
```

So use your good judgment with flags. If the flags are independent of each other, then they can be a useful technique to orchestrate your functions. However, if you find yourself combining flags in unhealthy ways, then don't do it.

Clustering Arguments

When dealing with a lengthy list of configuration objects that get passed to functions that orchestrate complex model training or feature engineering tasks. In the example below, we group arguments in

a configuration object that can be passed to the function. The function would have to extract the arguments and use them in its function body. This grouping can be abused if the argument operates at a different level of abstraction or are unrelated. However, they are configuring a shared concept. It is encouraged to add them as a configuration object.

```
#Too many arguments
def feature_transform(df,
                      with_mean=True,
                      with_std=False,
                      copy=True,
                      with_max_clipping=True,
                      with_min_clipping=True,
                      with_one_hot_encoding=True,
                      with_generate_embeddings=True):
    pass


# Better
# Consolidated function argument in an object
class FeatureConfig:
    with_mean: bool =True
    with_std: bool =False
    copy: bool =True
    with_max_clipping: bool =True
    with_min_clipping: bool =True
    with_one_hot_encoding: bool =True
    with_generate_embeddings: bool =True

def feature_transform(df, config: FeatureConfig):
    # Extract and use the configuration values needed.
    pass
```

Positional, Keyword, Args, and Kwargs

Python has some flexible arguments management. This includes positional, keywords, *args and **kwargs.

When choosing between positional and keyword arguments, I advise to always choose keyword arguments. Yes, they add a few keystrokes when adding the parameters, but the readability is greatly improved. For example `build_model(size, "high")` vs `build_model(model_capacity=size, model_threshold="high")`.

Concerning the `*args`, remember that args is only a name, so if you are passing a list of arguments that represent a variable-length array of similar objects, then use the names of those object concepts.

For example, `union_feature_columns(*args)` vs `union_feature_columns(*feature_columns)`.

`**kwargs` is the last one and should be avoided when possible. It is a powerful technique that allows the caller to pass in a variable-length set of key-value pairs. However, the function arguments become pretty opaque. For example, `push_model(**kwargs)` is as flexible as possible, but we could be considered too flexible. What are the expected arguments? How are we unpacking them? What are their types and the concepts they represent? We also lose any kind of intelligent code completion from IDEs. `**kwargs` can be a useful tool for your functions when you don't want to mess with a public API, and still accept a list of key-value pairs arguments. Use with care.

Have No Collateral Damage

You probably heard of this situation: The user calls a function that promises to add two numbers together but ends up sending an email to a client. Some functions do mysterious implicit things that are not expected from the way they are named. This behavior is called "side-effects," and we don't like side effects. Yes, we DON'T like side effects because they are lies [6]. Side effects can sometimes appear in a function modifying its object instance variables. Sometimes it does so with output arguments getting modified, and sometimes it impacts a global variable or some other state outside the realm of these functions, which by the way, should be SMALL!

Don't Use Output Arguments

The reader should not have to keep in their mind the fact that we are going to modify some variable that was passed in and return it to the user. Not only does it break the mental flow of the reader, but it is also considered an unwanted side effect.

```
def fill_na(df):
    df["Age"] = df["Age"] = df["Age"].fillna(df["Age"].median())
def get_is_adult_column(df):
    is_adult = df["Age"].apply(lambda x: 0 if x<21 else 1)
    return is_adult

fill_na(df)
#>>>>>>10 lines later
is_adult = get_is_adult_column(df)
df["is_adult"] = is_adult
#>>>>>>10 lines later
avg_age = get_avg_age(df)
df["avg_age"] = avg_age
```

Take a look at the above code. There is an implicit understanding that the `get_is_adult_column` function expects a dataframe with a column age that was already pre-processed. This relationship

between the `fill_na` and the `get_is_adult_column` functions is called a **temporal coupling**. We are encoding the fact that one function should happen before the next one. Running the `get_is_adult_column` without running the `fill_na` function would return strange results. This especially annoying because of the name of the `fill_na` function. There is nothing there that describes what it will `fill_na` in the "age" column and that it is pre-processing step for the next function. In addition, the `fill_na` has the global effect of impacting any functions that come after that. In our case, we have the `get_avg_age` function that will be silently impacted by the `fill_na` upstream. The order matters where the average age is definitely impacted by imputing the median age.

Side-effects in Feature Engineering Pipelines

It is extremely common to structure feature engineering pipelines as a set of transformations. The pipeline concept promoted in the major ML libraries is implemented one way or another in any ML pipelines. We take input datasets, and we pass them through a set of functions to generate model readable columns and/or more predictive columns. This includes things like dictionary vectorizers, PCA transformers, one hot encoder, and many other types of feature engineering techniques.

The side-effect problem in feature engineering pipelines occurs when we are tempted to modify the input data in place. This is done to save memory from getting filled up. Or to save keystrokes so that our keyboards stay healthy because we care too much about our high tech mechanical keyboards. In the following examples, I am using the pandas dataframes as the primary method of data transfer, but the ideas are applicable to numpy arrays or any other tabular formats popular with ML libraries.

Option 1: Copy everything

One strategy available to avoid the side effect of modifying incoming datasets is to make a copy of any data coming in the function and returning a fresh copy, with all the modified columns and features in the new dataframe. This way, the original dataframe is not impacted. The drawback of this is that we are making copies of the data, which might accumulate as the processing pipeline grows. This should not be an issue for small datasets. Periodically the garbage collection machinery of Python will do the right thing and delete the unreferenced copies.

Example of a dataframe being copied to avoid side effects:

```
1  def fill_age_nan(input_df):
2      df = input_df.copy()
3      df["Age"] = df["Age"].fillna(df["Age"].median())
4      return df
```

```
def append_is_adult_column(input_df):
    df = input_df.copy()
    df["is_adult"] = df["Age"].apply(lambda x: 0 if x<=21 else 1)
    return df

df = fill_age_nan(df)
df = append_is_adult_column(df)
```

Option 2: Return the new feature column to the caller

A second approach is to make the function take in the input dataframe, but only return the transformed columns as a separate object. The caller then can append the new column/columns to the original dataframe. This prevents the function from modifying the dataframe, at the cost of adding some level of noise in the caller function.
Scikit-learn and pyspark promote this kind of operation, where they encourage developers to format their feature transforms as feature generators, and then to use FeatureUnion transformers to combine the generated features in one feature matrix. One can use these concepts without having to actively use either of these libraries.

Example:

```
def get_median_age(input_df):
    median_age = input_df["Age"].median()
    return median_age

def get_is_adult_column(input_df):
    is_adult = df["Age"].apply(lambda x: 0 if x<21 else 1)
    return is_adult

median_age = get_median_age(df)
df["Age"] = df["Age"].fillna(median_age)
df["is_adult"] = get_is_adult_column(df)
```

Option 3: Append-only inside the functions

When your dataset grows large, it becomes prohibitive to make copies of the data. The strategy used in this case is to avoid making copies to modify existing columns and only append new ones to your dataframe. This preserves the original data, which will help you debug issues later on when you need to compare calculated vs. original columns. This permits post-processing lineage understanding. For example, pyspark circa 2020 promotes this method where the lineage of the transformations is recorded lazily and applied at once on the whole dataset as a chain of transformations. For pandas, this strategy can also be applied by convention. Any function that is applied to the input dataframe is only allowed to add new columns to the dataframe.

Example:

```
def fill_age_nan(df):
    df["Age_no_nan"] = df["Age"].fillna(df["Age"].median())

def append_is_adult_column(df):
    df["is_adult"] = df["Age_no_nan"].apply(
                                    lambda x: 0 if x<=21 else 1)

fill_age_nan(df)
append_is_adult_column(df)
```

Functional Programming 101

Functional programming was created early on in the 1930s with the advent of lambda calculus [7]. It is a deep and fascinating topic. However, three central concepts are useful to keep in mind when designing functions: Pure functions, immutability, and idempotence.

First, the concept of pure functions is related to pure mathematical functions. This sort of function always returns the same output when presented with the same input. They do not rely on any external state to perform their operations. Second, immutability tells us to only create new variables when using assignments. Nothing that existed before the function should be touched by our functions. This is related to not having side-effects. Finally, idempotence is the ability of a function to be called multiple times without any cumulative impact.

Not every function can be designed with these properties. After all, we write software to impact the external business world around us. However, the internal application functions that do not interface with the outside world should be designed that way. There are quite a few benefits:

- Removing a pure function is, by definition, a zero-impact action. No side effect means functions can be removed without fear of change.
- The performance of the functional style is also beneficial. Since we get the same results from calling a pure function with the same arguments, that means that we can cache and memorize the results of the function and use the same result on the next call.
- When two pure functions have no data dependencies, we can reorder their operations freely when using multiple threads without fear of stepping on each other's feet.
- Finally, TESTING pure functions are as simple as it gets. These functions do not depend on any external state and are therefore, predictable. We can present a set of inputs and be confident that the function will always return the same output for the same input.

Make Temporal Couplings Explicit

In the examples above, we can see how temporal couplings are getting created because we depend on columns to exist in specific formats, ranges, names, etc. This means that one feature transform expects to find some columns in the input data. Changing the ordering of the functions changes

the core output of the features engineering pipeline. We are usually not able to completely remove temporal couplings because they are an intrinsic component of data pipelines. However, we can at least make them explicit. This is done by simply grouping the temporally coupled function calls in a block. The primary method for resolving temporal couplings in Python are context managers. You probably have seen the "with" statement for opening and closing files.

```
with open("my_file") as file:
    data = file.read()
```

A similar pattern can be used to avoid leaving crummy side-effects columns in your dataframes:

```
def append_age_based_features(df):
    original_age = df["Age"].copy()
    fill_age_nan(df)
    append_is_adult_column(df)
    append_is_senior_column(df)
    append_is_millenial_column(df)
    df["Age"] = original_age
```

This function does two important things. First, it records the original values in the age column. Second, it groups the functions that depend on the processed age column in a single block. This makes reliance on the age column explicit. Finally, it resets the age column to its original values. Subsequent functions that rely on the age column will be able to access the original value without having to deal with temporal coupling.

Grokking Commands vs. Queries

Understanding the difference between commands and queries functions can help separate functionality in more understandable blocks. In essence, it is useful to think of functions as able to do two separate activity types:

- **Command**: A function is changing some external "state". For example `write_saved_model()`, `push_features_to_db()`, and `publish_metrics()` are functions that have external impacts.
- **Query**: a function is returning some "information". For example `get_age()`, `get_income()`, and `get_f1_score()` are functions that have no side effects and are only used to get back some output to the caller.

The key idea is to avoid as much as possible, merging these two activities. This is called Command Query Separation (CQS). For example, look at this function:

```
#Merged Command and Query
def fill_age_nan(df):
    df["Age"] = df["Age"].fillna(df["Age"].median())
    return df
```

`fill_age_nan(df)` seems to return a dataframe that the caller can use. However, it is effectively changing the state of the `df` dataframe column "Age". This function is doing both a query and a command. By writing this sort of function, we are forcing the user to do a double-take. "Is this function filling the dataframe in place?" or "Is this function returning a copy of the dataframe with the age column modified?". Calling this function is also problematic. Do we use `df=fill_age_nan(df)` to get the updated dataframe, or just `fill_age_nan(df)` assuming that the dataframe will change in place? The right thing to do is to either do the in-place replacement and not return anything or return a copy of the column with the modified values.

```
#Query version
def get_age_with_no_nan(df):
    return df["Age"].fillna(df["Age"].median())

#Command version
def fill_age_nan_inplace(df):
    df["Age"] = df["Age"].fillna(df["Age"].median())
```

Promote The Happy Path With Exceptions

One useful practice is to use exceptions instead of return codes. If something goes wrong in your code, it is better to use exceptions to notify the caller of the problem. On the other hand, using return codes forces the caller to deal with the problem immediately.

```
# Bad
def reset_experiment_components():
    success = 1
    if reset_accuracy_metrics() == success:
        if model.reset_trainable_layers() == success:
            if model.reset_gradients() == success:
                print("metrics and model reset successfully")
            else:
                print("model gradients reset failed")
        else:
            print("model layers reset failed")
    else:
        print("Unable to reset accuracy metrics")
```

As we see in the example above, the number of cases, and indentation levels, grow. This becomes harder to read and maintain because the cases are interlocked in a chain. A solution for this would be to have a custom exception for each case. The excessive indentation disappears, and we delay the handling of the exceptions to later. This makes the code more readable and more maintainable.

```
# Better
def safe_reset_experiment_components():
    try:
        reset_accuracy_metrics()
        model.reset_trainable_layers()
        model.reset_gradients()
        print("metrics and model reset successfully")
    except MetricsException:
        print("Unable to reset accuracy metrics")
    except ModelLayerResetException:
        print("model layers reset failed"
    except ModelGradientResetException:
        print("model gradients reset failed")
```

Separate the Happy Path from the Outliers

We are still faced with one more optimization. The function above does more than "one thing." It handles the exceptions but also runs the business logic. This can be easily solved by extracting the business logic into its own function as follows:

```
# Good
def safe_reset_experimentcomponents():
    try:
        reset_components()
    except MetricsException:
        print("Unable to reset accuracy metrics")
    except ModelLayerResetException:
        print("model layers reset failed"
    except ModelGradientResetException:
        print("model gradients reset failed")

def reset_components():
    reset_accuracy_metrics()
    model.reset_trainable_layers()
    model.reset_gradients()
    print("metrics and model reset successfully")
```

This promotes separation between error handling and business logic. If we need to change the business logic, we will not be slowed down by the error handling code.

Don’t Reuse Unrelated Exceptions Types

One pattern that happens when using the return code is unrelated reuse. Say we have built a set of custom return codes:

```
error_codes:{
    'ImportError':1
    'NotFittedError':2,
    'ChangedBehaviorError':3,
    'ConvergenceError':4,
    'DataConversionError':5,
    'DataDimensionalityError':6,
    'FitFailedError':7,
    'UndefinedMetricError':8,
}
```

When using such a pattern, all the functionality that needs these error_codes, start depending on this dictionary. Any change to these error_codes leads to code changes in all the callers. This leads to a fear of changing this error_codes dictionary. The developers start reusing the error even if they are not directly related instead of adding new ones, from fear of messing up other unrelated code.

The better method is to create custom exceptions classes with minimal efforts. When a new exception type is needed, we can add new exceptions classes that are specialized for the case at hand. This reduces the dependency pressure on the bundled error codes and removes the fear of messing up some other part of the codebase just to handle a new exception case.

```
# define Python user-defined exceptions
class Error(Exception):
   """Base class for other exceptions"""
   pass

class ImportError(Error):
   """Raised when the import fails to find target package"""
   pass
class NotFittedError(Error):
   """Raised when calling an estimator that was not fit yet"""
   pass
class ConvergenceError(Error):
   """Raised when training reached last iteration without loss convergence"""
   pass
```

Eliminate Duplicates, Doubles, and Homologues

No function best practices would be complete without the DRY principle. Notice that in the titanic listing, we started with a nasty duplicated section to make sure we apply the same transformations to the training and testing datasets. This is necessary to end up with the same feature space. However, we don't need to copy-paste at every step of the way. We handled that by extracting all the functionality and applying the same transformations to the training and testing data by raising the abstraction level to a "dataset", regardless if it was training or testing. This is usually the process of removing duplication. In almost every duplication, there is an abstraction hiding and waiting to be extracted.

One word of caution: do not prematurely remove duplication. Our IDEs allow us to remove duplication pretty easily with full code block replacements. However, we need to wait for the duplication to be ripe enough to be removed. "Code duplication is better than the wrong abstraction"[3] is one way of thinking about it. We sometimes jump at first sight of duplicate code. However, it is useful to wait and copy-paste away until the full meaning of the duplication level expresses itself.

Single Entry, Single Exit

Structured Programming 101

Dijkstra was a strong supporter of structured programming. He promoted the view that every function and every section within a function should have one entry point and one exit point. In python, we have the ability to break the control flow with break and continue. These go against the rule of having a single entry single exit system with a single return statement. The break and continue constructs allow the programmer to multiply the exit paths of a function or block. This decreases readability.

Long functions are one place where this is most annoying. Tracking down a loop or grapes of if statements to find a break/continue statement, breaks the flow of thought when thinking about code. If you are writing a single full algorithm that needs to terminate before the end of a loop due to a convergence condition, then it is fine to use a break or continue to do so. However, if we are mixing multiple actions in the same function and using break/continue to navigate across multiple functionality paths, then we are breaking two rules. The function is doing more than one thing, and it is doing it in an erratic manner that is hard to read. Keeping functions small automatically reduces the impact of break/continue because the scope is small by default. Let's aim for small functions.

A Method to the Madness

"Write drunk, edit sober" [8], while not advocating for any substance usage of any kind, there is some truth to the writing process. Do not try to write the perfect first draft. Get in a mental state of putting all the ideas on the screen. And only then, refactor, rearrange, minimize, summarize,

synthesize. The first draft of any function will be a hodge-podge of copy-pasted code and formulas copied from tutorials, docs, previous projects, and other files you just finished editing. This is fine, and that is the right process to follow.

The initial pass at a function is a foggy mess of assumptions and complications. Nested structures, loops that end on Tuesdays at 3 pm, and more argument lists that replicate like rabbits. Names are smashed together in a frenzy, duplicated code is slapped together, and anything goes. But then the code starts working, and we get a small set of tests in place. Then comes the next phase.

With the tests in place, we are not too afraid of changing the code we just wrote. So we rename that metric calculation, eliminate that train/test duplication, reorder the feature engineering steps, extract full methods and combine them in cleaner ways. All of this, while the tests are guaranteeing that the two faces of our code still do the same things.

There is no way anyone could write clean and organized code from the start. First, put the ideas on the screen, and only then refactor them into something clean and elegant.

Conclusion

Your past, current, and next machine learning pipeline does not live in a vacuum. It is a representation of a specific domain. This domain is represented with verbs for functions and nouns for classes to represent the flow of information that needs to happen to generate value from data. It is an art, and all machine learning engineers are constantly translating customer feelings and hunches into their interpretation of what the system should do.

As more and more ML and data projects land in our hands, we start noticing commonalities and related storylines. Functions are the basic building block for transforming a business problem into a series of acts and reversals that tell the story of your predictive systems.

We covered the inner workings of writing well-defined functions. The main message that we tried to cover is "Make Smaller Things." Software has the tendency to grow uncontrollably. It is our role to stop this sprawl with smaller, more manageable components, and small, well-named functions are our first line of defense.

References

[1] Rails Conf 2013 The Magic Tricks of Testing by Sandi Metz https://www.youtube.com/watch?v=URSWYvyc42M
[2] Titanic Dataset: https://www.kaggle.com/c/titanic
[3] RailsConf 2014 - All the Little Things by Sandi Metz https://www.youtube.com/watch?v=8bZh5LMaSmE
[4] https://cleancoders.com/. Write Code that Sings. Accessed 2020-06.
[5] Structuring Machine Learning Projects. https://www.coursera.org/learn/machine-learning-projects?specializatio
learning
[6] Clean Code: A Handbook of Agile Software Craftsmanship. Robert C. Martin. 2008. Prentice Hall
PTR, Upper Saddle River, NJ, United States

[7] https://en.wikipedia.org/wiki/Functional_programming
[8] Attributed to Hemmingway but contested

Chapter 4 - Style

We are in the golden age of IDEs, the 2010s to 2020s. IDEs can format code with a single press of a button. The long lost art of code formatting is disappearing and is not a problem anymore. We live in an age of orderly, consistent, and reliable tooling that can transform any mess into screen artwork. Easy to read, easy to maintain, and easy to admire.

"Wait a second," you say, "my code does not look anything like that, I have Jypyter notebooks and Python scripts that are virtually impossible to change or use because they have so much clutter and rotting lines of non-functional elements that I gave up on using them."

That's correct, joke aside, it is the state of most code that is being built under business pressures. We start our next project with the best intentions, but right after the first sprint, product managers start breathing down our neck to understand, "why is it taking so damn long to get this feature developed!".

We hack things together for a demo and forget to refactor the code. 2 or 3 sprints pass, and then there is a quarterly review, so we speed to the finish line by doing the bare minimum work to get the application working. This leads to a rodent-infested swamp full of working code that is hard to read. Then the final nail in the coffin gets placed: "We need to format and document this code so that v2 developers can find their way around". This is the topic of this chapter: Comments and Formatting.

Comments

Comments can be lifesavers. We have all been there where we want to use an ML library complex functionality and find an excellent summary of the function right above the code in a short, well-written comment. Comments can also be an anxiety-inducing thing to read. Say we want to use a function that we thought was well contained and having no side-effects, only to find in the comments, "Be careful with this function. I coded this ensembling strategy straight from pseudo-code I found in section 4.4 of 'The Elements of Statistical Learning' book. I think it works most of the time. Good luck!"

Our IDEs and programming languages have come a long way to provide expressive constructs that remove the need for comments. But unfortunately, newly minted PhDs in Statistical Learning or Masters in Data Science do not have the necessary knowledge to express their ideas with all the tricks and tips that modern languages such as Python can provide. This is usually remediated with comments, lots of them.

This is a pretty widespread process. We use comments to fill in the knowledge gaps that our code is not able to express. Comments are about "filling gaps," and they should be treated as such. If our code were able to express everything it was doing there would be no gaps to fill. We would be able

to carry on with our day without relying on narrative methods to explain the intricacies of our latest flavor of gradient descent.

Every time you type that '#' sign to start a comment, you should feel the cavernous gap that you are trying to patch. What is this gap you are trying to fill, and can't you fill it with more expressive code?

Comments are useful, don't get me wrong. However, they are hard to maintain. Code changes, features that were built one way are built another, models default change, and deployment assumptions fluctuate. Comments are hard to maintain, even with the best of intentions, because there is no direct way to automate their correctness, relevance, and redundancy levels.

See the below example of a simple feature engineering block that changed over time, but the comment did not get updated. Reading the comment, we see that we are extracting the titles Mrs and Mr, but the code itself is extracting Miss and Mrs, expecting to find Mr already in the title. Who wants mixed messages in their functions? I don't think you want that either.

```
def append_titles(self, input_df, extract_titles):
   df = input_df.copy()
   if extract_titles:
       # Extract title Mrs and Mr from the title,
       # and set everything else to special
       df['Title'] = df.Name.apply(
                     lambda x: re.search(r' ([A-Z][a-z]+)\.', x).group(1))

       df['Title'] = df['Title'].replace(
                                  {'Mlle': 'Miss',
                                   'Mme': 'Mrs',
                                   'Ms': 'Miss'})
       df['Title'] = df['Title'].replace(['Don', 'Dona', 'Rev', 'Dr',
                                          'Major', 'Lady', 'Sir', 'Col',
                                          'Capt', 'Countess', 'Jonkheer'],
                                         'Special')
   return df
```

ML developers, like any other software engineer, only have a limited amount of energy and time per day. They can spend their time updating comments to match the code they are describing. Or they can spend their energy on making their current and next feature transformation code more readable, smaller, and more maintainable so that they can skip commenting their code altogether.

It has been said many times, but "Truth can only be found in one place: the code"[1]. Only the code has the exact behavior of your next model spelled out with undeniable levels of detail. Comments can augment and clarify, but in the end, the code is what matters.

Don’t Hide Bad Code Behind Comments

“I mean it works, but I should really add comments to this!” said every ML engineer ever. This phase is said loudly or silently in our inner monologue. The issue is that ML software quickly gets complex and needs constant rework to get right. When time is tight, we skip the rework and leave the bad code for later. This is where comments show up with words such as “Ugly hack!” “We need a better way for this algo data ingestion”, “Q3 reporting is due, and I had no time to refactor this. Q4 goals!” So we spend time explaining in plain English the approximate workings of an Evaluator MAP@K metric function instead of spending that time refactoring this function to a more readable format. Next time you find yourself deep in a 10 line comment, maybe use the same time to rename some variables, extract some functions and make the function document itself.

Let Code Explain Itself

Yes, code is not plain English and sometimes needs some explaining. However, this is taken to an extreme by developers that say, “… this embeddings generator function is straight out of a paper I read two years ago. The authors gave the pseudo-code in terms of linear algebra equations. I just converted the whole thing to numpy ndarrays….” Does that mean the code should be math-based as well? Not at all. While math on an academic paper is compressed to its more important components to fit in a publication format, the code we write to represent that research does not need to do the same. Code has further abstraction mechanisms that allow the ML dev to group similar functionality in functions, for example. See the example below:

```
# Check to see if the model finished training because of convergence
if model.train_epoch!=MAX_EPOCH && \
   model.last_loss_difference <= MIN_LOSS_DIFF:
```

Or this?

```
if model.training_converged(MAX_EPOCH, MIN_LOSS_DIFF):
```

Useful comments

Legalese Comments

Another source of legitimate comments is corporate rules. The data science compliance office might need you to comment your ML code to match the standards of the legal team. Things like trademarks, copyrights, rules for usage, open source usage, and the like used to be placed in every file. Please automate that as soon as you can with a script that reads each file and adds the legalese on top. By following some standards in comment formatting, your IDE might even be able to collapse that and avoid cluttering your code.

```
# Copyright 2019-2020 The Magic ML Package Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Lately, for OSS software, with the advent of Git as a solid distributed version control and Github/Gitlab/Bitbucket/Stash as the central location for collaboration, I see more developers removing those comments from the source files altogether. There is usually a replacement in a LICENSE file that is used to provide a license for the whole repository.

```
COPYRIGHT
All contributions by Company Magic Machine Learning:
Copyright (c) 2015 - 2019, A, Inc.
All rights reserved.
Each contributor holds copyright over their respective contributions.
The project versioning (Git) records all such contribution source information.
LICENSE
The MIT License (MIT)
```

Legitimate PSA Comments

First, there is the PSA comment that comes before from a delayed rename. This PSA tells the user more details than the function name is able to convey. In the following example, the comment tells the users that we are training a model in a streaming fashion. The function name does not point to that immediately. The user will have to examine the comment and the code as well to make sure that it is a streaming model training function.

```
# Trains the model in a streaming fashion.
def train(input_data, batch_size=100)
```

A better scenario is to rename the function itself to say what it does.

```
def train_from_streaming_data(input_data, batch_size=100)
```

Another example is below. This is another reasonable PSA comment usage:

```
def append_age_categories(self, input_df, use_cut_points, use_quantiles):
    """
    This function makes a copy of the input dataframe.
    If the cut_points arg is true then the age column is bucketized.
    If the quantiles arg is true then the age column is quantized.
    The new column is assigned and the new dataframe is returned
    """
    df = input_df.copy()
    if use_cut_points and not use_quantiles:
        cut_points = [-1, 0, 5, 12, 18, 35, 60, 100]
        label_names = ['Missing', 'Infant', 'Child',
                       'Teenager', 'Young Adult', 'Adult',
                       'Senior']
        df["Age_categories"] = pd.cut(df["Age"], cut_points, labels=label_names)
    elif use_quantiles and not use_cut_points:
        df['Age_categories'] = pd.qcut(df.Age, q=4, labels=False)
    return df
```

This kind of PSA gives us directions on how the full name will be parsed and how the title of each person will be formatted. This passes the usefulness bar, but if there is any more logic that needs to be added to this function, then we would be better served by extracting the individual functionalities and giving them good names:

```
def append_age_categories(self, input_df, use_cut_points, use_quantiles):
    df = input_df.copy()
    if use_cut_points and not use_quantiles:
        df = self.bucketize_age(df)
    elif use_quantiles and not use_cut_points:
        df = self.quantize_age(df)
    return df
```

Why did you do it that way?

When you find yourself making a design decision that could have gone either way with no directly provable benefit, your users and teammates might end up saying to themselves: “I see what he is doing but I wonder if he looked at alternative implementations and why he picked this particular path?”. These are reasonable questions for users and teammates that need to edit and maintain your code. See the following example from the Spacy library:

```
def beam_parse(self, docs, int beam_width, float drop=0., beam_density=0.):
    ...
    beams = self.moves.init_beams(docs, beam_width,
                                  beam_density=beam_density)
    # This is pretty dirty, but the NER can resize itself in init_batch,
    # if labels are missing. We therefore have to check whether we need to
    # expand our model output.ss
    self._resize()
    model = self.model(docs)
    token_ids = numpy.zeros((len(docs) * beam_width, self.nr_feature),
                            dtype='i', order='C')
```

When you find yourself making a design decision that could have gone either way. With no directly provable benefit, your users and teammates might end up saying to themselves: "I see what he is doing, but I wonder if he looked at alternative implementations and why he picked this particular path?". These are reasonable questions for users and teammates that need to edit and maintain your code. See the following example from the Spacy library:

```
def _mini_batch_step(...):
    """Incremental update of the centers for the Minibatch K-Means algorithm.
    ....
    """
    # .....
    # reset counts of reassigned centers, but don't reset them too small
    # to avoid instant reassignment. This is a pretty dirty hack as it
    # also modifies the learning rates.
    weight_sums[to_reassign] = np.min(weight_sums[~to_reassign])
```

Semantic Mapping

Frequently, we end up dealing with libraries we have no control over. The libraries have APIs that are outside of our control. So we go and explore the real meaning of the functions we are using that are hiding behind a veil of bad third-party naming. We become the local expert in these names. One beneficial way to use comment is to report back on these names and create a semantic mapping that can bring some clarity to obscure third party API naming convention.

Say we end up using a proprietary linear algebra that was adapted from an R package. This package uses a set of abbreviations to represent matrices.

```
# calculate the singular value decomposition of a sparse matrix
math_lib.sp_mat_svd(input_matrix)
# calculate the jacobian of a low rank matrix
math_lib.lr_mat_jac(input_matrix)
# calculate the matrix multiplication of two matrices
# that are assumed to be positive semi-definite matrices.
math_lib.app_mat_mul_psd(input_matrix_1, input_matrix_2)
```

In this case, we are not able to rename the functions of the third-party package, and we might not even have the original source case for it. In this case, clarifying function calls with localized comments can be beneficial. However, if you find yourself duplicating the same function calls comments every time they are called, then it is better to wrap those functions inside a "semantic adapter" of sorts that would be more readable:

```
def calculate_low_rank_matrix_jacobian(input_matrix):
   return math_lib.sp_mat_svd(input_matrix)
```

Red Flags Planted in the Sand

Here is one example comment (from the famous spacy library) that warns the users of a complicated implementation ahead:

```
def MultiHashEmbed(config):
   """For backwards compatibility with models before the architecture registry, we h\
ave to be careful to get exactly the same model structure. One subtle trick is that \
when we define concatenation with the operator, the operator is actually binary asso\
ciative. So when we write (a | b | c), we're actually getting concatenate(concatenat\
e(a, b), c). That's why the implementation is a bit ugly here."""
```

In this case, the "subtle trick" the function author is pointing to, is worth mentioning to the reader of this function.

Here is another example from that same library that would benefit from more details:

```
# Currently segfaulting, due to l_edge and r_edge misalignment
def test_issue1537_model():
  nlp = load_spacy('en')
  doc = nlp('The sky is blue. The man is pink. The dog is purple.')
  sents = [s.as_doc() for s in doc.sents]
  print(list(sents[0].noun_chunks))
  print(list(sents[1].noun_chunks))
```

Another good place to warn other developers is with optimizations.

Warning the users that we already tried a couple of tricks to optimize this function and, this is the best way we found. In the following example, we remind the user that we are updating a global variable in a loop that will create a race condition if put in a parallel loop. While there are probably some better ways to fix this, the comment warns against wanting to optimize this loop:

```
def count_global_lines_in_files(files):
   """ Do not parallelize this loop. We are handling legacy code that requires the e\
xistence of the total_number_of_lines counter in the global scope. Do not multi-thre\
ad or multi-process until the global variable issue is resolved."""
   for file in files:
       with open(file) as f:
           lines = f.readlines()
           total_number_of_lines += len(lines)
   return total_number_of_lines
```

Eventual Consistency with TODO Comments

TODO is a helpful method to tell your future self that something needs fixing. As the number of TODOs grows in your codebase, it reminds you to give more honest estimates when dealing with project planning. Always add 30% to 50% more time to any of your estimates. These types of comments are used extensively in OSS packages:

From the otherwise excellent Spacy library:

```
# TODO: add more cases from non-English WP's

# TODO: There may be more architectures we can white list.

# TODO: This function's API/behaviour is an unholy mess...
# 0 means missing, but we don't bother offsetting the index.

# Source: https://github.com/stopwords-iso/stopwords-sl
# TODO: probably needs to be tidied up - the list seems to have month names in
# it, which shouldn't be considered stop words.
```

From the famous Scikit-learn library:

```
# TODO: add support for CSR input

# TODO: once the `k_means` function works with sparse input we
# should refactor the following init to use it instead.

# TODO: Remove in 0.24

# TODO: remove hard coded numerical results when possible
```

From the rising Tensorflow library:

```
# TODO(b/131417512): Add equal comparison to types.Artifact class so we
# can use asserters.

# TODO(b/139281215): this will be renamed to 'statistics' in the future.

# TODO(b/141874796): Implement OSS version of `multi_process_lib`.
```

Plugging that Amp

Some of us know the impact of an "@channel" on a slack messaging system. Everyone in that messaging channel will get a notification, and that notification cannot be muted. This one way to amplify a message's priority and importance. A similar thing happens on the comments that scream at the user to leave this piece of code alone. Here is an example where we see how the author wanted the reader to avoid messing with the order of arguments to keep coherent size information between C and Numpy data structures. Note the "XXX," the "careful," and the "it is important" keywords. This emphasis the needs of this function and amplifies the intent of the original developer:

```
""" XXX: Careful to not change the order of the arguments. It is important to have i\
s_leaf and max_width consecutive as it permits to avoid padding by
the compiler and keep the size coherent for both C and numpy data structures."""
```

Just make sure that the distortion level on that Amp is low, and we can still hear the lyrics.

Docstrings

Python docstrings for public APIs of your application and your libraries are great. They are a standard feature of Python and integrate well with IDEs that will help you maintain the docs strings input and output variables. The linter will notify you if you are missing an argument in the docstring and if it is spelled correctly. The IDEs integration has limits, though. Docstrings suffer from the same level of abuse as other comments, so use with care and avoid things that can mess with the meaning of the docstrings.

Useless Comments

This is the place where I tell you what NOT to do. These handle cases where we are hiding bad code, covering up for lack of decisive action, and curbing that inner voice to become a productive monologue.
Most comments fall into this category. Usually, they are crutches or excuses for poor code or justifications for insufficient decisions, amounting to little more than the programmer talking to himself.

Enigmas and Charades

When writing comments for corner cases and special situations, it is easy to write a blurb of broken English sentences to describe the half-baked thought we want to express to the reader. When writing a comment, use full sentences that give enough information about the scenario and the expected corner case we are dealing with.
Here is an example where we have an enigmatic message due to either a last-minute comment before running to another meeting or the author got a "high-priority" slack message midway through his thought process:

```
def load_features_from_file():
   try:
      df = pandas.read_csv(self.features_filepath)
   except:
      # No features_filepath means remote not initialized
      raise Exception()
```

What does that mean? features_filepath is supposed to be loaded ahead of time? What is remote? Is it some sort of remote storage? Or is the author putting a TODO here to come back and implement the initialization of the remote storage? Unfortunately, we will have to dig into the source of that feature_filepath property to understand who sets it and when. That's pretty annoying because there is a communication break down. This complexity is forcing the reader to look outside the scope of this function to understand what is going on.

Split-brain backups

The following example shows a simple model publishing function that has a redundant comment upfront. It restates exactly what the function does, but the function is not that complex in the first place. The danger of such a situation is that the comment will probably not evolve at the same pace as the code inside this function. If the model uploading process changes and the comments do not get updated, then we have a split-brain situation. The reader will assume the comments are correct, but they are not. So they will get burnt and then will read the code itself to understand the process in this function.

```
def upload_new_model_to_repository(self,
                                   input_model,
                                   repository,
                                   timeout):
    # creates a new model location for the model
    # if the repository is unable to create a new model location
    # raise an exception
    # Otherwise attempt to upload, and if that does not fail raise another
    # exception
    try:
        repository.create_new_model_location(input_model.uuid)
        if repository.is_model_location_set(input_model.uuid):
            self.upload_model_to_repository(input_model, repository)
        else:
            raise Exception('Repository not available')
    except:
        raise Exception('Unable to upload model to repository')
```

Fake news

Note that in the above listing, we had redundancy but also a clear sense of potentially fake news. Did you notice how the comment said that if model upload succeeds, then raise an exception? Obviously, the code itself is not doing that. My point is that long, complicated comments are a symptom of long functions with complex logic. As we covered in the functions chapter, the function should be small, which would lead to small comments. Smaller comments tend to disappear when the code explains itself. Short functions usually explain themselves.

Mandated Comments / Executive Comments Policy

Imagine a git commit system that checks the length and contents of your git message before allowing you to push your code to a remote repository. “Oh no! You are missing your user id, address, and social security number in your commit message”. Rules and policies like this should be kept to a minimum. The more policies get put in place, the faster the programmers will automate away their pain. This is especially true for comments. Once policy says that every function needs a docstring and every variable needs a comment, then we end up with codebases that are littered with scar tissue ala Frankenstein.

Journal Comments / Curbing that inner monologue

```
"""
   Changes (from 29-Feb-2040)
   * --------------------------
   * 24-Oct-2040 : Fix Autograd parameters
   * 07-Dec-2040 : Add feature engineering for upgrade to support new model
   * 17-Apr-2041 : Fix hyper-parameter pro-search parameters
   * 02-Dec-2001 : Fix bug in model selection module
   ......
"""
```

Inner monologue is better contained in git commit messages. The power of the git blame functionality is more than enough to track the evolution of a module. Try to avoid adding entries to a log of activities that were done to a source file:

Other famous useless comments

This section would not be complete without mentioning the "I can't hear you" comment, and the famous "functional but commented code" comment. The "I can't hear you!" is where there is so much noise commenting around the actual code that we cannot read it. The "functional but commented code" is worse, how many times do we find or write lines that validate something, just to comment it out to make the program pass. This is especially true for machine learning and data science, where "Experimental code paths" are left in place but commented out. If an experiment did not succeed, and we are moving to the next version, it is best to DELETE the code in question. We have the full history in the git version control system. I mean, this is given that you gave source version control tools a chance. Give them a chance.

Formatting Goals

Use First Impressions To Communicate

Imagine you just got a message on your internal messaging system (slack). It is a link to a Bitbucket/Github repo. You open the link and get faced by a wall of code that you are supposed to help maintain. You scroll down slowly, skimming the file you need to edit, and you are pleased by

the organization. This code has something special about it. Very symmetrical, neat, and consistent in style. Every class, method, and variable is well placed and structured. You keep scrolling and realize someone really cared about this piece of code. You are energized to join the same level of professionalism that is exposed in this module.

Isn't this an ideal situation? The code is energizing team members with sheer style. The code might have bugs and all, but there is something about this module that brings a sense of orderly piece about it.

It is very different from your last project. That codebase was probably written by a cross between hyper-caffeinated software consultants and a school of toddlers. Fingerpaint, dirty diapers, and foot-hurting Lego pieces everywhere.

This section is about how to care about your ML code and how to keep it well-formatted. The principal rule is consistency. Everyone on your team should agree about a set of standard formatting rules, and everyone should follow those roles for that project. I report a couple of directions for formatting your code, but they are menu items rather than hard rules.

First, let's talk about explainability. One way style is important is how it is related to model explainability. This aspect is an important part of improving models based on generated feedback. White box models such as linear models, logistic regressions, decision trees, and even random forest are easier to understand and debug than complicated black box models such as bi-directional LSTMs or Transformers. Only the model knows what is inside the model.

Second, think complex feature transformation pipeline. The pipeline starts simple, you get it to work, and reorganize it with some team level style guide. The readability improves dramatically, and a new teammate also wants to contribute to this feature transform pipeline. He finds a well-formatted source file, and so they might change everything about the business logic of the pipeline. Still, the essence of the formatting rules will probably survive the many iterations of this pipeline. Good looking code breeds more good looking code. Highly expressive code breeds more highly expressive code.

Python File Size and Notebook Size

Similar to everything in software, we want to "make smaller things" [3]. This is both for Python modules and Jupyter Notebooks. Here are some heuristics:

- Python modules file length:
  * Majority of modules should be below 200 lines
  * Some might be around 500 lines.
  * Some other exceptions can go to 1000 lines if needed.
- Jupyter Notebooks file length:
  * For notebooks, let's use mouse page scrolls: How many times do you need to swipe/scroll your mouse to reach the end?

* Most notebook should be 5-page-scrolls or below 50 cells to reach the end of a notebook.
* Some might be around 100 cells if we have cells that are non-code and only informational.
* Some notebooks are just monsters and can live to be 200 cells. Please don't.

For Python modules, the reader's process is:

- If this is a "main" module, then scroll to the bottom and read the module bottom up.
- If this is a "non-main" module, then we find the functions and classes top down.

For Jupyter notebooks, when the code reaches more than 20 to 50 cells, it is time to extract some of the functionality to a proper Python module. This is usually done with an nbconverter, but merging cells and copy-pasting in an importable module is also an ok option.

PEP-8 When You Can

Here is the star of the show: https://www.python.org/dev/peps/pep-0008/

For the Python community, pep-8 is a classic. This document "[...] gives coding conventions for the Python code comprising the standard library in the main Python distribution. " [2]. It handles a set of formatting goals. Study it and use it.

It can be hard to follow the formatting rules from memory, but today's most Python IDEs have pep-8 as their default formatting strategy. All it takes is putting your python code in the module, opening them in a project otherwise in your favorite IDE (in this example PyCharm), and pressing some keys combo:

This little shortcut will save you hours of discussion with other engineers. "Should we just use the default of PyCharm for this project?" asks one engineer, "Yeah, I am down to using the reformat code function. If we don't like something, we can change it with an editorconfig file". Yes, the guideline is to go with the IDE default and, if needed, use the editorconfig configuration that allows you to add a codestyle configuration in your IDE to unify the team's coding style for a specific project. Here is what that file looks like, and you can see that it is pretty self-explanatory:

```
{
 "schemeName": "Default",
 "version": "1.0",
 "codeStyle": {
   "all": {
     "formatter_off_tag": "@formatter:off",
     "formatter_on_tag": "@formatter:on",
     "formatter_tags_accept_regexp": false,
     "formatter_tags_enabled": false,
     "max_line_length": 120,
```

```
    "wrap_on_typing": false
  },
  "python": {
    "align_collections_and_comprehensions": true,
    "align_multiline_imports": true,
    "align_multiline_parameters": true,
    "align_multiline_parameters_in_calls": true,
    "blank_line_at_file_end": true,
    "blank_lines_after_imports": 1,
    "blank_lines_after_local_imports": 0,
    "blank_lines_around_class": 1,
    "blank_lines_around_method": 1,
    "blank_lines_around_top_level_classes_functions": 2,
    "blank_lines_before_first_method": 0,
    "continuation_indent_size": 8,
    "dict_alignment": 0,
    "dict_new_line_after_left_brace": false,
    "dict_new_line_before_right_brace": false,
    "dict_wrapping": 1,
    "from_import_new_line_after_left_parenthesis": false,
    "from_import_new_line_before_right_parenthesis": false,
    "from_import_parentheses_force_if_multiline": false,
    "from_import_trailing_comma_if_multiline": false,
    "from_import_wrapping": 1,
    "hang_closing_brackets": false,
    "indent_size": 4,
    "indent_style": "space",
    "keep_blank_lines_in_code": 1,
    "keep_blank_lines_in_declarations": 1,
    "keep_indents_on_empty_lines": false,
    "keep_line_breaks": true,
    "new_line_after_colon": false,
    "new_line_after_colon_multi_clause": true,
    "optimize_imports_always_split_from_imports": false,
    "optimize_imports_case_insensitive_order": false,
    "optimize_imports_join_from_imports_with_same_source": false,
    "optimize_imports_sort_by_type_first": true,
    "optimize_imports_sort_imports": true,
    "optimize_imports_sort_names_in_from_imports": false,
    "smart_tabs": false,
    "space_after_comma": true,
    "space_after_number_sign": true,
    "space_after_py_colon": true,
```

```
54      "space_before_backslash": true,
55      "space_before_comma": false,
56      "space_before_for_semicolon": false,
57      "space_before_lbracket": false,
58      "space_before_method_call_parentheses": false,
59      "space_before_method_parentheses": false,
60      "space_before_number_sign": true,
61      "space_before_py_colon": false,
62      "space_within_empty_method_call_parentheses": false,
63      "space_within_empty_method_parentheses": false,
64      "spaces_around_additive_operators": true,
65      "spaces_around_assignment_operators": true,
66      "spaces_around_bitwise_operators": true,
67      "spaces_around_eq_in_keyword_argument": false,
68      "spaces_around_eq_in_named_parameter": false,
69      "spaces_around_equality_operators": true,
70      "spaces_around_multiplicative_operators": true,
71      "spaces_around_power_operator": true,
72      "spaces_around_relational_operators": true,
73      "spaces_around_shift_operators": true,
74      "spaces_within_braces": false,
75      "spaces_within_brackets": false,
76      "spaces_within_method_call_parentheses": false,
77      "spaces_within_method_parentheses": false,
78      "tab_width": 4,
79      "use_continuation_indent_for_arguments": false,
80      "use_continuation_indent_for_collection_and_comprehensions": false,
81      "wrap_long_lines": false
82    }
```

Minimize Conceptual Distances

Y-Axis

Openness

In all popular “English-based” programming languages, reading code follows a familiar pattern, left to right, top to bottom. Other options, like the Qalb functional language might need a different set of rules and guidelines.

Python is a good representative of a common English-based programming language. In this context, we write clauses and expressions in terms of lines. Each line represents a single activity, and each

group of lines represents a cluster of activities that combine into a complete idea. To communicate effectively, it is best to use blank lines to separate ideas from each other.

Let's see the impact of removing lines on the expressivity levels of a set of functions. Here is a code snippet with well-spaced components:

```
def append_age_categories(self, input_df, use_cut_points, use_quantiles):
    df = input_df.copy()
    if use_cut_points and not use_quantiles:
        df = self.bucketize_age(df)
    elif use_quantiles and not use_cut_points:
        df = self.quantize_age(df)
    return df

def quantize_age(self, input_df):
    df = input_df.copy()
    df['Age_categories'] = pd.qcut(df.Age, q=4, labels=False)
    return df

def bucketize_age(self, input_df):
    df = input_df.copy()
    cut_points = [-1, 0, 5, 12, 18, 35, 60, 100]
    label_names = ['Missing', 'Infant', 'Child',
                   'Teenager', 'Young Adult', 'Adult',
                   'Senior']
    df["Age_categories"] = pd.cut(df["Age"], cut_points, labels=label_names)
    return df
```

Let's remove the "negative space" around the code components:

```
def append_age_categories(self,input_df,use_cut_points,use_quantiles):
        df=input_df.copy()
        if use_cut_points and not use_quantiles:df=self.bucketize_age(df)
        elif use_quantiles and not use_cut_points:df=self.quantize_age(df)
        return df
def quantize_age(self,input_df):
    df=input_df.copy(); df['Age_categories']=pd.qcut(df.Age,q=4,labels=False)
    return df
def bucketize_age(self,input_df):
    df=input_df.copy(); cut_points=[-1, 0, 5, 12, 18, 35, 60, 100]
    label_names=['Missing','Infant','Child','Teenager','Young Adult','Adult','Senior']
    df["Age_categories"]=pd.cut(df["Age"],cut_points,labels=label_names)
    return df
```

The impact of removing the spaces and blank lines turns readable code into a cryptographic machine-only-readable mess that is unsuited to support human life on its planet. Remember the "Squint test" we covered in previous chapters. What happens when you zoom out, do you still get a sense of the structure of the code and its intent? The core concept that the CC book calls this is "vertical openness", and we want to promote that in our code.

Density

Compared to the openness idea, we want to group things that are associated with each other. We want to make the lines that cooperate to be packed densely. See the impact of adding a bunch of unnecessary comments to an otherwise simple **init** function:

```
def __init__(self, use_cut_points,
             use_quantiles,
             extract_titles,
             generate_dummies,
             dummies_columns):
    # Flag to enable bucketizing age
    self.use_cut_points = use_cut_points

    # Flag to enable quantizing age
    self.use_quantiles = use_quantiles

    # Flag to enable extracting age
    self.extract_titles = extract_titles

    # Flag to enable generating dummies
    self.generate_dummies = generate_dummies

    # List to specify the dummies columns
    self.dummies_columns = dummies_columns
```

The CC book uses the term "eye-full". This means you can get the gist of the code without having to move your head, eyes, or mouse-fingers. See if you can get an "eye-full" from the following example:

```
def quantize_age(self, input_df):
    df = input_df.copy()
    df['Age_categories'] = pd.qcut(df.Age, q=4, labels=False)
    return df
```

Distance

How much clicking and scrolling around are you doing during your coding? The lesser, the better. Do you have to navigate up and down your Jupyter notebook or Python Scripts to find where a

variable is defined? Do you end up with four editor tabs just to be able to find where a function you are using is implemented? All this map building is energy expensive, and we would rather use that energy to produce more quality code than to retrace the inner workings of the system we are working on.

We want to avoid having to travel far away from your current edited line. Variables, functions, and other components that are related to each other should be kept next to each other in the same file. The distance between components should be a proxy to their cooperation level. The more cooperation, the closer they are.

A variable should be as local as possible. As we covered in the function chapter, we want our functions to be small and to do a single thing. The variables go at the top part of the function and will be close by default to the business logic in the middle of the function. Control variables used in for loops should be declared alongside the iterator that will populate them.

```
def append_one_hot_encoded_columns(self, input_df, use_cut_points, use_quantiles,
                                   extract_titles, generate_dummies, dummies_columns):
    df = input_df.copy()
    if generate_dummies and dummies_columns:
        for column_name in dummies_columns:
            if column_name == 'Age_categories' and not use_cut_points and not use_qua\
ntiles:
                continue
            if column_name == 'Age_categories' and use_cut_points and use_quantiles:
                continue
            if column_name == 'Title' and not extract_titles:
                continue
            else:
                dummies = pd.get_dummies(df[column_name], prefix=column_name)
                df = pd.concat([df, dummies], axis=1)
    return df
```

Sometimes functions get longer than the usual small functions we advocate for. In this case, we are tempted to declare variables close to step 3 in a 4 step process. Your first reaction should be to break down this long function into smaller ones. If that is not possible at the moment, it is ok to declare variables closer to the business logic. This will help when you have time to refactor into smaller functions. Here is an example with a multi-step model selection function that has three functions hidden inside it. The variables are declared as needed near the business logic that uses them:

```
def model_selection_v1(models):
    step_1_outputs = []
    for model in models:
        step_1_outputs.append(step_1_function(model))

    step_2_outputs = []
    step_2_config = 0.15
    for model in models:
        step_2_outputs.append(step_2_function(model,
                                              step_1_outputs,
                                              step_2_config))

    step_3_outputs = []
    step_3_threshold = 0.75
    for model in models:
        step_3_outputs.append(step_3_function(model,
                                              step_1_outputs,
                                              step_2_outputs,
                                              step_3_threshold))

    return step_3_outputs
```

Classes are a special case. We usually group all the instance variables up in the **init** function, where they are usually initialized. These variables are used by a multitude of functions inside the class. One warning is with variables that are created and linked to "self" in the middle of other functions. When this happens, IDEs auto-complete usually gets confused about the variables that should be auto-completed for an instance of your class. It is also hard for the users of your class to realize when and how variables are getting linked to their instances at runtime. This extra flexibility of python should be used with care. It is nice to have the option to set an instance variable at any time during the life of an object. However, it obscures the creation and initialization process of those ad hoc instance variables. Here is an example where an instance variable is added outside the **init** function:

```
class SimpleFeatureTransformer:

    def __init__(self, feature_1, feature_2):
        self.feature_1 = feature_1
        self.feature_2 = feature_2

    def get_transformed_data(self, input_df):
        df = input_df.copy()
        append_estimated_credit_score(df, self.feature_1)
        append_estimated_net_spending(df, self.feature_2)
```

```
11          self.random_mid_code_config = True
12          return df
```

Ordering

More cooperation means close distances between functions. In python, applications, we usually have the classic 'if **name** == "**main**":' block that is used to trigger the run of an app or script. However, we don't have to keep the "def main():" function at the bottom of the file as well. When someone opens your application, they should be able to scroll top to bottom and see an increasing level of detail in your functions. The caller of a function should be above the callee. This callee also calls other functions that will be placed wisely below itself. This is similar to what happens in an academic paper. We start with the more generic functionality, which is abstract, then we move on to the more detailed concepts, methods, and results in the details of the paper.
Here is an example where the importance of order can be visualized in code. The append_age_-categories calls two functions and the reader can find them right below the caller:

```
def append_age_categories(self, input_df, use_cut_points, use_quantiles):
    df = input_df.copy()
    if use_cut_points and not use_quantiles:
        df = self.bucketize_age(df)
    elif use_quantiles and not use_cut_points:
        df = self.quantize_age(df)
    return df

def quantize_age(self, input_df):
    df = input_df.copy()
    df['Age_categories'] = pd.qcut(df.Age, q=4, labels=False)
    return df

def bucketize_age(self, input_df):
    df = input_df.copy()
    cut_points = [-1, 0, 5, 12, 18, 35, 60, 100]
    label_names = ['Missing', 'Infant', 'Child',
                   'Teenager', 'Young Adult', 'Adult',
                   'Senior']
    df["Age_categories"] = pd.cut(df["Age"], cut_points, labels=label_names)
    return df
```

X-Axis

Horizontal Alignment

Ideally, I would like to be able to review pull requests code on my phone. The tiny screen is too crammed sometimes, and I get evil "soft-wrap" text to read. Soft-wraps should be reserved for text

files and documentation when we need to write paragraphs that nature might take more than one line. What soft-wraps do to your code is that it gives you the impression that you are not breaking the rules of 80 to 120 characters per line. Yes, our screens these days have impressive resolutions. Also, let's not forget that you can span code across two (or three?) screens. This makes for epic horizontal treks when debugging a lambda function inside a lambda function inside a join inside a group by.

Please don't make your lines too long. There is a good reason why the two-column academic paper format is so popular. It makes reading super easy, without moving my head left to right and almost without moving my eyes. Next time you read a paper that uses the one-column format, see how much slower you are engaging with the ideas in the paper.

The same happens in Jupyter notebooks, if you see the little scroll bar at the bottom of the cell, then that is a sign that you should make your lines shorter. Also, if you stretch the notebook horizontally with some clever HTML configuration cells, then remember that it is to accommodate the data frames that are displayed in the output cells. Jupyter developers, in a way, are suggesting a reasonable width of the cells to help with code readability without hindering displaying dataframes and plots that are larger than 80 characters wide.

The rule of thumb is a maximum of 80 characters per line in most cases, and never more than 120 characters. Ever.

Horizontal Openness and Density

Similar to the vertical openness and density concepts, the horizontal dimension also has something to tell about readability.

It sounds a bit silly, but white spaces are used to separate what is not related. The absence of whitespace glues characters and concepts together. Check the next example. We have a function where we have extra space in front of the parentheses that are otherwise very closely related to the function. Then we have some shenanigans with equal signs. When it is closer to the destination, it looks as if it has a different meaning than the assignment, as if it was part of the name of the function. When it is too close to the value, it looks like we have a variable that starts with the equality sign. Next is the function alignments. I saw this a while ago when exploring some Fortran code. We don't need that level of order anymore. Furthermore, check out the two-headed monsters where we have two statements separated by a semi-colon. We want high openness and high density, but this is too dense to have any transparency.

```
def run_rl_white_space_explorer (self,
                                 initial_spatial_location,
                                 max_steps,
                                 target_reward):
    current_location=      initial_spatial_location
    current_step =0; current_reward= 0
    while is_training_done(current_step,max_steps,target_reward):
        current_reward=    self.get_reward ( current_location)
        current_location = self.get_next_location(current_reward )
    self.total_reward =    current_reward
```

Hopefully, with our IDEs, we can reformat this faster than a high-frequency system can place an order in a dark pool.

```
def run_rl_white_space_explorer(self,
                                initial_spatial_location,
                                max_steps,
                                target_reward):
    current_location = initial_spatial_location
    current_step = 0
    current_reward = 0
    while is_training_done(current_step, max_steps, target_reward):
        current_reward = self.get_reward(current_location)
        current_location = self.get_next_location(current_reward)
    self.total_reward = current_reward
```

One regrettable and unfortunate element we lost with auto-formatted is custom formatting. Now that we can automate formatting, the IDEs treat everything the same. However, one exception was reported in the CC book, where we have to deal with equations. Equations? Did anyone say Equations? Yes, we do have a use for Equations in ML code. Lots.

Take a look at (a*bc* + f/g). This equation is much easier to grok where we group variables by operator precedence. Unfortunately, automated formatters do not know that this is an equation and it gets formatted like any other code statement to this:
(a * b * c + f / g).

Add a more complex equation with a picture of the actual math thing, its non-spaced version, and its custom-spaced version.

Dedentation and Indentation

Ah! The good old debate: Tabs vs. 2-Spaces vs. 4-Spaces. I personally don't have a preference as long as it is four spaces. What I think does not matter to your team. Choose one schema as a team, and everyone on the team should use that.

Indentation is pretty useful. It allows us to create hierarchical structures. The drawback of hierarchies is that they can be too deep. Again we touched upon this in the function structure chapter with the "Landscape test" that asks you to turn your code sideway and trace the peaks and valleys of your code. In a way, indentation contributes to the logical grouping of your algorithms. Remember that you can always extract a method with a single click of a button. Sorry Jupyter notebook users, the only way I know of is copying the code from a notebook to a python file, open it with an IDE, run the extracting and reformatting, and move your code back to the notebook. But, by the way, if we are refactoring this code, maybe it is stable enough to live in its own file that can be imported in your notebook. Just a suggestion here…

SQL queries are one beast that we have trouble with deciding if they should go to the leftmost part of the screen, or be formatted like the rest of the nearby code. Again, just decide on a single way as a team and get on with it. Don't forget that there is a handy textwrap.dedent function in the standard library. But also remember that if the SQL operates a separate level of abstraction than the rest of the code, then it should be extracted elsewhere.

```
# SQL query hugging the left of the editor
def get_data_from_db():
    query = """
SELECT feature_1 , feature_2
FROM products
WHERE feature_3 IN
(
    SELECT feature_4
    FROM orders
    INNER JOIN promotions
    ON orders.product_id = promotions.product_id
)
    """
    part_of_the_function()

not_part_of_the_function()
```

For some DBs the extra indentation might create some errors. This leads the devs to push the SQL queries to the left. This creates a visual problem for the scope of the function. Since we pushed the query to the left, it looks like the correctly indented function part_of_the_function is out of place, while the top-most scoped function not_part_of_the_function() looks like it is at the same level as the SQL query.

If the db we are using really has an issue with the indentation of the SQL query then we can indent the SQL text correctly and then use the dedent function to remove the extra indentation.

```
from textwrap import dedent

# SQL query with extra spaces.
# If the more readable query creates issues with the DB,
# then we dedent the query to remove the extra indentation.
def get_data_from_db():
   query = dedent("""
               SELECT feature_1 , feature_2
               FROM products
               WHERE feature_3 IN
               (
                   SELECT feature_4
                   FROM orders
                   INNER JOIN promotions
                   ON orders.product_id = promotions.product_id
               )
                   """)
```

Dictionaries also suffer from this formatting problem. Do we align all the values at the same indentation level? Or leave them to grow organically. The auto-formatter will usually push them to the left side to reduce the number of spaces to the equality sign. This is one rule I have to agree with because the keys of the dictionary grow over time, and if we don't update the indentation of the dictionary keys, then we end up with a squiggly line of values anyways. So left-justify away auto-formatter.

```
# stage 1: attempting to keep the values aligned
model_configuration = {
    "model_type":           "random_forest",
    "n_estimators":         "100",
    "criterion":            "gini",
    "max_depth":            "None",
    "min_samples_split":    "2",
    "min_samples_leaf":     "1",
    "max_features":         "auto",
    "max_leaf_nodes":       "None",
    "min_impurity_decrease":"0."
}

# stage 2: adding a long key, breaks the formatting
model_configuration = {
    "model_type":           "random_forest",
    "n_estimators":         "100",
    "criterion":            "gini",
```

```
    "max_depth":            "None",
    "min_samples_split":    "2",
    "min_samples_leaf":     "1",
    "min_weight_fraction_leaf": "0.", # This new key value pair broke the formatting
    "max_features":         "auto",
    "max_leaf_nodes":       "None",
    "min_impurity_decrease":"0."
}

# stage 3: giving up and using the default formatting rules without indentation.
model_configuration = {
   "model_type": "random_forest",
   "n_estimators": "100",
   "criterion": "gini",
   "max_depth": "None",
   "min_samples_split": "2",
   "min_samples_leaf": "1",
   "min_weight_fraction_leaf": "0.",
   "max_features": "auto",
   "max_leaf_nodes": "None",
   "min_impurity_decrease": "0."
}
```

One last thing about one-liners

They make the author reach the pinnacle of hubris driven cleverness. On the other hand, the reader will probably indent everything and add spaces everywhere to re-constitute the intent of the code. Please avoid complex one-liners but here are some classic one-liners:

Want to know many bytes a terabyte is?

```
import pprint;pprint.pprint(zip(('Byte', 'KByte', 'MByte', 'GByte', 'TByte'), (1 << \
10*i for i in xrange(5))))
```

What's the largest number that can be represented by 8 Byte?

```
print '\n'.join("%i Byte = %i Bit = largest number: %i" % (j, j*8, 256**j-1) for j i\
n (1 << i for i in xrange(8)))
```

Quicksort Python One-liner?

```
lambda L: [] if L==[] else qsort([x for x in L[1:] if x< L[0]]) + L[0:1] + qsort([x \
for x in L[1:] if x>=L[0]])
```

Sieve of Eratosthenes Python One-liner?

```
reduce( (lambda r,x: r-set(range(x**2,n,x)) if (x in r) else r), range(2,int(n**0.5)\
), set(range(2,n)))
```

Parallel Processing?

```
print list(multiprocessing.Pool(processes=4).map(math.exp,range(1,11)))
```

Madness during parsing? [4]

```
lambda s: map(lambda x: {x[0] : {'economic' : x[1], 'social' : x[2]}}, map(lambda x:\
 [x.split('=')[0]] + x.split('=')[1].split('%2C'), s.split('?')[1].split('&')[:-3] +\
 ['{}={}%2C{}'.format(*map(lambda x: x.split('=')[1], s.split('?')[1].split('&')[-3:\
]))]))
```

Conclusion

In his CC book, R. C. Martin said it best: “Code formatting is important. It is too important to ignore and it is too important to treat religiously”. There is no hard and fast way for “design” work. veryone on the team must agree on a common formatting method, automate it, and stick by it as long as the project is alive. Expressing the intent of the code in a clean, precise, and standardized manner will go far in simplifying the development of the team and any future maintenance.

References

[1] Clean Code: A Handbook of Agile Software Craftsmanship. Robert C. Martin. 2008. Prentice Hall PTR, Upper Saddle River, NJ, United States
[2] https://www.python.org/dev/peps/pep-0008/
[3] RailsConf 2014 - All the Little Things by Sandi Metz. https://www.youtube.com/watch?v=8bZh5LMaSmE
[4] https://rainier.io/crazy-python-one-liners.html

Chapter 5 - Clean Machine Learning Classes

I Know Classes in Python Why Are You Wasting My Time?

I was recently asked to explain, "Why should I care about object-oriented (OO) design? I use classes here and there, and they are like bags of functionality. Is there really more than that?" Well, as usual, it depends. Here is a snack to get you going: **The main goal of object-oriented design is managing dependencies**.

Let's assume that every time we draw a line between two boxes that represent some functionality, we are describing a dependency. Below is one example where a `HashingFeaturizer` class depends on a `DataLoader` class. As you probably guessed, anytime the `DataLoader` changes, breaks, or burns in multiple columns of flaming hell, the `HashingFeaturizer` will be impacted and will probably need to be changed. Easy enough.

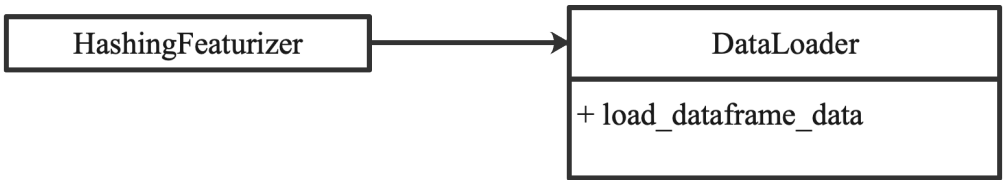


Figure 1: Simple Dependency

But things get interesting when we have transitive dependencies. In this case, a change in the `APIAdapter` would splash its modifications all the way to the `HashingFeaturizer`. Reducing this shockwave radius would be nice.

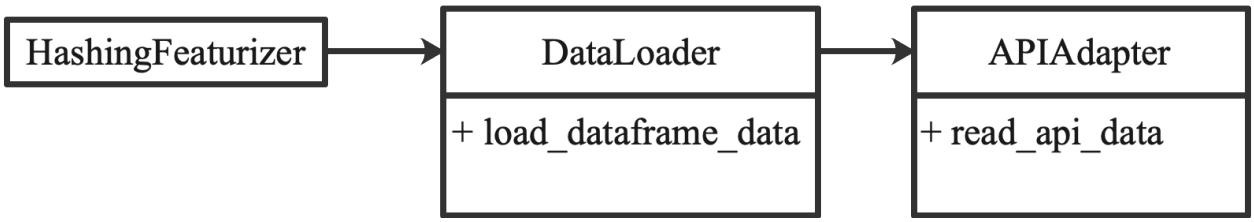


Figure 2: Transitive Dependency

However that is not the end of the story. Take a look below at what happens when a single class gets to be expanded with more functions.

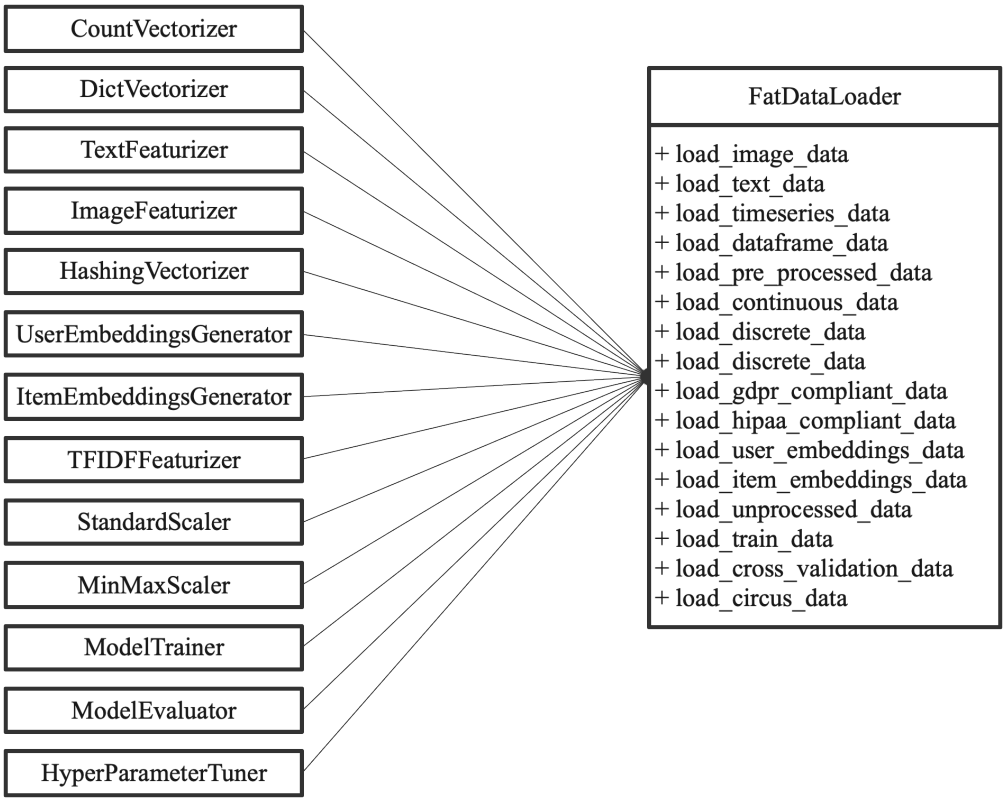


Figure 3: Fat Class in a Hurricane

This puppy is called a "Fat Class." This is a classic example of the "**that's how things are done around here**" mentality. Which we are all guilty of sometimes.

I don't have to tell you how coupled, fragile, rigid, and immobile the development will be. Disaster will strike if anyone introduces a change to the `FataDataLoader` class. There is a potential for breaking all the other classes that depend on it. I surely would rather avoid that. Classes like the above example reduce the development speed of teams.

Things are different If you work on your own. You cajole your single file Jupyter notebook that only you can operate. If that's the case, then so be it. But sooner or later, you might be asked to promote that notebook to be part of a feature engineering pipeline. Or worse! You might be assigned a teammate. That second ML engineer will have to make changes to that `DataLoader` class. How nice would it be if they were able to extend your class without raising a dust storm so thick you wouldn't be able to find a screaming mule?

Sometimes we can't totally avoid the pain. Sometimes we can only decrease the pain in a principled and goal-oriented manner by carefully managing dependencies. That is one version of the promise of OO and the SOLID principles. They were defined to help us mere mortals navigate the multitude of design decisions that we need to write maintainable ML software.

Goals for ML Class Design

We don't really need principles, rules, constraints, and Waffle makers. Well, maybe waffle makers. But the next few points are abstract principles we can use to promote clean machine learning code. These techniques can sometimes seem too extreme. Yet, they are easier to understand than a multi-author model pipeline. Add to that 5 different team members that included 10 assumptions each. We got ourselves a source code that is ripe with fragility, viscosity, rigidity, and that is terrifying to change or maintain.

Let's see what these component-level goals are and how they can benefit the practice. Here are the basic goals:

Figure 4: Goals of ML Class Design

1. Loose Coupling

Two machine learning components are coupled when at least one of them uses the other. The less these ML components know about each other, the looser they are coupled. An ML component that is only loosely coupled to its environment can be more easily changed or replaced than a strongly coupled component.

2. High Cohesion

Cohesion is the degree to which machine learning elements of a whole belong together. Think peanut butter and jelly. Methods and fields in a single machine learning class, as well as classes of an ML component, should have high cohesion. High cohesion in ML classes and ML components results in simpler, more easily understandable machine learning code structure and design. This is similar to the Single Responsibility Principle but at the component level. Things that change for the same reasons and, at the same time, should be grouped together. ML components that do change for different reasons or at different times should be kept separate.

3. Change is Local

ML software systems have to commonly be maintained, extended, and changed for a long time. Keeping change local reduces involved costs and risks for the machine learning pipelines. Keeping

ML components changes local means that there are boundaries in the design, which changes do not cross.

4. It is Easy to Remove

We normally build machine learning software by adding, extending, or changing component features. However, removing ML elements is important so that the overall design of the machine learning pipeline can be kept as simple as possible. When an ML block gets too complicated, it has to be removed and replaced with one or more simpler
ML blocks.

5. Mind-Sized Components

Break your machine learning system down into Data/ML components into sizes you can grasp within your mind. The goal is to predict the consequences of changes easily (dependencies, control flow, etc.). Machine learning classes should be around 100 lines. Machine learning methods such as transform, fit, predict, and predict_proba methods should max out at 10~15 lines.

Remember that being able to test a machine learning component in ISOLATION is a sign of good architecture. Not being able to test a machine learning component in isolation is a sign of a crony architecture.

S.O.L.I.D Design Principles for ML Classes

Figure 5: SOLID Principles in fancy clothes

The SOLID software design principles were collected/refined by Robert Martin, “Uncle Bob,” in his excellent books. These principles aim to give practical guidance to software engineers. They act as strong guidelines that are as applicable to traditional software as to machine learning software.

Let’s see what happens when we adapt them to the machine learning domain, shall we?

Small Cohesive Classes: The Single Responsibility Principle

Figure 6: Yes, yes, a metaphore for doing too much…we get it!

A machine learning component should have one, and only one, reason to change.

To determine if a class/function has too many responsibilities, check the actors it is serving [1]. A warning sign is when more than one actor could ask for changes to this component. There is a high chance that the component contains more than a single responsibility. The owner needs to split this component into more fine-grained logical parts.

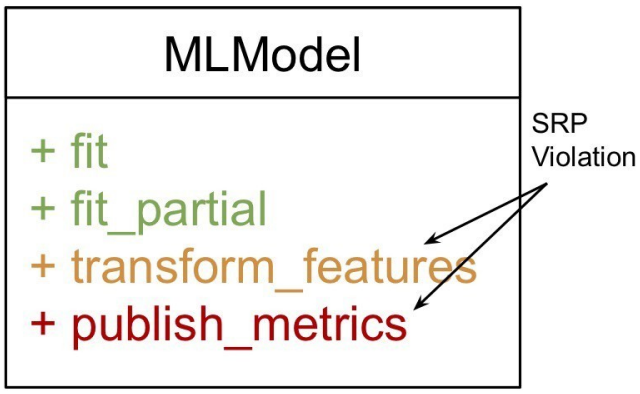


Figure 7: SRP Violation Detected!

In the above diagram, the `MLModel` class will have to change due to many separate reasons. Here we have the fitting method, the transformer of the input data, and metrics publishing. This class will have to react to changes in data processing, feature engineering, and model selection. The owner needs to partition this class into machine learning sub-components. How to deal with SRP violations is the concern of much of this book.

In this example, we have a class that does too much. The `MLModel` class responds to multiple actors/roles. Changes to the `fit` and `fit_partial` will come from the need to experiment with different model types. Changes to the `transform_features` will come from changes in the candidate feature transformations. And finally, `publish_metrics` will respond to changes in what/where metrics need to be published.

These are three different roles, a clear violation of the SRP principle.

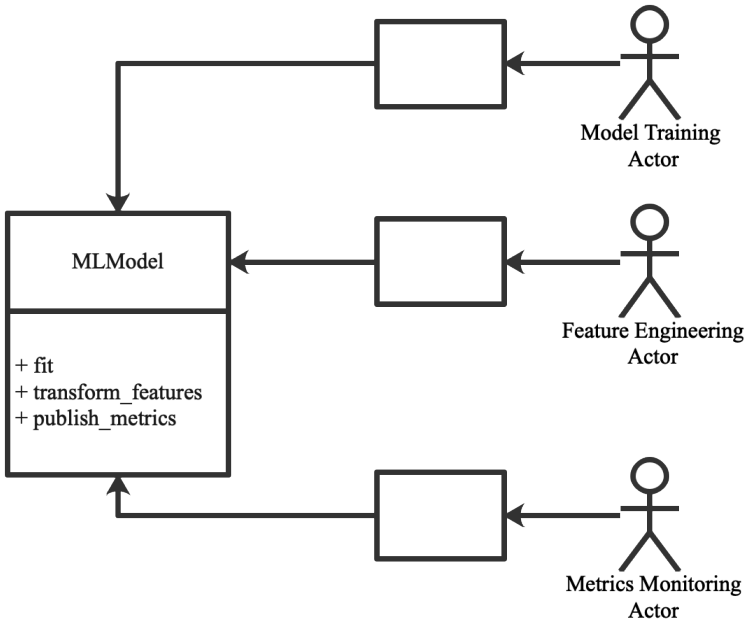


Figure 8: Three roles, one class, what could go wrong?

To support the `fit` functions, the `MLModel` will surely contain low-level code related to specific machine learning libraries, such as Scikit-learn, Tensorflow, Pytorch, and the like. To support the feature transform, the class will have to deal with extensive SQL and pandas/numpy operations. To support the publishing of the metrics, the class will have to include low-level details such as Prometheus, Mlflow, Tensorboard code, and the like. Now, as you can tell, these are not very cohesive responsibilities.

These disparate responsibilities mean that a change in the feature transformation code would likely impact the operations of the metrics publishing. What if changing the feature transformation pipeline to support hashing ends up breaking the metrics functions? What will the managers and product owners think of our software? They will see the fragility and think that the software engineers that built this product have lost control of their code, and a formal change freeze will set in. “Don’t touch anything I said!” grunts the demoralized product owner that promised a tight deadline to the clients. Nobody wants to be in those shoes.

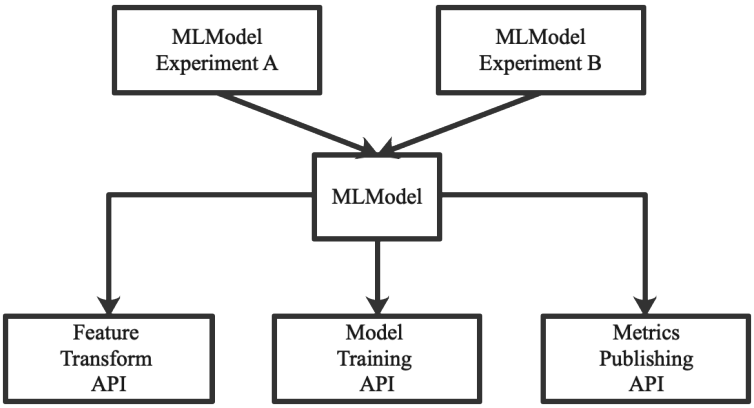


Figure 9: Class with three coupled APIs

By being in the same class, the training functions and the feature transformations are sure to be coupled. It starts pretty simply because both functions share too much knowledge with each other. For example, the feature engineering function skips the missing data handling because it knows that missing data points are ignored by the training code. The training code skips validating the labels proportions because it knows that the feature engineering function takes care of balancing the class labels. These two functions are growing roots inside each other.

In the following sections of this chapter, we will cover strategies to drive toward the single responsibility principle. Here are a couple of strategies in action.

Inversion of dependencies

We describe previously one of the great things about OO design is that it helps us manage dependencies. In this case, we can decouple the actors from the implementation. This can be done by splitting the class into an interface and an implementation.

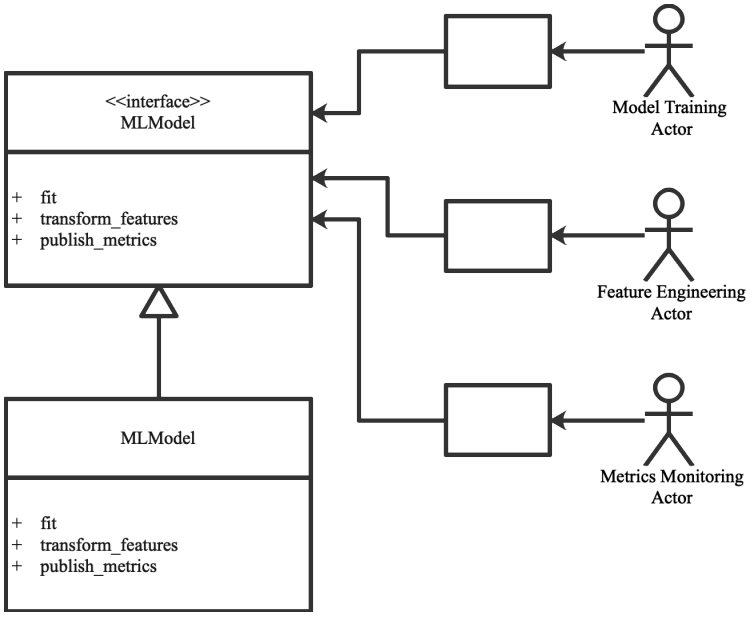


Figure 10: Basic Inversion dance

With this setup, the actors are decoupled from the implementation, but they are not decoupled from each other. First, all three actors depend on the same interface. Second, the implementations are still interconnected inside the same class. This means that any change to one of the functionalities will likely impact the other functions' behavior. Let's try something else.

Class extraction

The next attempt would be to split the class into three classes and place each function in its own class: The `fit` function in the `Trainer` class, the `transform_features` in the `FeatureTransformer` class, and `publish_metrics` in the `MetricsPublisher` class.

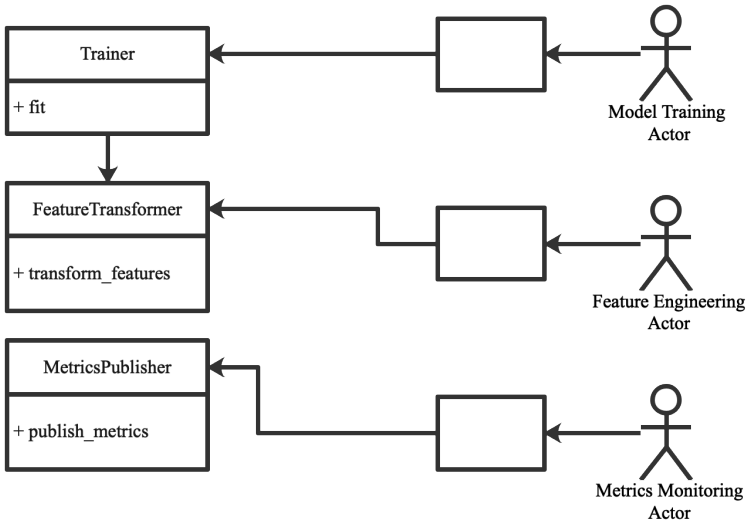


Figure 11: Splitting Classes by Role

This method makes the actors depend on three separate classes. If we need to introduce a change in one of the modules, the other modules will be untouched. However, this is not a zero-problem solution. First, there is a transitive dependency between the trainer and the feature transformer. The trainer needs to know what sort of data is coming out of the feature transformer. Second, the `MLModel` concept is now fragmented. This will force the developers to track down where the individual functions are and how they are related.

Facade

The next potential solution is the Facade pattern, which we will discuss in later chapters. The class extraction fragmented the MLModel concept into many parts. The facade pattern, in turn, groups the functions together into a facade class. This class then delegates the functionality to the individual implementations.

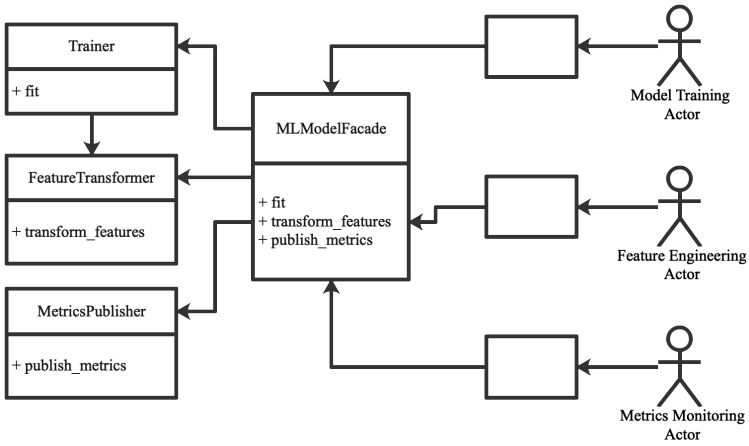


Figure 12: Building a Facade

This method enables users to find the functions faster because they are in the same class. However, the actors are interconnected again, since any change to the facade will impact all the actors.

Interface Segregation

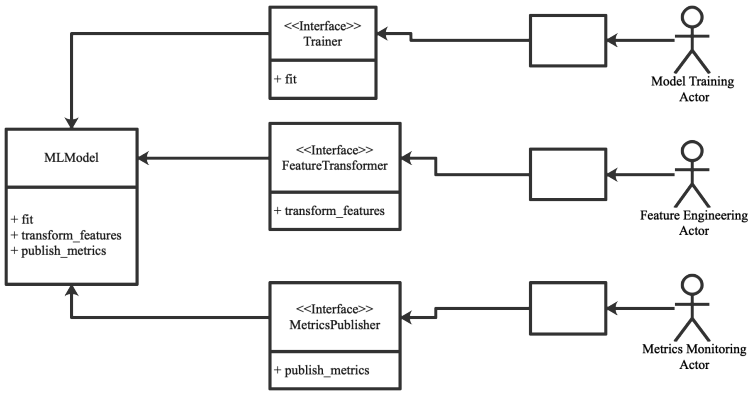


Figure 13: Segregating the Interface

To again decouple the actors, we can create three interfaces, and implement the interfaces in a single shared class. This has the advantage of keeping the actors fully decoupled because each of them only depends on their individual interfaces. However, this reintroduces the discoverability problem. The development team will have to track down the interfaces they require. Besides, the functions themselves are coupled since they live in the same class.

Perpetual Engineering Tradeoffs

I wish I had a magic wand that would solve this problem exactly. Unfortunately, there are forces that pull our software in multiple directions at the same time. The solutions in this book try to expose the various design options available so that you can make an informed decision about your current project at hand.

Maybe you are confident that your team is small enough. That the engineers on your team will have no issues navigating the codebase. That they will not have any issues finding the adequate functions they need. Or maybe you are exposing an API to a wide range of users, and you want to keep it as simple as possible. Or perhaps you don't mind enduring the high level of coupling because it is a sacrificial piece of software that will be thrown away anyway.

Dealing with these sorts of software engineering challenges is at the forefront of your task as an ML engineer/Data scientist/Data analyst.

Organizing for Change: The Open-Closed Principle

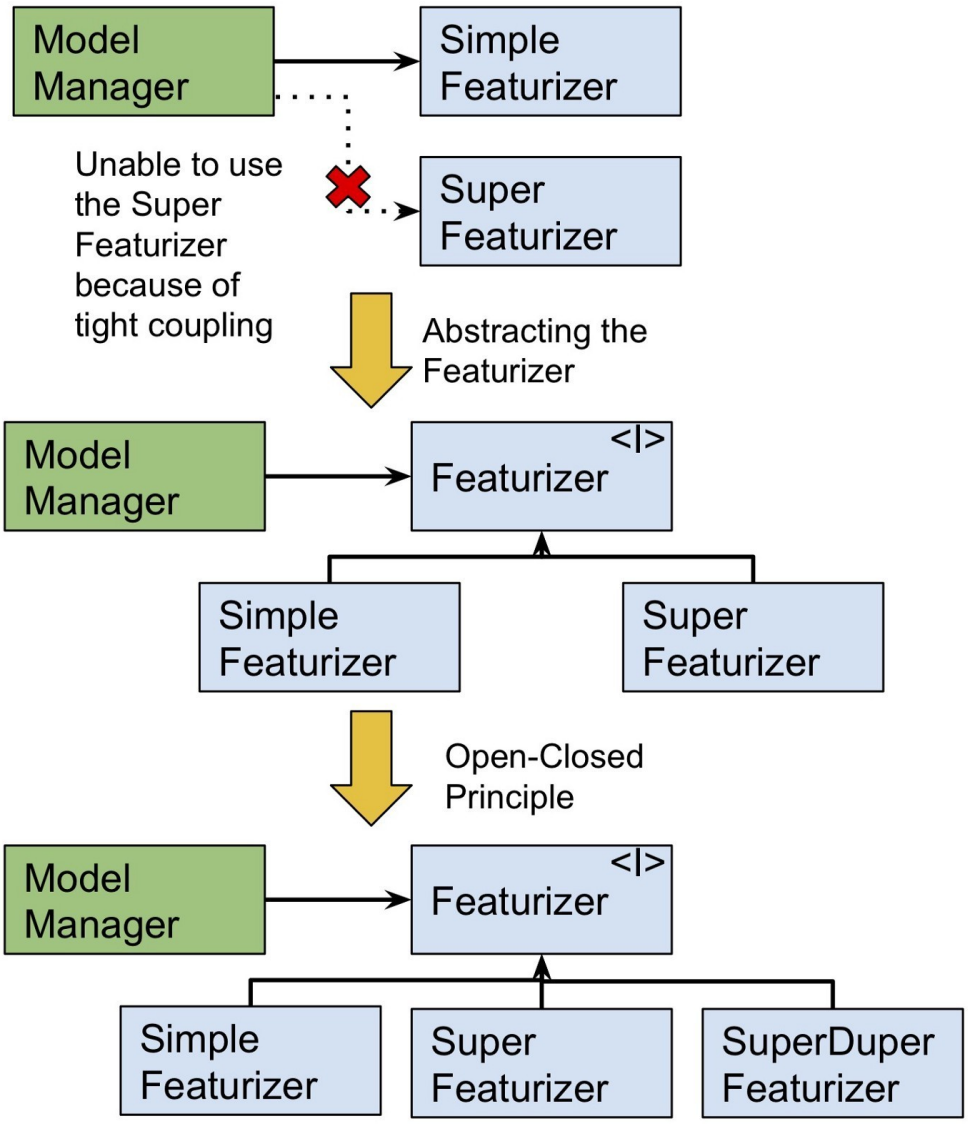


Figure 14: ML code evolution when using the open-closed principle

For example, in this diagram, the `ModelManager` started by using the `SimpleFeaturizer` class. However, in version 2, it needed to use the `SuperFeaturizer`. Due to the tight coupling, it was not possible to swap in the `SuperFeaturizer`. To support the new `SuperFeaturizer` would require changing the `ModelManager` class. This is a violation of the open-closed principle: Old code should not have to change to add new functionality.

The solution is to abstract away the `Featurizer` in an interface/abstract class. Then we make the dependency arrows point towards the abstract components. After the modification, the `SimpleFeaturizer`

and `SuperFeaturizer` are interchangeable. The `ModelManager` can use either one without knowing the internals of each strategy. Besides, the two `Featurizers` do not need to know about the `ModelManager`.

This way the developer can test them in isolation. In more dynamic languages like Python, there is no explicit need to build the interface. That is unless there is strict type checking using the "isinstanceof" method. The important part is to design the interacting classes to point toward abstraction. By making the `ModelManager` only aware of the interface, it can use either type of `Featurizers`.

This allows the developer to add a third `SuperDuperFeaturizer`. This is without any changes to the `ModelManager` or the other sub-classes of `Featurizer`. This is true as long as the `Featurizers` implement the shared interface. The `ModelManager` can ignore the implementation details of the concrete sub-classes. Hardwiring components usually make the model space exploration tedious, viscous, and slow. This helps the experimental velocity of model types, architectures, and other model customizations.

We want to cut the amount of old ML code that needs to change to add the new ML code. This is done by partitioning the full ML pipeline into components. Then we arrange those components into acyclic directed graphs. To achieve that, we make the dependencies arrows point towards abstract components. This allows us to isolate the components that we want to protect from change and/or are more stable. Information hiding and directional control protect the stable ML components. The fast-moving ML components also benefit from being free to change at will.

The Great OCP Fiction

Is it really possible to fully conform to this principle? Unfortunately, this is not how the real world works. Sometimes it is so hard to follow this principle that it is almost impossible to perfectly follow it.

First, there are the responsibilities of the `main` function. Usually, that is where all the configuration, creation, strategies, and factories are initialized. We will cover the role of `main` in the next chapter, but it is reasonable to assume that `main` will depend on many different parts of the application. Changing anything inside the application will surely require changes in `main`. In the example below, we see how all the dependencies are pointing away from the main partition. They are made to point towards the concrete implementations. Due to this multitude of dependencies that the `main` partition naturally contains, it hard to follow the OCP in the `main` partition.

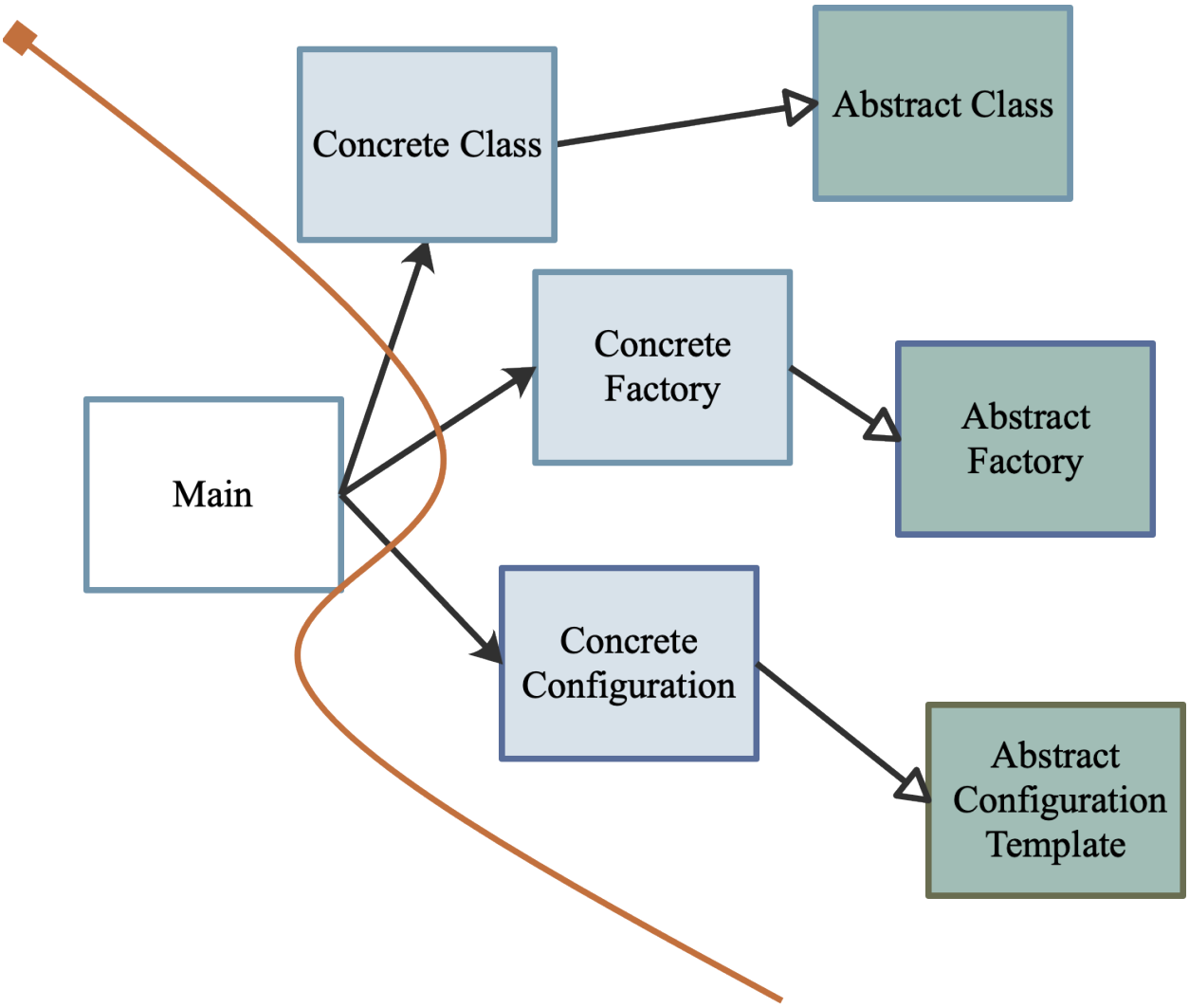


Figure 15: Main Partition and Boundaries

Second, the OCP works magic when we can perfectly predict the future. This principle does well when we "know", ahead of time, what change of functionality we want to protect our code from. But that's not as easy as it sounds. The OCP is very sensitive to the size of the system at hand. In large systems, it can be daunting to add all the abstraction layers necessary to achieve the OCP. However, for smaller packages, classes, and functions, there is still hope, even with imperfect information.

Unfortunately, building an example that would expose this problem would be hard to cram in a chapter. Still, we are going to look at a basic model that I hope will give you a sense of what can go right and what can go wrong with the OCP.

Here is the basic premise. Let's pretend that the following function is part of a larger machine learning pipeline.

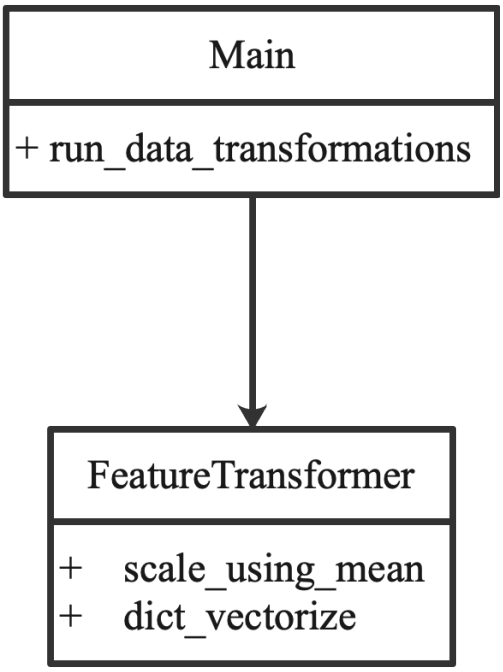


Figure 16: An OCP violation

```python
from typing import List
import pandas as pd
from pandas import DataFrame


class FeatureTransformer:
    def __init__(self, name: str):
        self.name = name

    def scale_using_mean(self, data: DataFrame):
        print(f"Running Mean Scaler using: {data.shape}")
        return "scaled data"

    def dict_vectorize(self, data: DataFrame):
        print(f"Running Dict Vectorizer using: {data.shape}")
        return "vectorized data"


def run_data_transformations(train_data: DataFrame, feature_transformers: List):
    res = []
    for transformer in feature_transformers:
        if transformer.name == 'mean_scaler':
            res.append(transformer.scale_using_mean(train_data))
```

```
        elif transformer.name == 'dict_vectorizer':
            res.append(transformer.dict_vectorize(train_data))
        else:
            pass

    return res


transformers = [
    FeatureTransformer('mean_transform'),
    FeatureTransformer('dict_vectorizer')
]

train_data = pd.DataFrame(
{'user_id': ['0', '1', '2', '3'],
'age': [20, 21, 25, 23],
'income_id': [1, 1, 3, 4]})

res = run_data_transformations(train_data, transformers)
```

The function `run_data_transformations` violates the open-closed principle. This function cannot be "closed" against new kinds of feature transformers. If we add a new feature transformer function, `median_imputer`, we have to modify the `run_data_transformations` function.

In essence, for every new transformer function, a new logical branch is added to the `run_data_transformations` function.

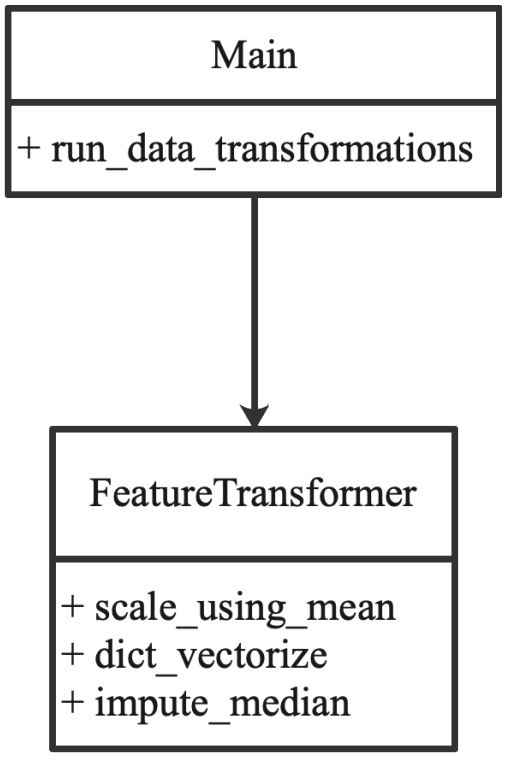


Figure 17: OCP and the broken windows mentality

```python
class FeatureTransformer:
    def __init__(self, name: str):
        self.name = name

    def scale_using_mean(self, data: DataFrame):
        print(f"Running Mean Scaler using: {data.shape}")
        return "scaled data"

    def dict_vectorize(self, data: DataFrame):
        print(f"Running Dict Vectorizer using: {data.shape}")
        return "vectorized data"

    def impute_median(self, data: DataFrame):
        print(f"Running Median Imputer using: {data.shape}")
        return "median imputed data"


def run_data_transformations(train_data: DataFrame, feature_transformers: List):
    res = []
    for transformer in feature_transformers:
        if transformer.name == 'mean_scaler':
```

```
            res.append(transformer.scale_using_mean(train_data))
        elif transformer.name == 'dict_vectorizer':
            res.append(transformer.dict_vectorize(train_data))
        elif transformer.name == 'median_imputer':
            res.append(transformer.impute_median(train_data))
        else:
            pass

    return res


transformers = [
    FeatureTransformer('mean_transform'),
    FeatureTransformer('dict_vectorizer'),
    FeatureTransformer('mean_imputer')
]
res = run_data_transformations(train_data, transformers)
```

This is, of course, quite a straightforward example

The problem becomes real when your ML pipeline grows and becomes more complex. You will surely see that those if statements will be peppered all over the code. This sort of branching tends to spread in multiple modules down the dependency chain. Each time a new transformer is added, multiple modules, classes, and functions will need to be modified. One apparent problem with that is that with each modification, a new bug could be introduced in the code. A more hidden problem is that all the branches are now coupled to each other. It will take a high level of vigilance to keep all of the instances of this branching in sync at all times. This is unless we like runtime exceptions pager calls at 2am on a Sunday, such as: "Runtime Exception: `median_transform` is not a valid feature transformer type." These are symptoms of a "Refused Bequest" [4], and that's no fun at all.

So how can we make this change less painful? How can we avoid violating OCP? The solution is our two good old friends: Abstraction and Inversion. The first issue we need to take care of is that the `FeatureTransformer` class breaks the SRP. It does too much. We need to break it apart. One way of doing it is so:

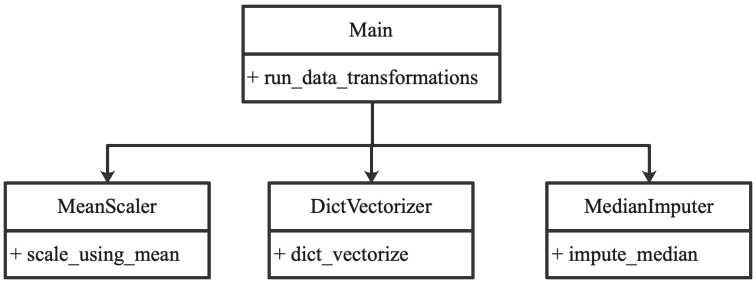


Figure 18: Getting close to satisfying the OCP

```
class MeanScaler:
    def __init__(self, name: str):
        self.name = name

    def scale_using_mean(self, data: DataFrame):
        print(f"Running Mean Scaler using: {data.shape}")
        return "scaled data"


class DictVectorizer:
    def __init__(self, name: str):
        self.name = name

    def dict_vectorize(self, data: DataFrame):
        print(f"Running Dict Vectorizer using: {data.shape}")
        return "vectorize data"


class MedianImputer:
    def __init__(self, name: str):
        self.name = name

    def median_impute(self, data: DataFrame):
        print(f"Running Median Imputer using: {data.shape}")
        return "median imputed data"


def run_data_transformations(
    train_data: DataFrame,
    feature_transformers: List):

    res = []
    for transformer in feature_transformers:
        if transformer.name == 'mean_scaler':
          res.append(transformer.scale_using_mean(train_data))
        elif transformer.name == 'dict_vectorizer':
            res.append(transformer.dict_vectorize(train_data))
        elif transformer.name == 'median_imputer':
            res.append(transformer.median_impute(train_data))
        else:
            pass

    return res
```

```
44
45
46 transformers = [
47     MeanScaler('mean_transform'),
48     DictVectorizer('dict_vectorizer'),
49     MedianImputer('mean_imputer')
50 ]
51
52 res = run_data_transformations(train_data, transformers)
```

We fixed the SRP problem somewhat, but we are still unable to get rid of that pesky branching section. The `run_data_transformations` still checks the name of each transformer before using it. This is where abstraction and inversion of dependencies can help. First, we make an abstraction out of thin air: The `FeatureTransformer` class has a `transform` abstract method. Then we make the individual transformers implement that function. Finally, we invert the dependency of the main loop toward the abstraction rather than the concretions. One way of doing that is as follows:

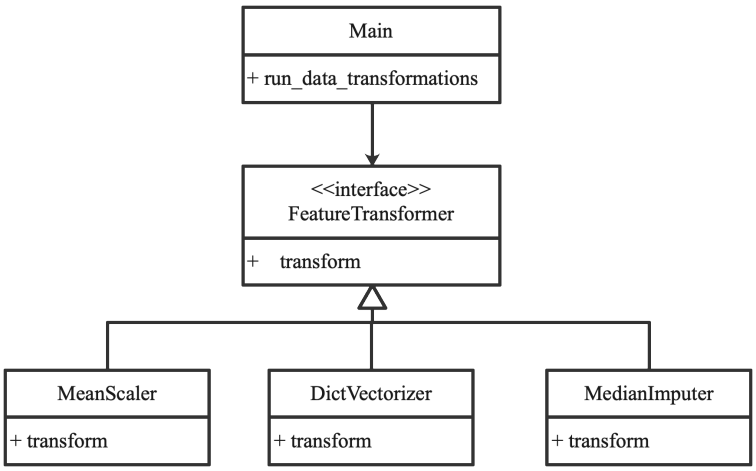


Figure 19: Inversion and Abstraction to get closer to the OCP

```
1  from abc import ABC, abstractmethod
2
3  class FeatureTransformer(ABC):
4      @abstractmethod
5      def transform(self, data: DataFrame):
6          pass
7
8
9  class MeanScaler(FeatureTransformer):
10     def transform(self, data: DataFrame):
11         print(f"Running Mean Scaler using: {data.shape}")
12         return "scaled data"
```

```
13
14
15  class DictVectorizer(FeatureTransformer):
16      def transform(self, data: DataFrame):
17          print(f"Running Dict Vectorizer using: {data.shape}")
18          return "vectorize data"
19
20
21  class MedianImputer:
22      def transform(self, data: DataFrame):
23          print(f"Running Median Imputer using: {data.shape}")
24          return "median imputed data"
25
26
27  def run_data_transformations(train_data: DataFrame, feature_transformers: List[Featu\
28  reTransformer]):
29      res = []
30      for feature_transformer in feature_transformers:
31          res.append(feature_transformer.transform(train_data))
32      return res
33
34
35  transformers = [
36      MeanScaler(),
37      DictVectorizer(),
38      MedianImputer()
39  ]
40
41  res = run_data_transformations(train_data, transformers)
```

`FeatureTransformer` now has an abstract method, `transform`. We have each `Transformer` extend the `FeatureTransformer` class and implement the abstract method, `transform`. The `run_data_transformations` iterates through the array of `Transformers` and just calls each `transform` method.

Now, if we add a new transformation, `run_data_transformations` doesn't need to change. All we need to do is add the new Transformer to the transformers array. Finally, `run_data_transformations` now conforms to the OCP principle. One added benefit is that all dependent classes and modules can also rely on this abstraction.

So now listen up folks, what would happen if we wanted to configure each transform with a threshold, default value, or some other constant that changes the behavior of each Transformer? Does following the OCP really protect our code from this sort of change? We could blame the initial requirements, and say that nobody told us about making the transformers configurable. If we have known the requirements ahead of time, we could have designed the transformers to be configurable.

If we have known, we could have conformed to the OCP. Yeah, but we didn't know, and that's reality of software development. How are we supposed to know something like that? Is it the case that the OCP really only works when we know what is going to happen to our requirements? One thing that is clearly related to machine learning experimentation is that machine learning components change depending on offline or online results. While we have some minor control over offline features, as soon as the customer interacts with our model, new types of data and behaviors are generated. Those will surely be added to the experimentation pipeline, creating havoc to existing abstractions. What good is Object-Oriented design if it requires knowing the future?

How to deal with the "crystal ball" problem if knowing the future is how we can conform to the OCP?

The first method relies on thinking extremely hard. First, we build a model of the domain, then we dress them up with abstractions, and we keep at it until we think up everything that could change. We try to anticipate the experimentation pathways that our ML application will need. Make sure to cover every potential change with a sea of abstractions. The drawback of this method is that it is tough to know ahead of time the strategies that will work for our current predictive task. Especially once we start serving the models to customers, we get non-intuitive results in predictive performance. So we try to modify the system that is already full of abstractions and indirections. That makes it hard for the ML engineers to know what part of the code to change.

The second approach is agile design. It is based on a pragmatic and reactive methodology. The idea is described much better in other writing, such as [3] . However, the core goal is to operate in very short iterations and expose our ML system to the customer as quickly as possible. Rushing the ML pipeline to a baseline state does two things. It provides a baseline method that can be used as an anchor point for future model improvements. But also, it exposes the ML engineering team to production system constraints early on. These deployment constraints, coupled with the baseline predictive performance, are crucial for evolving the system. This helps build the right abstractions to protect our source code from the discovered types of changes using the open-close principle.

In reality, projects live somewhere between these two extremes: agile and big design upfront. But two things remain constant. Changes in the ML requirements will happen in unexpected ways, and feedback is critical to evolving ML systems.

Maintaining Contracts: The Liskov Substitution Principle

Inheritance is a useful tool to write clean and maintainable code. However, the same abstraction that gives us the flexibility of design can also lead to rigidity. Subclasses are bound by a contract to their base class. They promise to be a complete substitute for the parent class. They must conform to the interface that binds them to their base class. They need to respond to every function call defined in that interface, accepting the same kinds of parameters, and returning the same types of outputs. The subclasses must not force the caller to check their type to treat them differently.

The subclasses can do more than the base class but never less. They can specialize and expand the functionality defined in the interface of the base class. Still, they must always implement the base contract. When classes diverge from this rule, the type hierarchy is put in question. It forces the developer to become paranoid and litter the code with extra checks such as `if isinstance(obj, 'MyType')`. Instead, the client should be confident that the objects it is handling abide by the interface defined in the superclass.

This is what the Liskov Substitution Principle [2] points to:

- Let `q(x)` be a property provable about objects `x` of type `T`. Then `q(y)` should be true for objects `y` of type `S` where `S` is a subtype of `T`.

Let's see that in action in a casual "imaginary" scenario.

Here is a prime example of the LSP violation idea. Clear immobility, fragility, and rigidity. The client is a large retailer. The team is in charge of the recommendation system that runs during the week leading up to the christmas shopping season: no big deal, just 10's of millions of dollars in sales on the line. No pressure.

Here we have three Metrics classes that specialize in calculating various metrics. Due to the clear LSP violation, the function `evaluate_models` uses these metrics classes but needs to check their types. Which is, as you guessed it, less than ideal. Python specifically does not enforce this sort of inheritance relationship. We will show we can enforce that in the next section.

Oh, and also you probably, or maybe you did, did not spot the bug we introduced. Initially, the application supported the Mean Average Precision metric, and all was well. But then the business needed to add the Mean reciprocal rank and the Normalized Discounted Cumulative Gain metrics because the customer thinks that those correlate better with their sales model.

The team ended up adding more classes that inherit from the Metric class. However, they did not want to modify the metrics classes because the calculations were terribly complex. They were so hyper business-specific and written two years ago by Fred that left the company 6 months ago. Everyone on the team was afraid of touching these metrics calculations, so the team just piled new metrics without even having the guts to change the metrics calculation signature. Because if it breaks, then it becomes the problem of the last person that touched that function. Nobody wants to be "that" person. The team hid and accepted the technical debt, since who knows if more metrics will be added or we can be happy with this as is.

Two weeks later, the team starts receiving notifications that a third-party provider. They regularly audit the models that the team produces. Their audit report is saying that the MRR output value does not match their internal benchmarks. They are basically saying that the ML team has been doctoring the offline metrics to win the contract. The product manager is getting agitated because the client, a pretty big one, is threatening to cancel their contract with the AI recommender product at hand. An ML squad is formed to dig into this metrics issue. Other projects are put on hold and then put on ice to make time for this metrics debugging expedition.

The team is split into analyzing the data sources, the MRR class itself, and the output database where the metrics are published. The assigned team readily dives in the MRR class to debug it entirely only to find that the metrics match the theoretical bounds. One of the team members runs an in-depth debug session on the prod system. This takes a good part of the week to get real data, from live traffic, that can be compared to the third party metrics auditor. This effort finally leads the team to see that the bug was in the branching of the client code.

```
class Metric():
    def calculate_metric(self, data: pandas.DataFrame):
        pass


class MAP5(Metric):
    def calculate_map5(self, data: pandas.DataFrame):
        print(
        f"Calculating Mean Average Precision @ 5")
        return "metric"


class MRR(Metric):
    def calculate_mrr(self, data: pandas.DataFrame):
        print(
        f"Calculating Mean Reciprocal Rank")
        return "metric"


class NDCG(Metric):
    def calculate_ndcg(self, data: pandas.DataFrame):
        print(
        f"Calculating Normalized Discounted Cumulative Gain")
        return "metric"


def evaluate_models(train_data: pandas.DataFrame,
                    metrics: List[Metric]):
    res = []
    for metric in metrics:
        if isinstance(metric, MAP5):
            metric.calculate_map5(train_data)
        elif isinstance(metric, NDCG):
            metric.calculate_ndcg(train_data)
        elif isinstance(metric, MAP5):
            metric.calculate_map5(train_data)
```

```
        res.append(metric.calculate_metric(train_data))
    return res


metrics = [
    NDCG(),
    MRR(),
    MAP5()
]

res = evaluate_models(train_data, metrics)
```

There are most likely more confounding factors, but the LSP violation played a significant role in the bug in question. Imagine if we put the following two fixes. First, we can make the metrics classes actually implement a unified method, `calculate_metric`.

Just by doing that, the branching disappears since we rely on polymorphism to dispatch the call to right Metrics class implementation. Second, we can enforce a runtime error when a new metric does not implement the contract defined in the base class, using the ABC and the abstractmethod construct . This is how it looks:

```
from typing import List
from abc import ABC, abstractmethod


class Metric(ABC):
    @abstractmethod
    def calculate_metric(self, data):
        pass


class MAP5(Metric):
    def calculate_metric(self, data):
        print(f"Running Mean Average Precision @ 5")
        return "metric"


class MRR(Metric):
    def calculate_metric(self, data):
        print(f"Running Mean Reciprocal Rank")
        return "metric"


```

```
class NDCG(Metric):
    def calculate_metric(self, data):
        print(f"Running Normalized Discounted Cumulative Gain")
        return "metric"


def evaluate_models(train_data, metrics: List[Metric]):
    res = []
    for metric in metrics:
        metric.calculate_metric(train_data)
        res.append(metric.calculate_metric(train_data))
    return res


metrics = [
    NDCG(),
    MRR(),
    MAP5()
]

res = evaluate_models(train_data, metrics)
```

We can directly see the driver function shrunk in terms of what it needs to know to operate on the various metrics. It is also shrunk in terms of lines. Reading and debugging become a lot easier. The team releases this fix and checks with the client to confirm that they are getting similar estimates for the MRR metric.

Unfortunately, the customer lost 2 weeks in December, tracking this error. So they switched to their legacy recommender system. It will take more effort on their part to re-enable the team's recommender as the primary pipeline. The customer is starting to completely lose faith in the team's results. They are thinking of hiring further third party auditing services, and the costs are accumulating. Not to mention that peak holiday season is upon us.

The customer ends up taking a second chance on the team's recommender system. The client deploys the latest packaged pipeline. "That was close folks! We could have lost this customer forever!" the product manager says as he reads the latest email from the retail customer. "They confirmed that they are re-enabling our recsys pipeline" he adds as he lets out a sigh of relief.

Heuristics

Happy endings aside here are a couple heuristics to detect if your code is violating the LSP:

- There are degenerate functions in your class that do not do anything except to satisfy the parent class contract.

- There is a derived function that raises a NotImplemented exception because the author of that function does not want you to call that function.
- There is a bunch of branching logic with the `isinstanceof` check all over the place.

These should not happen inside the application context. In the Main division, with all its natural dependency on the application modules, it is hard to avoid such an issue. This is because on the main side, the application needs to know a exactly what to do, what to load, and how to configure a bunch of dependencies.

Isolating from Change I: The Interface Substitution Principle

Make fine-grained machine learning components interfaces that are client-specific.

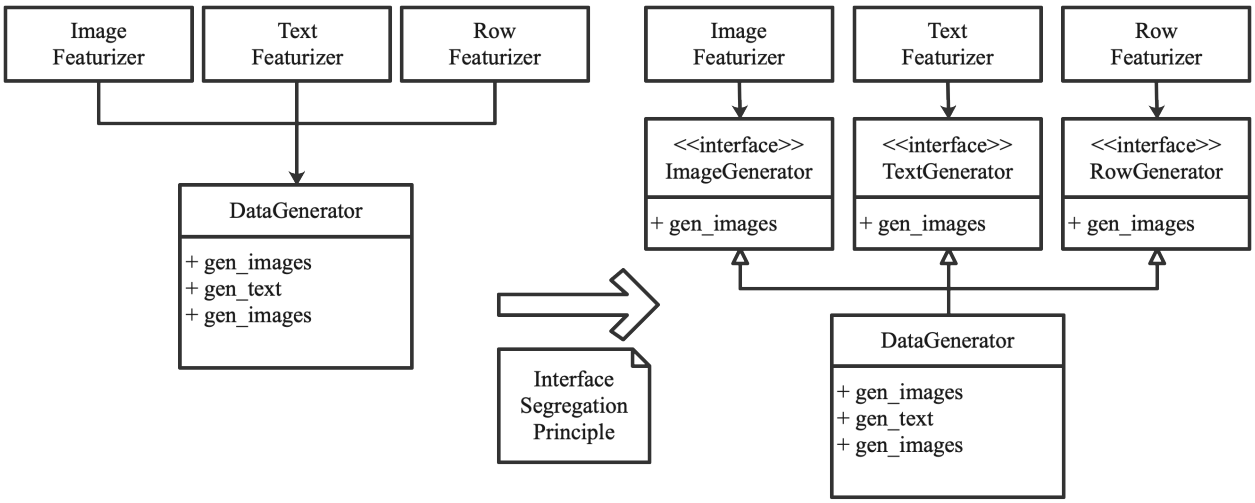


Figure 20: Interface Segregation in action

In the above diagram, we have a simplified example of an interface segregation strategy. Here an ML component, such as `ImageFeaturizer`, depends on another component `DataGenerator` for the function `gen_images`. The `DataGenerator` has many other unrelated operations: `gen_text` and `gen_rows`.

The problem is that changing gen_text in component `DataGenerator` might lead to the need to change another component such as `ImageFeaturizer`. This might happen even if `ImageFeaturizer` does not care about `gen_text` in the `DataGenerator`.

The goal of this principle is to avoid depending on components with extra baggage. This extra functionality might impact users laterally. Users may need to implement unnecessary functions to be compatible with their dependencies.

The Interface Segregation Principle advises creating client-specific interfaces. This is considered better than one big generic interface for all clients. The clients would depend on less stuff, which means that they have fewer reasons to change.

But that's now all! If you buy one, we give you a second one 50% off. The very next week, the ML project manager shows up. The PM asks us to modify the `ImageFeaturizer` to filter out any images that might contain sensitive categories (drugs, nudity, firearms, etc.). The auditing team is concerned that the training data might bias the classifier. If only they knew; that's all the data does; bias the model. But no harm is done. Indeed, by segregating the interfaces, we can provide an alternate implementation of the data generator for the images shown below.

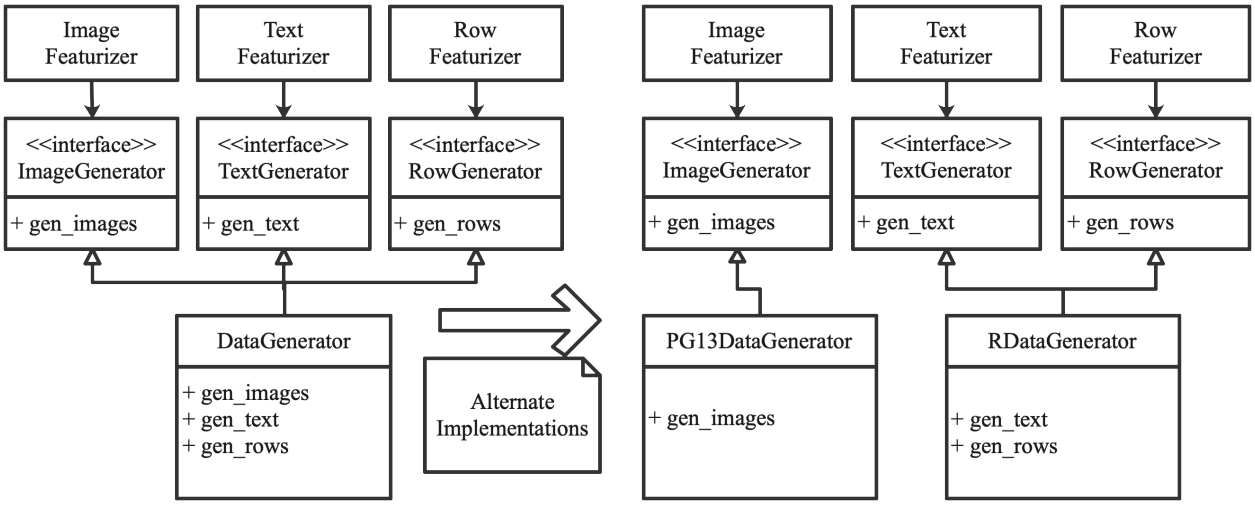


Figure 21: ISP evolution

One final benefit of this ISP is that the data generators do not need to implement unnecessary methods.

In dynamic languages like Python, there is no absolute need to implement this ISP explicitly through the abstract classes method. The language features are flexible enough to accommodate fat classes. But who wants fat classes to be littering their codebase. The ISP's core message is that modules and classes should not depend on things they don't need.

By knowing too much, a class becomes hard to modify, and developers become afraid of touching such classes. Also, large classes live in the same source file. The mere mechanics of editing the same file becomes a hassle faster than melting ice cream in a toddler's hands. The developer will be stepping on each other's work and merge conflicts will be the norm rather than the exception.

For these reasons, we are encouraged to build small classes with narrow interfaces to satisfy the ISP and make clients and users less prone to collateral damage.

Isolating from Change II: The Dependency Inversion Principle

Depend on machine learning pipeline abstractions, not on machine learning pipeline concretions.

How can we still be talking about dependencies? This even has it in the name.

It is common sense that dependency arrows should point toward the stable components in your machine learning pipeline. If you expect an unstable component to change often, protect the components that need to depend on it. The dependency inversion of control design method helps do that.

This principle allows for a plugin architecture. The application can be extended with external plugins. This is done by creating interfaces. These interfaces isolate the application core components from the plugins.

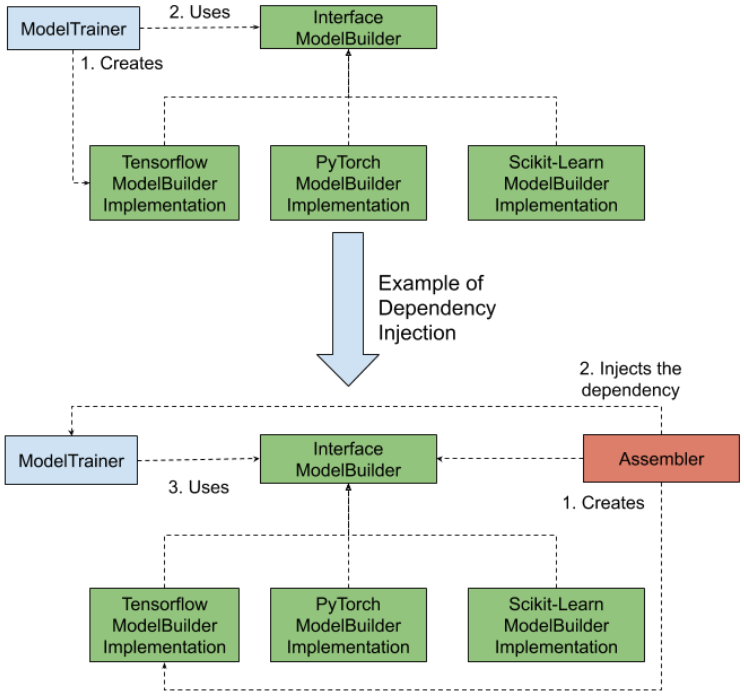


Figure 21: DIP in action

Take the interaction, in the above figure, between a ModelTrainer class and various machine learning libraries. The ModelTrainer goal is to build some sort of classifier. Depending on the predictive performance, we might need different libraries. Here the candidates are Scikit-Learn, Tensorflow, and PyTorch.

The application's core goal is NOT: "Let's use one of these excellent ML libraries in our project to sound smart and cutting edge at my next salary review."

The CORE goal is to build models/applications that bring value to users. That's how you win. The ML libraries and fancy big data processing libraries are nothing but details that should not interfere with the application core.

Dependency injection is one of the popular methods to perform this dependency inversion. It helps isolate the core classification application from the ML library plugins. First, we make the model trainer depend on the `ModelBuilder` interface. This makes the `ModelTrainer` able to use any model builder implementation interchangeably. However, we can take this further.

The `ModelTrainer` is a high-level component that should not know about low-level concerns. These low-level concerns are in the individual implementations. To resolve this, we can use an `Assembler`. This "wires" the model builder implementation with the model trainer at run time. The `ModelTrainer` can ignore the specific model builder the application is using. This removes the dependency from the model trainer to the model builder implementations.

The Dependency Inversion principle has been called a *"25 dollar explanation for a 5 cents concept."* And that's ok. If you find this principle dull to grasp, more power to you. For the rest of us, we need to see a code example don't we?

```
class ScikitLearnModel(object):
    def fit(self):
        pass

class ModelTrainer:
    def __init__(self):
        self.model = ScikitLearnModel()
    def train_model(self, ):
        self.model.fit()
```

Here, the `ModelTrainer` class is made to depend on the `ScikitLearnModel` class. `ModelTrainer` is the high-level policy, whereas `ScikitLearnModel` is a low-level detail. This design violates the DIP. A high-level module depends on a low-level level component. It should not depend on a concrete implementation.

Imagine that we are in charge of a facial recognition classification model. All was good for a while, bonuses were flowing, PMs were happy, customers were inviting us to their HQ for talks, we published a paper and two blog posts about it, the CEO herself sent us a box of chocolates to thank for delivering such a ground breaking product. However, 3 months in, a core predictive performance metric started to drop on the latest data from this month. We are now required to explore other model types. Maybe we want to try some of the deep learning libraries such as Tensorflow or PyTorch. We are confident that it is a good investment. The runner-up to the best paper award at a recent AAAI conference was using them.

And by chance, they were studying a specific type of fine-grained regularization. We hope it can reduce our model variance, which has been skyrocketing for a specific sub-sample of the data.

Now imagine that in this complex image recognition pipeline, we have multiple locations where this ModelTrainer is used. A while back, Benji was pleased with the results we got from the stacked ensemble. David agreed to give it a try. We were sold on the idea. So 4 months ago, we ended up having to stack 20 different models. We strategy was to average them to produce the final propensity score for predicting what toktikot video the users will watch to completion based on the video content's faces.

Today, nobody wants to touch those 20 models, and the possibility of a full rewrite lays in the balance. The PM hears about this and says "A full rewrite! Please don't do this to me folks! I have PTSD shivers when I think about the amount of time this will take!". The problem is that we need to MODIFY ALL THE INSTANCES of the ModelTrainer class around an already tedious code base. This violates the OCP and is terrifying for everyone on the team.

The ModelTrainer class must be made to care less about the type of Models it is using. We end up video calling our Mark our OO Design friend. "Are you still calling me about this? It is 10AM for goodness sake, I am barely awake!" he says. "I am tired of repeating this! *Abstraction and Inversion*" he adds. We don't even say anything, he sees us nodding and terminates the video call.

We end up making a Model interface:

```
from abc import ABC, abstractmethod
class Model(ABC):
    @abstractmethod
    def train(self, ):
        raise NotImplementedError
```

The `Model` interface has a `train` method. With this abstraction, we pass in a model object of type `Model` to our `ModelTrainer` class:

```
class ModelTrainer(object):
    def __init__(self, model: Model):
        self.model = model

    def train_model(self):
        self.model.train()
```

So now, we feel pretty good. No matter the type of `Model` we pass to the `ModelTrainer`, it can easily orchestrate the model training without bothering to know the type of model. We can now extend our `ScikitLearnModel` class to implement the `Model` interface:

```
class ScikitLearnModel(Model):
    def fit(self):
        pass

    def train(self):
        self.fit()
```

Now we are free to create multiple `Model` types and pass them to our `ModelTrainer` class without any complaints about runtime "Refused Bequests." Which is for the rest of us errors that happen when we call a function that either does not exist, or requires more, or provides less functionality than expected.

Here are some example of models classes that would play well with this framework

```
class TensorflowModel(Model):
    def train(self):
        pass

class PytorchModel(Model):
    def train(self):
        pass
```

Now in the `main` partition, we can wire this up in a nice assembler block.

```
def main():
    # Get your ingredients ready.
    model_of_the_day = PytorchModel()
    # Whisk well and serve chilled!
    model_trainer_of_day = ModelTrainer(model_of_the_day)
```

Finally, we can see that both high-level policy and low-level classes depend on abstractions. `ModelTrainer`, a high-level module, depends on the `Model` interface, and the `ScikitLearnModel` types, low-level modules, also depend on the `Model` abstraction.

Did we reach our goal?

Partly.

Dependency Inversion decreases coupling between a class and its dependency. The less the coupling, the less the number of components that need to change in lockstep.
But that's not all. As we have seen, it is a refactoring that does not alter the behavior of functions. The behavior is the same. However, the connections to this behavior are less sticky. This principle also allows us to externalize the configuration as follows.

```
def main():
    model_type = os.environ['MODEL_TYPE']

    # Get your ingredients ready.
    if model_type == 'pytorch_v76':
        model_of_the_day = PytorchModel()
    if model_type == 'tf_v91':
        model_of_the_day = TensorflowModel()
    else:
        model_of_the_day = TensorflowModel()

    # Whisk well and serve chilled!
    model_trainer_of_day = ModelTrainer(model_of_the_day)
```

What about that ugly branching here? Is that not an OCP violation? Observe that we are making these choices and these wirings in the `main` partition. This is far as we can get from the application core. Here all the dependencies point toward the application core. This also means that the dependencies of this `main` function all point towards more stable components.

This is precisely the place where you should put all the configuration, loading of classes, generation of objects, and so on. Since no component depends on this main partition, it is much easier to change without fear of breaking a dependency. We will discuss this in the next chapter when we look at the components of a clean architecture[3]. This is all going towards promoting reusability, testability, and maintainability. I just had to end with a bunch of ilities.

Conclusion

So here we are, we learned about a bunch of design principles. These are supposed to help you navigate the tradeoffs you will face in your ML code. We started by talking about the need for using classes and objects to manage dependencies. Then we laid out the guiding principles of clean machine learning classes. We talked about the Single Responsibility Principle, the Open-Closed Principle, the Liskov Substitution Principle, the Interface Segregation Principle, and the Dependency inversion principle.
We went in-depth with code examples to show how to use our two essential tools for designing code: **Abstraction** and **Inversion**. In the next chapter, we will dive into the next higher level of abstraction: The Clean Machine Learning Architecture.

References

1. Robert Martin about the single responsibility principle: https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html

2. Liskov, Barbara; Wing, Jeannette (1994-11-01). “A behavioral notion of subtyping”. ACM Transactions on Programming Languages and Systems. 16 (6): 1811–1841.
3. Clean Code: A Handbook of Agile Software Craftsmanship. Robert C. Martin. 2008. Prentice Hall PTR, Upper Saddle River, NJ, United States
4. Fowler, Martin. Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.

Chapter 6 - ML Software Architecture

The purpose of ML Software Architecture

"*Damn it is Tuesday already!*", 11 am, and it is product milestone review week.

The little icon next to your calendar pops up. It notifies you that you have 15 mins to think up something to say about the architecture of the movie recommendation system for a brand new streaming platform that has some unique content.

We promised the client three months ago a better MRR metric, and we made good offline progress so far.

"*How hard can it be?*" you tell yourself.

You pull up the project directory in PyCharm and look at the mess. You remember that you changed a single line of code in the trainer code yesterday and broke 63 tests. You don't even know what parts of the system broke. You are knee-deep in a sludgy soup that is slowly turning into a sewer channel. It is as if a toddler was scratching at the screen or something. The code is so intertwined that it is starting to look like a Dostoevsky novel in the original handwriting, with wet pages stuck to each other from the smelly humidity of this codebase.

"*I have to tell them about the refactoring sprint. Even planes need servicing after 100k miles or so*", you think. "*Maybe I can play this card at the next stand up to get some time to slow down the feature requests and focus on cleaning up this mess.*"

"*Yeah, that should work!*" you tell yourself.

You grab that strong brew from your desk and walk triumphantly to the conference room, ready to play your newly found card. You enter the conference room. Dan, the product manager, is sweating bullets. He is saying that the streaming service data engineering folks had some resources constraints. Basically, John, Xun, and Srivdar all left in the past month, two on sabbaticals, and one left to a competitor. They are behind in delivering the browsing behavior data by 2.5 months.

"*The browsing data?!*" you hear yourself blurt out. "*That was our most valuable feature set; the model's performance will be completely messed up if we can't get that data in the training pipeline.*"

The project manager, who recovered a bit at this point, looks up at you and asks you to clarify.

You explain that "*without the browsing data, we cannot apply the item-to-item collaborative filtering, nor the user-to-item collaborative filtering, nor any collaborative filtering for that matter. The viewing events data is way too sparse for us to use. It would be like boiling the ocean.*"

"*What about the content-based algos? Aren't they enough?*" Dan, the PM asks.

You started sweating again. You start thinking to yourself, "*It is damn hot in this conference room. Those PID controllers on the AC unit were probably built by some Fortran and COBOL loving nut heads. Ah, those days when you could program firmware for devices and have a single, unique, and final release. It must have been nice. I guess the product recalls were not that great when they found a software bug in there. Giving refunds to 1000's of customers because the AC units were catching fire because of a software change must have been a nightmare...*"

"*Hey! Are you ok? Are you with us?! Were you going to tell us about the content filtering part? Can we recommend the movies based on the content alone?*".

A crazy 5-hours debug session flashes in your mind in 2x speed, where you had to literally print the code on paper to be able to study what it did.

"*Well yes, we can wing it and do better than the* `most-popular-v5` *recommender the client uses right now. But not by much. 3 to 5% improvement, max, if we stack the models we have now. We promised the client that the new pipeline would reach 10 to 15% improvement on the MRR metric. We could probably explain that it is an intermediary result towards the final product. But that's not the real issue. The code is in no state to switch from behavioral to content-based data sources." you say, slumping on the meeting room chair. "The whole codebase was hardcoded to fit the behavioral data stream. That assumption runs so deep that the whole thing is so coupled to the behavioral data pipeline. I had to dream about it to organize it in my mind.*"

Dan moves his chair further away from you and says, "*So a code quality issue then?*".

"*Nah..*" you respond indignantly, "*more like a deep architectural issue.*"

The PM's face was slowly changing to a pale yellow color, and you don't like that when it happens. You worked with this team for so long, and you have a reputation to keep.

The PM asks you, "*Can we take this offline? I don't want to derail the conversation about deliverables by talking about code quality issues.*"

"*Offline?*" you think, feeling offended. "*This is not an issue that can be handled offline. I mean, even planes need servicing after a certain number of flying miles, right?*"

"*Yes, but we didn't even release the first version? How could it have gone that bad that fast?*" says the PM.

"*Well, you know it is not like building a car in an assembly line, this kind of work. It is more like a soup. The more you add to this codebase, the more its flavor and smell change. I can tell you it is not good at this point.*"

Excuse me for interrupting this mythological story for a quick second. Does this sound familiar? Do non-functional requirements such as the architecture of your software projects get second class treatment? What is this "**architecture**" thing that people talk about anyways?

Third-party packages are NOT an Architecture

Brian from the MLOPS group starts doing his best to explain our "architecture." He describes how it is based on Python for the programming language, Pandas for the data transformations, Scikit-learn and Tensorflow for the modeling, MLFlow for the model registry and experiment tracking, Fastapi for the API, Airflow for the job scheduling, Kubernetes for the runtime, Prometheus for CPU/memory for compute metrics, S3 for the data storage, and MySQL for the recommendation system's machine learning metadata.

When he finishes, you glance at him and tell him, "*This is all good and correct, but what you just listed for the 'architecture' is like saying that a house is built with stones, cement, hammers, nails, doors, and walls. This has nothing to do with the architecture of the system.*"

Your remember reading somewhere,[1], that:

ARCHITECTURE

is about

USAGE

PSA 6.1

<sarcasm> Oh yeah, that's useful! Usage right. What a strange concept? The system should be designed around its usage?! Well, that's fresh. Are we really going to let the usage of this system pollute our shiny new Tensorflow/Keras graph embedding pipeline? Besides, we sold this whole system on the premise that it will be interoperable with the production Airflow instance. That's what the system is about, the dependencies, those sweet open source dependencies. Nobody cares about the actual application of the system and the value it will create. As long as we use the latest shiniest tools in the market, with the fully automatic machine learning pipeline generation, we should be golden. **<sarcasm>**

You burst out, telling the team, "*We are not even using the most basic architectural design principles of Clean Architecture.*" [1]

Architecture is about Usage

What are these principles, you ask?

If you open the source project of your repository, what do you see? Do you find a smattering of third-party packages, intertwined in a single `job.py` file? Do you find the delivery mechanism ruling over the application? Do you find that the data movement libraries are drowning the use cases?

You want to see the use cases first and foremost when you open that folder of code. We want a strong separation between the data loading libraries and the use cases. We want to be able to avoid having to download a large file from S3 and integrate with boto, just to satisfy the need of some obscure section of the code. We want to be able to avoid spinning up a whole 16 GPU machine to write a simple test. We want to be able to prevent any side effects when testing the pipeline. We want to be able to test our code without having to launch a full airflow job to test a simple hyper-parameter change. We want to be able to swap cloud providers in less than two weeks. Is that too much to ask?

Well, sometimes, it is too much to ask. However, what we strive for is having the option to delay decisions. We want to postpone the decision about the serving layer, the data source layer, the modeling framework, the feature transformation libraries. We want to be able to wait for as long as possible before making such important, and hard to change, decisions. Remember that "We will never know less than we know right now" [2]. We want all of these decisions to be independent of the use cases.

A good architect is someone that is good at deferring decisions. The implementation details, such as the data sources, storage, database, UI, ML frameworks, and tools, should be made to be deferred. A good architecture allows you to postpone decisions. A good architect knows how to keep options open for as long as possible. "A good architecture maximizes the number of decisions NOT MADE." [3]

Avoiding Chaos using Architecture

I can feel your skepticism radiating from here. Maybe an example would help. Here is a canonical story about the benefits of good software architecture.

Team Lambda was responsible for building a scoring pipeline for a batch-oriented recommendation system in the movie recommendation domain. The team needed to add a Scorer class that scores each user-level row of features and writes it to a database. Initially, the team thought they needed to decide on a database type, and define the database schema asap. They estimated two weeks to select between the eight different database products that would fit their "scores" use case and two additional weeks to get the database up and running, which was not in the initial plans. This would have pushed the initial release promised to the client by one month. However, one of the team members, Jimmy, pointed out that the database was not needed so early in the app lifecycle. The team decided on using a MockScoreWriter for now.

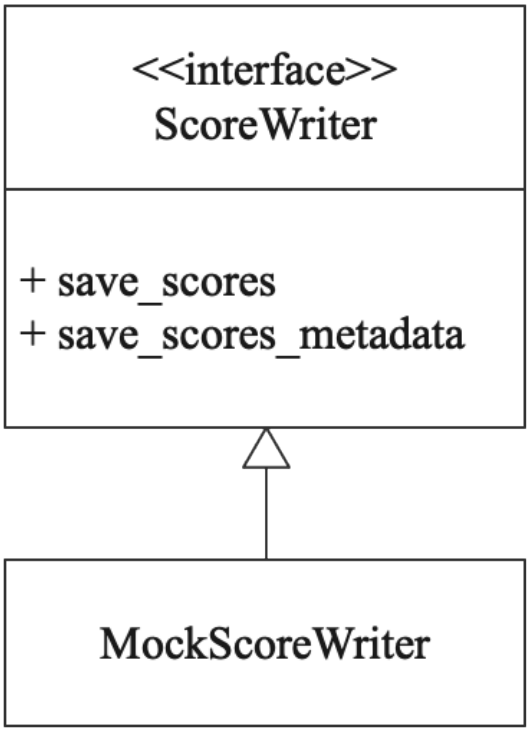


Figure 6.2: Initial MockScoreWriter

They were able to add all sorts of the feature engineering steps, test the feature pipeline, add the numerical standardization, test the one-hot encoding, add the model checkpointing code. I mean, they were flying with the features. Two months in, they had a working single-pass pipeline.

Then came the time to add the hyper-parameter tuning functionality. For that, the team had to combine multiple scores that needed to be persisted between runs. They required the multiple scores to compare the metrics and the metadata of each model candidate. The MockScoreWriter was good enough for the single model training runs. But when they had multiple models to compare, they needed to persist the scores across runs. The MockScoreWriter could not do that in its current form.

The team thought that maybe now is the time to select the database product, wait for procurement

to approve the license purchase, start it up, implement the DB specific code, decide on a schema, and start storing the scorers' data into the database.

But then Jenny pointed out that they can avoid the database for a little longer. The strategy was to use a derivative of the ScoreWriter class called InMemoryScoreWriter. This InMemoryScoreWriter was using a simple hashmap to store the multiple scorers across runs. This essentially acted as a test double for their application.

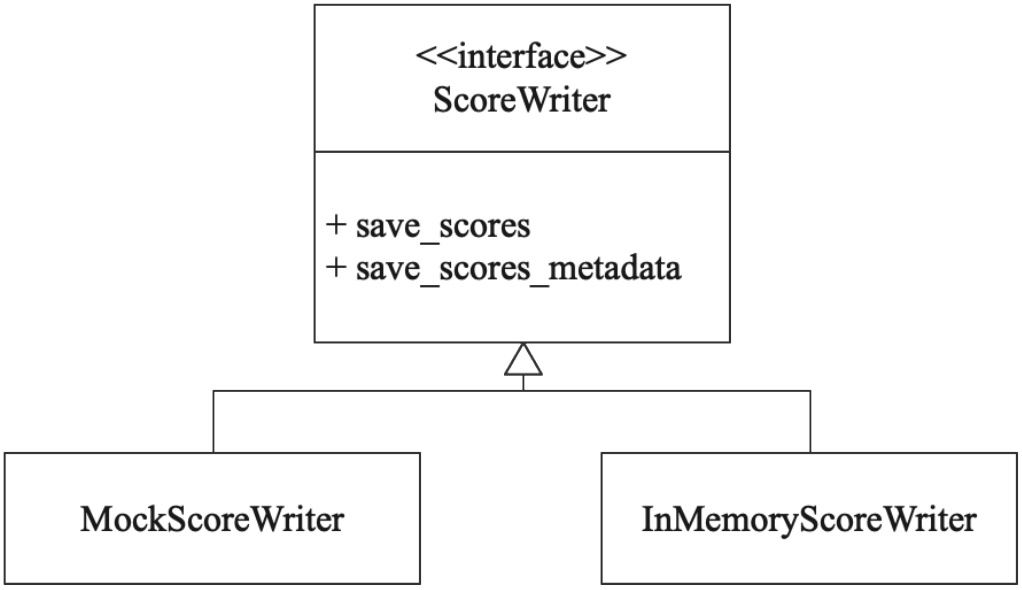


Figure 6.2: Adding the InMemoryScoreWriter

With the InMemoryScoreWriter, the team was cranking feature after feature related to feature selection, hyper-parameter tuning, and model selection at a diabolic rate. They basically got the whole system working. They were able to get all the use cases to work, they could write tests, they could create whole suites of tests, and most importantly they were able to run them. The only thing that was not working was saving the scores metadata to a persistent database.

At some point, the team reached a point where there were left with very few functionalities left that didn't require a database. Any new test the team came up with was bound to persistence across restarts of the application. So it dawned on them to use the same test double strategy to give the team the illusion of persistence while freeing their hands from complex integration tests.

A new FileSystemScoreWriter derivative was created from the ScoreWriter interface. This specialization gave the team another test double that can simulate a database. It could read and write scores data. The team was able to write a whole set of tests and run them after every change. With every

new feature came new tests, and the usefulness of the system grew.

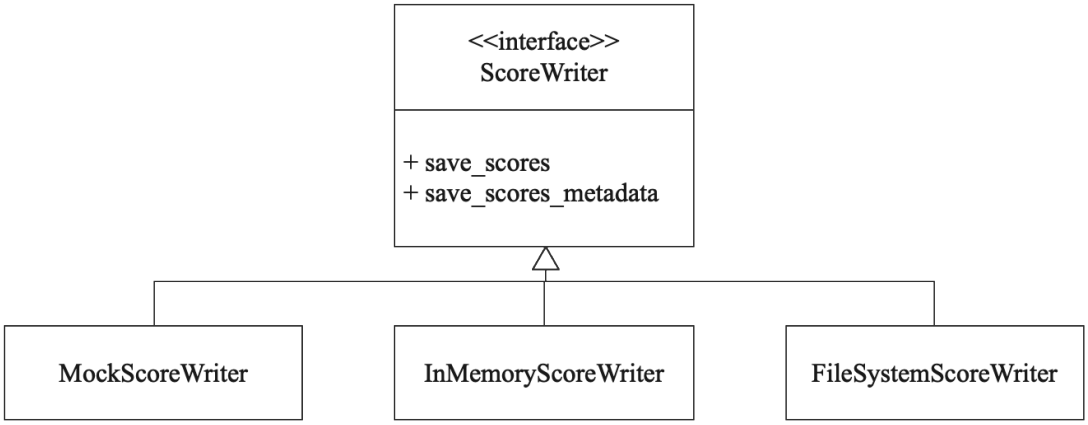


Figure 6.4: Adding the FileSystemScoreWriter

Then, Xiao, the product manager, looked at the state of the product and decided that from her perspective, this was enough to be a v1 release. The team was in disbelief, but after discussing it made sense to all of them. The application was useful, created value, and was self-contained. The team realized that they didn't need a database. The flat files system was good enough.

What the team did here is that they were able to defer a major infrastructure decision. This is primarily because the simple solution was good enough. Some decisions, like the database, framework, and tools, can be delayed for a very long time.

Six months after the first release, the team had a call with the customer's audit service. They said that they had to have the scoring metadata in a MySQL database. The team showed one of the client's support engineers how it would be possible to add support for a MySQL DB by extending the ScoreWriter class.

One week later, that support engineer came back with an implementation. It was working, it passed the tests, it had new tests, and the team accepted it in the repository as a DB plugin. That was a huge win for the team because the customer had a custom-compiled version of MariaDB that was set in stone for the past 3 years. The fact that the support engineer was able to contribute to this codebase raised his interest and trust in the team. He went on to give the team a non-negative review to his manager that was leading the integration.

It is pretty impressive to hear stories like this. Many new architects consider the data systems as the core abstraction of the whole system. They think that it would be impossible to develop a single piece of code before the database was up and running, and the schema was applied.

However, this canonical example shows what can happen when the central organizing principle is the ScoreWriter and all the entities surrounding it. The database was left as a detail, something we

can decide later, and inject at the last possible moment.

Frameworks and Harems

We can come up with many examples like the one above. I suspect that one of the primary reasons why you are reading this book is that all the frameworks, tools, databases, schedulers, and other ML tools have documentation that put themselves at the center of your architecture. The initial users of such tools find the examples and documentation easy enough to get started and included as-is in the system without any protective layer. When the framework changes, the system breaks, and immobility sets in. Software teams, and I am personally guilty of this as well, get usually seduced by the ease of use, the mirage of flawless execution, and the false promise, by the maintainers, of un-ending support for your particular business use case.

That is a dangerous and flawed assumption about framework owners. They build the tooling and framework because they love their product. They would not be able to work on such intricate pieces of code without having a deep passion for what they do. We, on the other hand, celebrate their efforts by adopting their tooling in our production pipelines. But that is not a strictly symmetrical relationship. Remember that even if you get married to the framework, the framework owner does not know you, and probably does not care about your product roadmap.

It is easy to corner ML pipelines into being part of a "harem" of users impacted by the whims of a framework maintainer, no matter how honest, helpful, and responsive this maintainer is.

Decoupling our system from the frameworks employed to achieve certain goals is a critical role of any good architect.

Let's get a visual. What is wrong with the following diagrams?

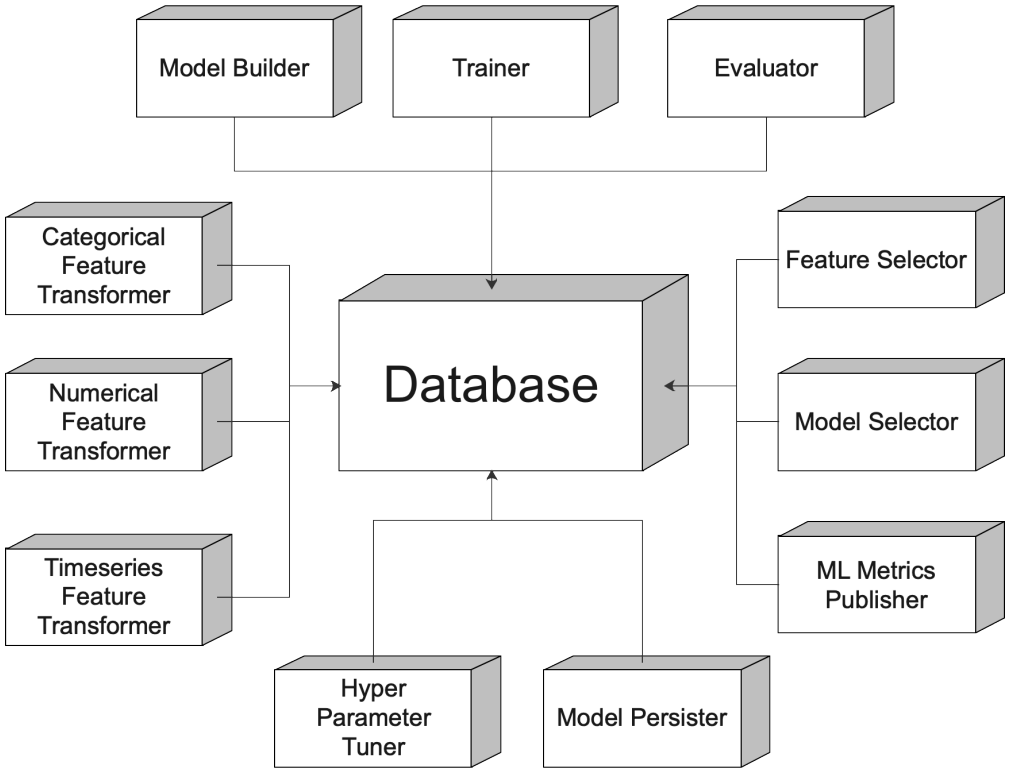


Figure 6.5: Should the database be at a the center of your application?

Should the database be the core abstraction of your system? Should your application be joined at the hip to a specific version of MySQL, MariaDB, Oracle, and PostgreSQL?

What about the scheduler?

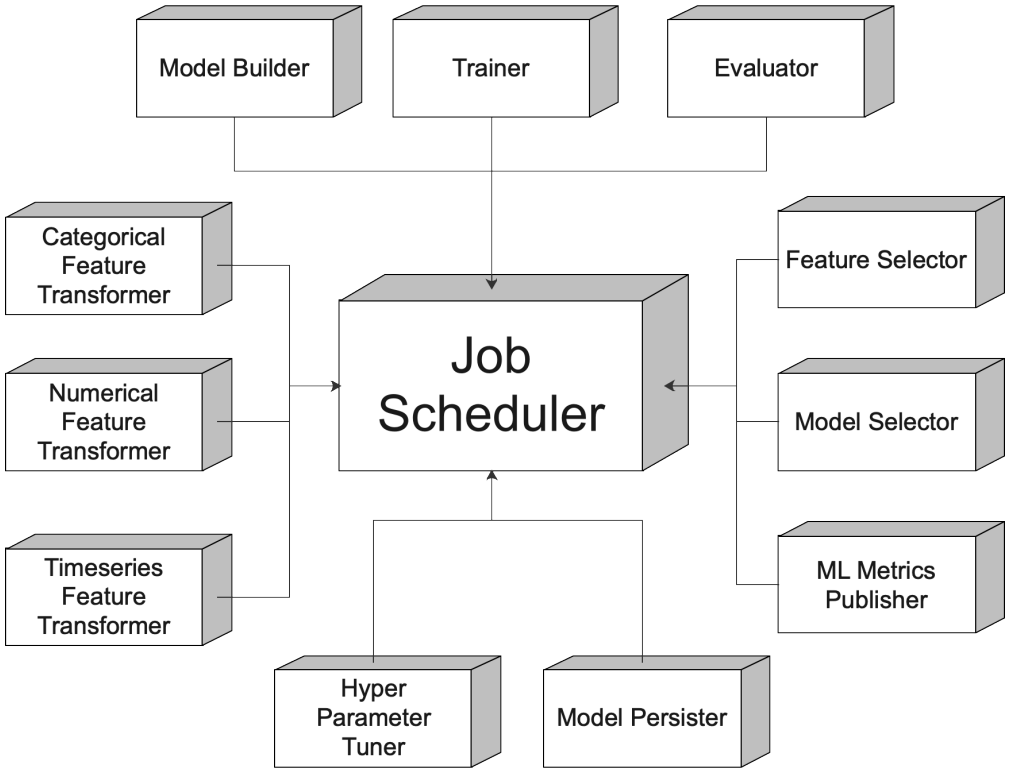


Figure 6.6: Should the scheduler be at a the center of your application?

Should we be ok with building the whole codebase for the primary purpose of running on a specific job scheduler? Does your application need to know if it is running on Airflow, Kubeflow, Celery, or any cloud provider scheduler?

What about the "Machine Learning Framework"?

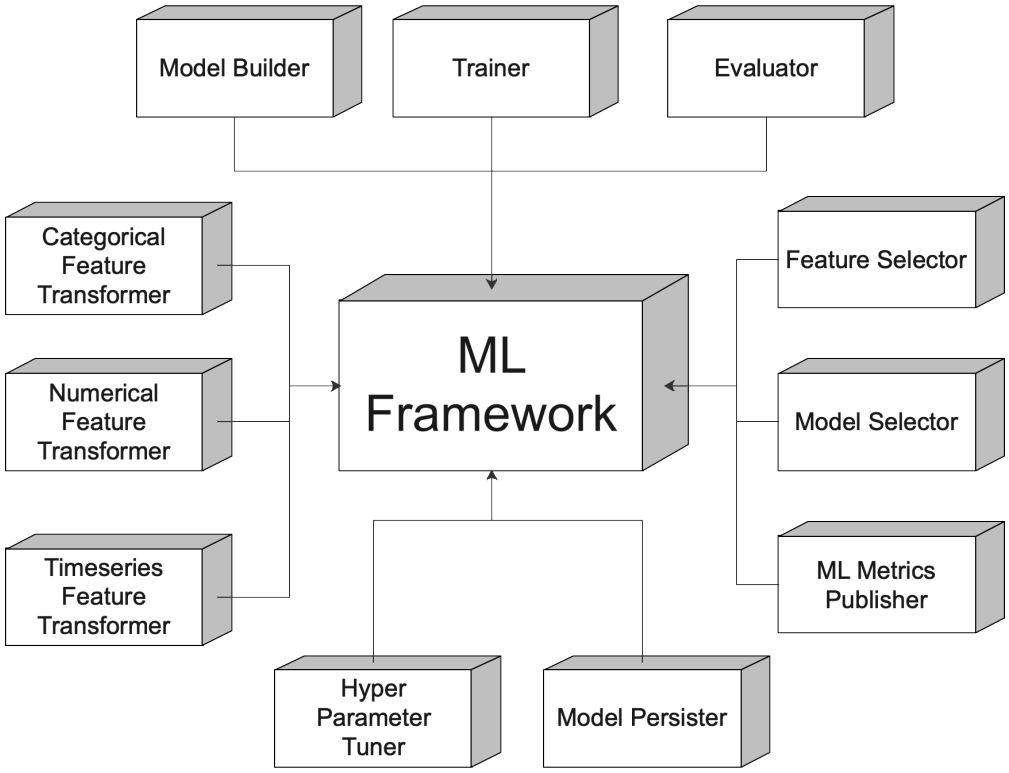


Figure 6.7: Should the Machine Learning Framework be at a the center of your application?

Is it sane to bind your application to a specific ML framework? Are you building a movie recommender system or a PyTorch model? Are you building a revenue forecasting model or a Tensorflow Extended model pipeline? What is more critical to the business? The framework or the use case?

The goal is to design a structure that decouples your application from these secondary concerns. But how do you do it?

Focus your architecture on your

Use Cases

NOT on the software environment.

PSA 6.8

The software environment is a detail that should serve the use case. Not the other way around.

Deferring decisions about the software environment allows us to keep our options open for as long

as possible. That means that we will be able to react to the changes in the market, roadmap, data availability, reliability SLAs, and change our direction if we need to. Probably not once, or twice, but multiple times throughout the lifetime of our project.

An added benefit is that this strong separation allows the business to evaluate the cost vs. value of each supporting component separately. If the business decides that the cost of GPUs for the deep learning solution is too high for the target use case, compared to the business value, we can swap in a non-GPU solution in a principled way. Focusing the architecture on the use cases promotes safe exploration as well as prudent decision making.

Defining ML Use-cases

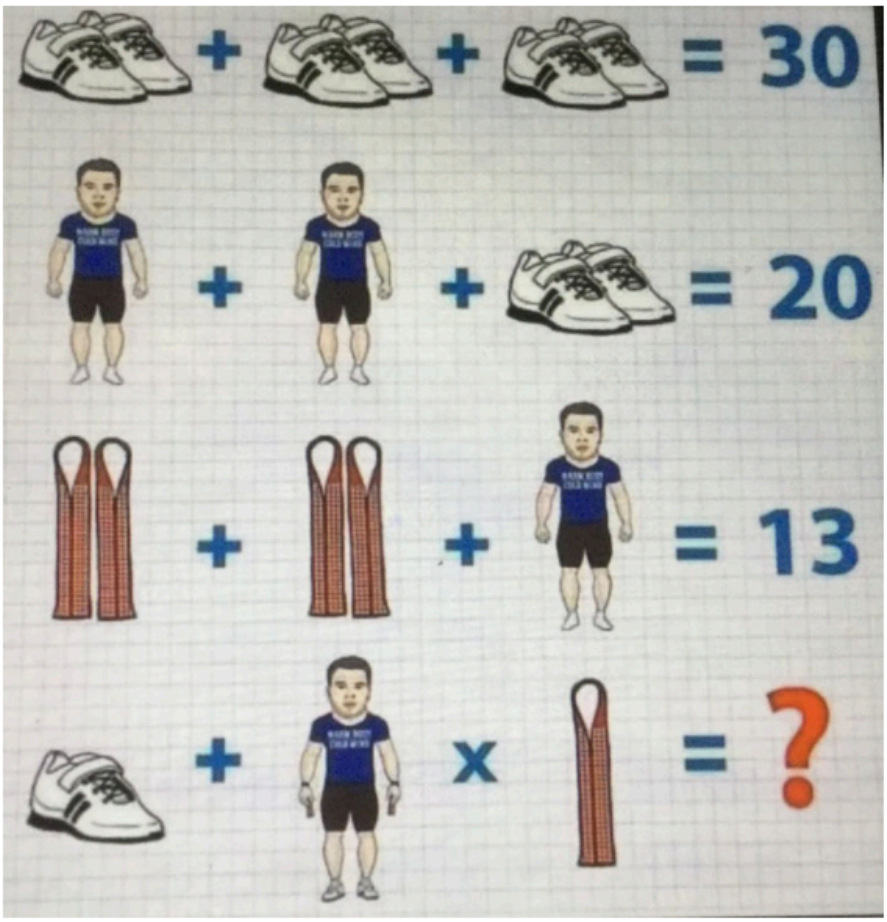


Figure 6.9: From Ridiculous Math Problems [4]

We have been talking about arranging our architecture around use-cases. What are these strange creatures? And what's up with the doll in the last line wearing shoes?

Let's be clear that I am not saying that all Machine Learning is doing is basically a bunch of Gaussian elimination and running large matrix multiplications. But I might be.

The use cases are quite different for ML applications. In web systems, the industry has taken 30~50 years to start being able to talk about CRUD (Create, Read, Update, Delete) use cases in a unified way. It took a revolution to create the Agile Practices. The concept that feedback was so central to the whole software enterprise seemed entirely new for many business people. The textbooks said

that when you ask someone to build an apartment building or a warehouse, the builders build that monument in layers. One layer after the other stacking things until the building is completed. This is an over-simplification of the building process, but it closely matches the business's interaction with the builder. That is not the case in software development.

After 30∼50 years, it still takes quite a bit of training to describe what the business is trying to build. The layered approach falls short dramatically. Software project managers understand now that software products are a soup that will be added to as time goes on. Therefore they must forget about writing a requirement document and check-in in 1 year. In this new world, frequent feedback is crucial to the success of software projects.

The developer builds something, shows it to the stakeholder, gets feedback, and goes back to fixing the issues and adding the new features that the stakeholder initially didn't know they needed.

The way requirements are communicated is through verbal directions. These directions usually mix business-level concerns with instructions for the technology, the frameworks, the tools, and databases we should use. But as we discussed, those are details that should be deferred. It is a hard thing to do, especially under deadlines and business pressures. This drives some developers to mix business concerns with system-level concerns. Not great.

Hopefully, people have thought long and hard about this problem as early as 1992 Ivar Jacobson released a book [5] where he describes a "use-case driven approach."

In that book, he promotes use-cases to understand how the users interact with this system in a **delivery-independent way**. The use-cases are supposed to describe how the user interacts with the system without using words like slider, dropdown, click, page, or app. He explains how the words and concepts that are used should not mention nor imply the existence of a delivery mechanism. The use-case should not have any language that would point towards a specific framework, database, or any component that is external to the business use-case.

This "simple" change of mindset shifts the conversation from an imperative "how" to do something to a more declarative "what" we want to do.

This changes the conversation. The use cases focus solely on the business process we are trying to automate or optimize instead of the implementation details.

These use-cases should be the central guiding principles, and the abstractions should be designed around these use-cases.

When we open the source code of the architecture, we should see the use cases of the system first. We should see the **intent** of the system as a 10x10 feet poster flashing in neon light in front of us.

Here is an example of a hyper-simplified use case diagram that covers a simple model training task.

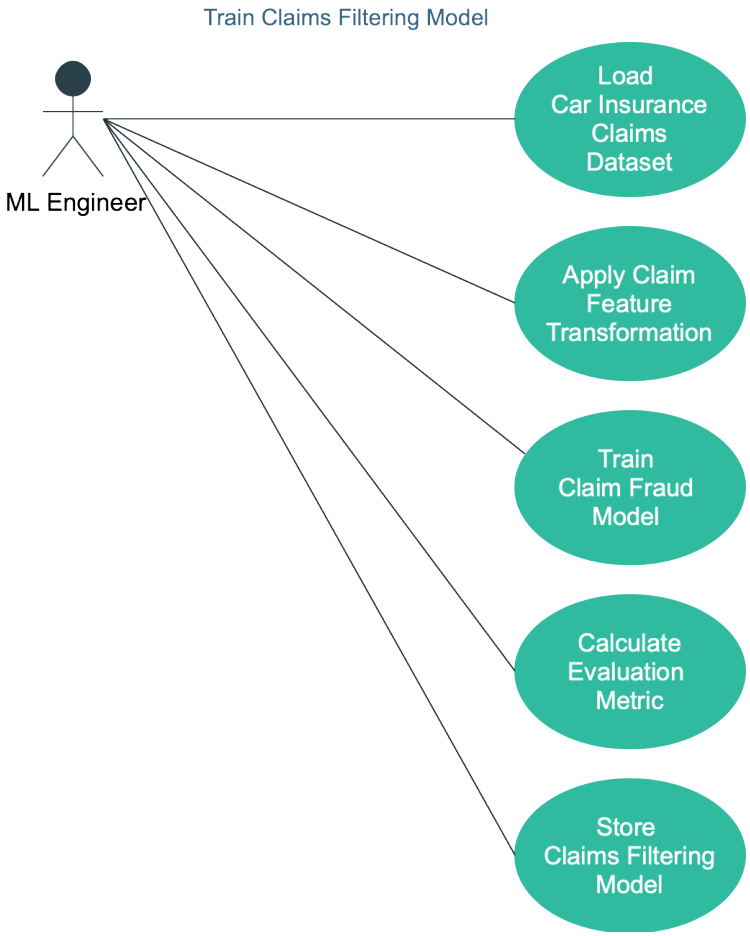


Figure 6.10: Basic use case diagram

The written version of this would look like this:

```
Use Case Name: Train claims filtering model
Actor: ML Engineer
Data: Car insurance claims dataset
Pre-condition: Claims data is available
Post-condition: The trained claims filtering model is
stored

Primary Course:
• System loads the claims dataset.
• System applies claim-level feature transformations.
• System Instantiates a new claim filtering model and
  trains using configuration.
• System uses the trained model to generate evaluation
  metrics using the evaluation dataset
• System stores the trained model.

Exceptions: Data Validation Error
• System reports the error message to the ML engineer.
```

Figure 6.11: Basic use case written version

As you can see, this use case does not have any vocabulary related to the underlying infrastructure, tools, frameworks, API, logs, or databases. It describes the data and operations that the system is expected to perform.

The point is that if we aim to have delivery-independent and infrastructure-independent architecture, we need to start with **delivery-independent** and **infrastructure-independent** use cases.

In a way, an ML use case is an algorithm for the ingestion of input data and the generation of useful artifacts like data, models, and reports. That means that we can implement an object that can perform that use case.

This hyper-simplified scenario is the gateway to much more intricate scenarios. As we see in the use case above, we need to include exceptions and exceptional course of actions for when things go wrong.

Writing use cases is an art form that can be learned with the help of some competent experts. Books such as "Writing effective use cases" [6], "User Stories Applied" [7], "Use Cases: Patterns and Blueprints" [8], and "Requirements: The Masterclass" [9] are great sources to learn how to build use cases. Unfortunately, ML specific use-case books are probably still being incubated in various publishing houses. For now, in 2020, we still need to rely on traditional software engineering practices to build ML use cases.

Assuming that we can create infrastructure-independent use cases, where do these business rules go? What kind of objects should hold these algorithms? And where would these use case objects fit in our ML system architecture?

Separating High Level Policy from Low Level Implementation

As we talk to stakeholders, we usually discover more business objects and more use cases that need implementation.

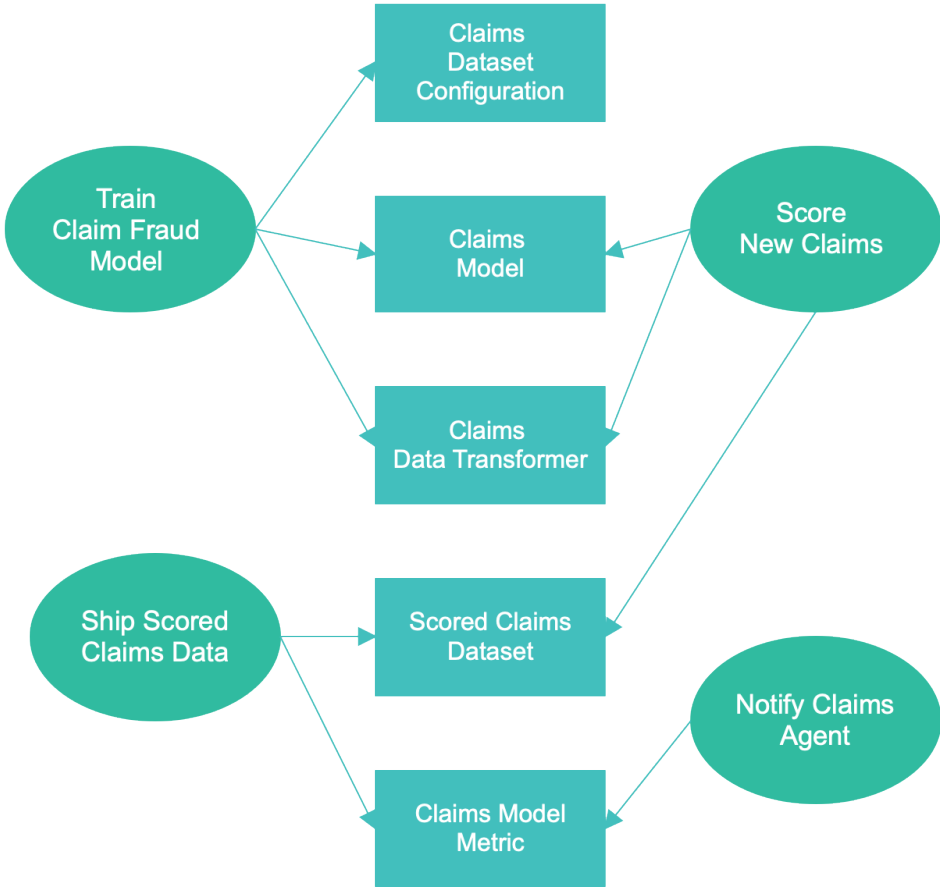


Figure 6.12: Discovering additional use cases and actors

At this point, if you are trying to fit this into an MVC (model view controller) model or a data transformation DAG (directed acyclic graph), please stop trying. We are discussing concepts that are one level higher than MVC and DAG concepts.

In the use-case approach described in the clean architecture series [3], we can identify three concepts:

- **Entities**
- **Use Case Objects**
- **Boundaries**

First, **Entities** are objects that contain application-independent business rules. For example, `ClaimDataset` purpose could be useful for making insurance plan recommendations, filtering fraudulent claims, and forecasting claims turnaround times for next quarter. The methods in entity objects should be able to serve the business at the level of all those applications.

Second, we have the **Use Case Objects**. The clean architecture calls them "Interactors," but I like the term "use case objects." They are application-specific, and they implement the rules defined in the use cases. For example, the `Train Fraud Detector Model` is a valid use case with a set of application-specific steps and a set of expected application-specific outputs. `ClaimsFraudModelTrainer` is a name for an interactor that would be responsible for implementing the use case.

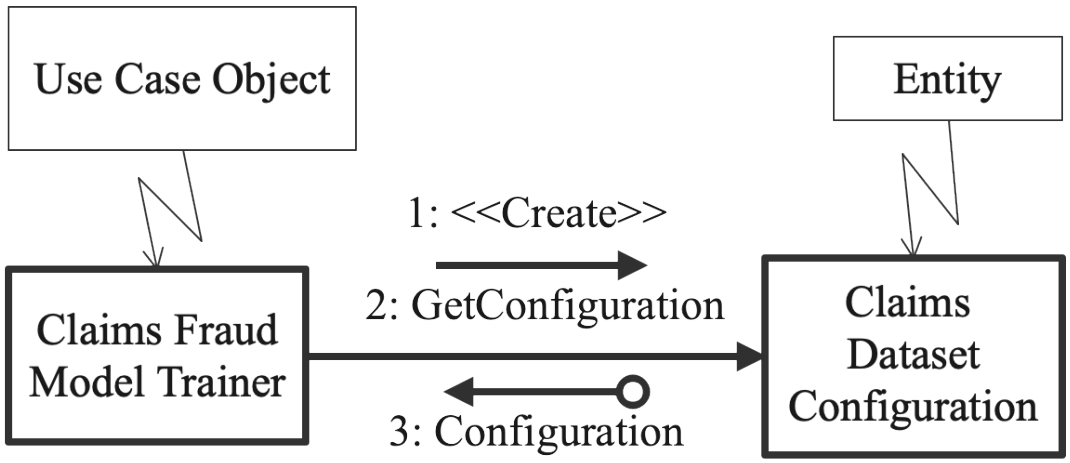


Figure 6.13: Use case object interacting with entity

While the entities are application-agnostic, it is the Interactor that orchestrates those methods to achieve the goals of the use case. In the picture above, the Interactor knows about the entity and uses its application-agnostic methods. However, the entity has no idea that it is being used by an application-specific interactor.

Third, we have the **Boundary** objects. This type isolates the use case from the delivery mechanism and the infrastructure and provides a communication medium. Things like CLI commands, ML API interfaces, Databases, and UIs are all on the delivery/infrastructure side of the boundary.

The Clean Architecture in One Picture

So how do these all fit together? Here is one version of the diagram that puts these three concepts together:

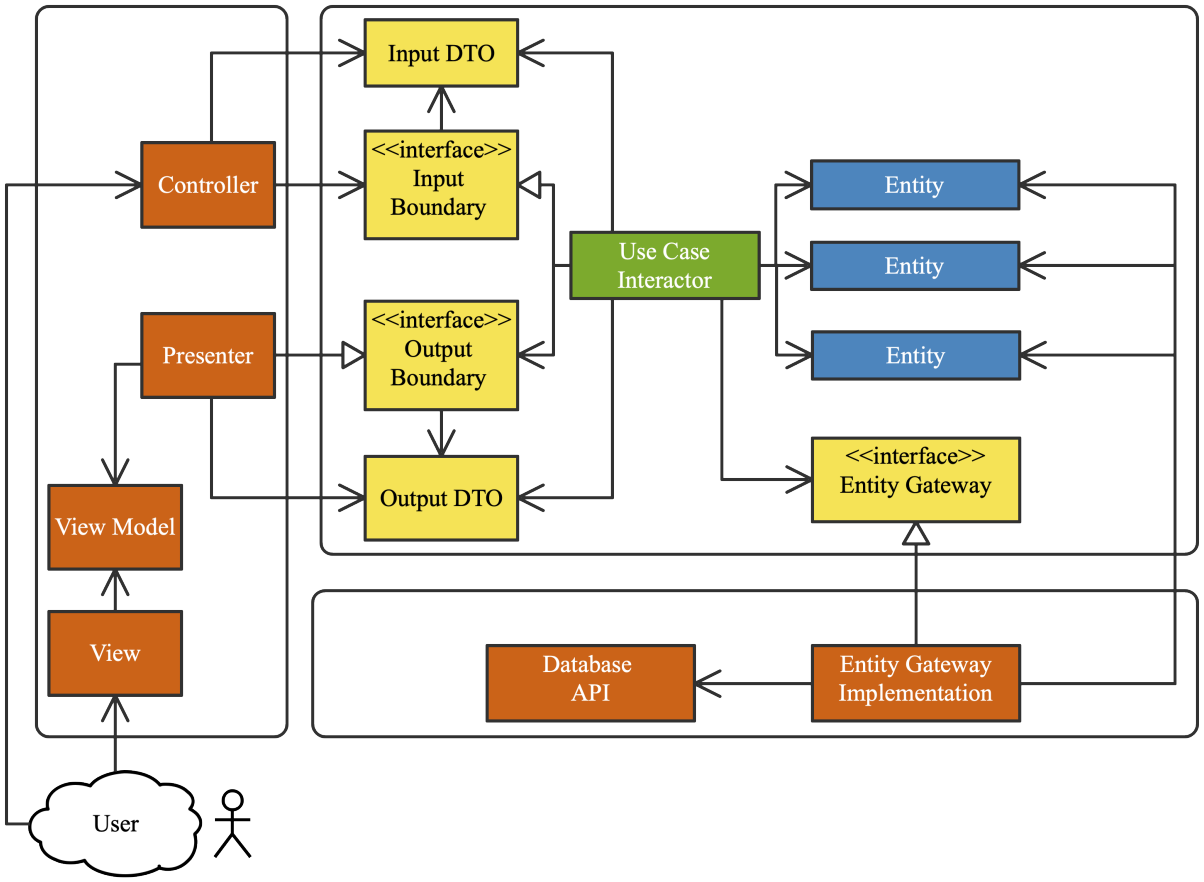


Figure 6.14: Standard Clean Architecture Components [3]

In this picture above, we have the three components working together. The flow of control goes like this:

1. The controller triggers the process (API, CLI command, Timed command) of running a specific use case.
2. The controller packages up the data in a nice little package using the input boundary object interfaces, in an input data transfer object (InputDTO) and sends it to the use case.
3. The use case interactor knows how to read the InputDTO because it implements the input interface.
4. The use case then starts the orchestration:
   a. Unpack the InputDTO.
   b. Create adequate entities for the use case.
   c. Read some data through the entity gateway interface.
5. The entity gateway implementation calls the concrete database API and retrieves some data and passes it to the entity gateway implementation
6. The entity gateway converts the data into entities and returns the entities to the use case through the interface.

7. The use case then gets these new entities and applies the business logic defined for the use case.
8. Once done, the use case can push data back to the data source through the interface, such as writing to a database some metrics or new entities.
9. The use case also returns data to the presenter through the output boundary interface, by creating an OutputDTO and passing across the architecture boundary
10. The presenter knows how to unpack that OutputDTO because it implements the output boundary interface.
11. The presenter does the formatting needed to send back output to the user.
12. The presenter generates a view model with all the data required by the view.
    a. The view model is another boundary object between the presenter and the view
13. Finally, the view exposes the data back to the user.
14. Aaand y'done!

Related Architecture Names and Concepts

This can seem pretty abstract, but before we look at an ML specific example, let us look at other representations of the same idea to get that sweet multimodal perspective. We'll take a look at the following:

- Concentric Layers Clean Architecture View
- Hexagonal Architecture, a.k.a Ports and Adapters
- Onion Architecture
- Clean Pragmatic Architecture

The Concentric Layers Clean Architecture View

In the original blog post and book on clean architecture, we are presented with the following view of this architecture:

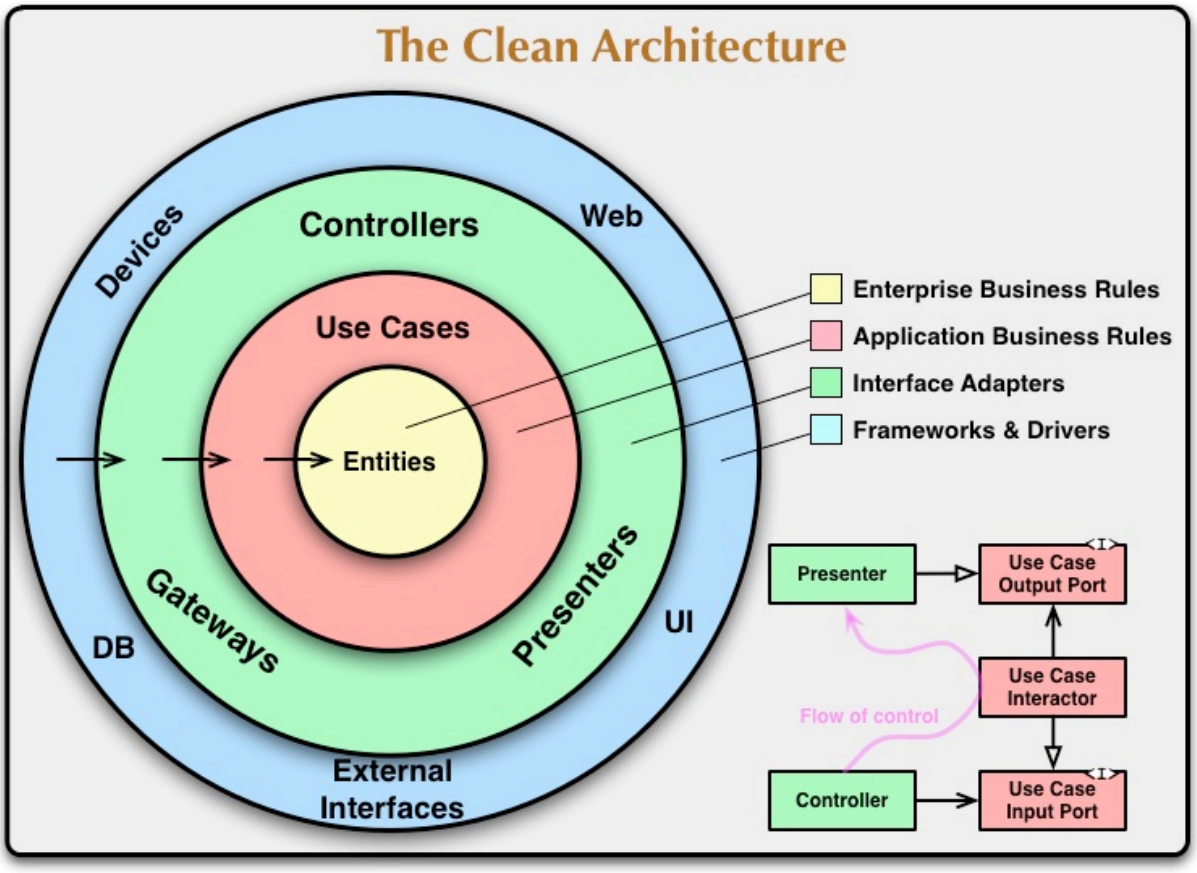


Figure 6.15: Clean Architecture Conceptual Diagram [3]

At the core of this architecture sit the entities. They are the same business domain objects we discussed. They speak the domain language and are application agnostic.

The next layer up is the use case with the interactor concept that sits between the entities and the boundary. This layer is where the use cases are implemented, and application-specific business rules are defined.

The next layer is the boundary. This is where we find the controllers, the gateways, and the presenters. They are the last frontier of sanity that is under our control. They are used by the interactor to orchestrate input/output operations.

The outer layer is where the madness of external dependencies starts. This is the wild world of devices, databases, UIs, web APIs, and anything necessary for our full application but is entirely outside of the application boundary.

On the right side of the figure, we find a simple demonstration of the dependency rule [10]. We see that the flow of control goes from the controller to the presenter. Isn't that a violation of the principles we described? No, of course. Notice how the source code dependencies point inwards. Even if the controller has to instantiate and control the use case, the use case does not know nor

care about what the mechanism is in the outer layer calling it. No component in the inner circles can know anything about the shenanigans of the outer rings. For example, a name declared in an outer circle must be completely hidden from an inner circle. That includes all our good friends, such as functions, classes, and variables.

The only interaction boundaries between the outer layers and the inner layers are the interfaces, and those are under the application control. It is the responsibility of the outer layers of our application to implement them and provide them with the use cases without snitching on their real concrete type. The use cases should not know or care about the concrete class passed to them to operate on. This is usually performed using the dependency inversion principle.

The same idea applies to the data movement. The data that passes from the outer layers to the inner layers need to be as generic as possible. The entity gateways sit in the adapter layer and are responsible for converting data from the raw, barbaric, and filthy data sources to the nice, tame, and civilized entities world.

In addition to the increased testability, this architecture promotes the removal of any inflamed external component. This protects the application from any external collaborators of the system that might become incompatible, upgraded by the providers, or just too old school. These include the storage system, database, or the machine learning framework. In that architecture framework, we can replace those collaborators with a minimum of effort.

The Hexagonal Architecture a.k.a Ports and Adapters

This idea has a longish history because people have been trying to decouple their application from their dependencies for a long time. One example of this history is the Hexagonal architecture driven by the following goals:

"*Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.*" [11]

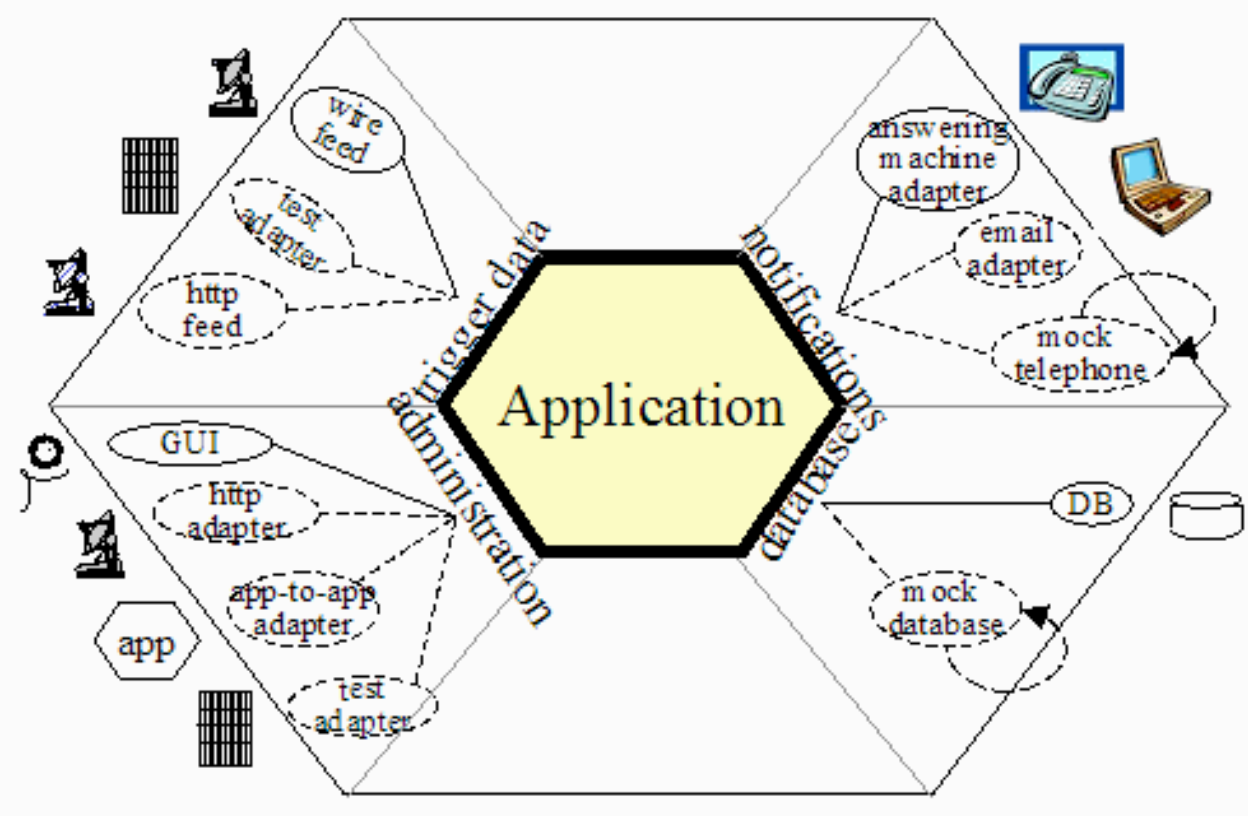


Figure 6.16: Original Hexagonal/Ports and Adapters architecture diagram [11]

The central concept of Ports and Adapters is that the core of your system is your application. All the inputs and outputs interact with the core of the app through ports and adapters. These ports and adapters isolate the application and protect it from changes in tech stacks, tools, and delivery mechanisms.

The application itself does not have any knowledge about who is talking to it. This has similar goals of securing the core system from the random evolution of technology, at one end, and business requirements at the other end. In this framework, the developer decouples the application from the supporting systems, in case they become inflamed, rebellious, or obsolete and need replacement.

The Onion Architecture

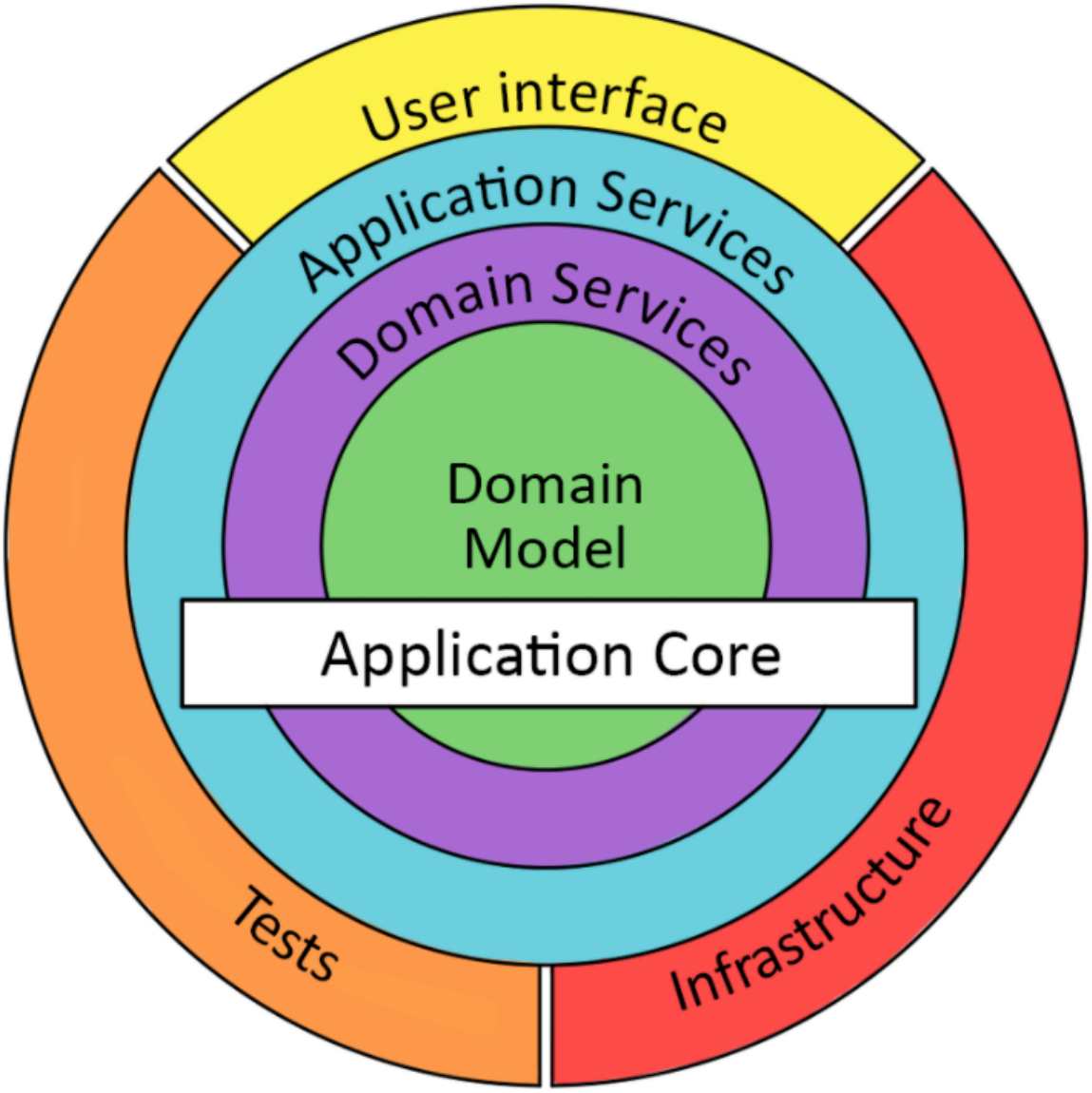


Figure 6.17: Onion architecture diagram [12]

This architecture style has a striking resemblance to the clean architecture and the hexagonal architecture. This architecture is also composed of a domain-centric design.

The domain model layer has the entities, the domain service layer has the domain-defined processes, the application services are the application-specific logic, and the outer layer is everything else (UIs, DBs, Tests, etc.).

Similarly, the **dependency rule** holds in this model: "**Outer layers can depend on lower layers**,

but no code in the lower layer can depend directly on any code in the outer layer."

In the original description of this architecture style [12], the author stresses that the major difference between the layered architecture and the onion architecture is the dependencies' direction. The point he makes is that in a layered architecture, almost all the layers can depend on the infrastructure. At this point, I don't have to tell you that that creates bad couplings. A change in an API framework could impact the business logic components.

Clean Pragmatic Architecture

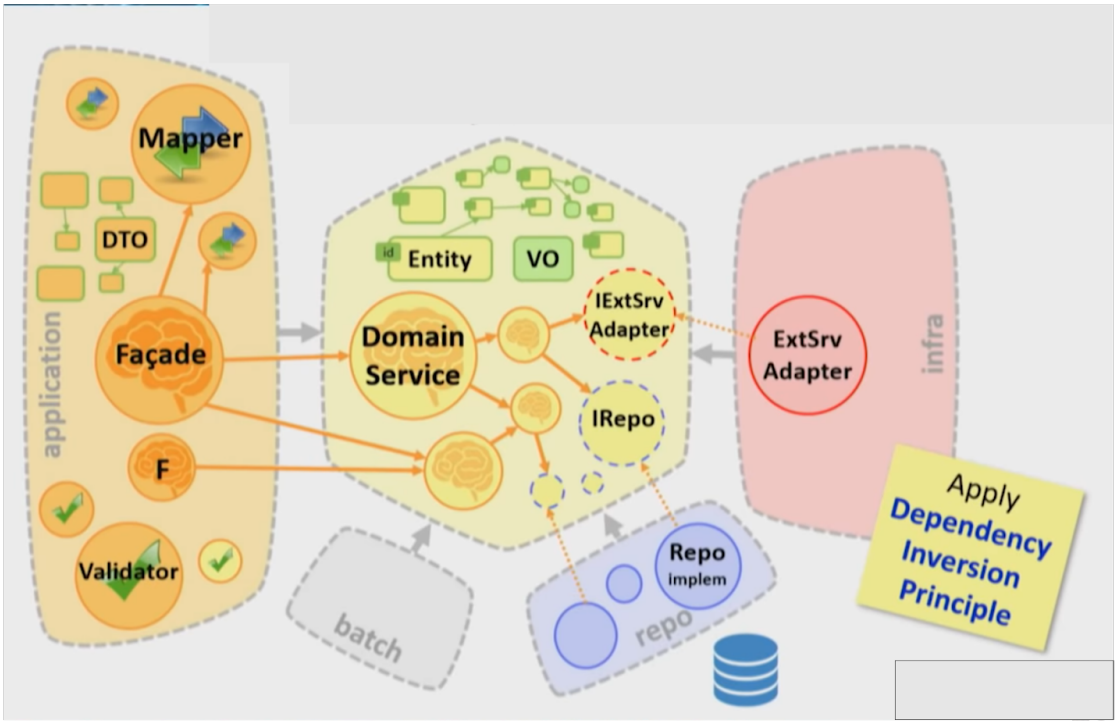


Figure 6.18: Clean Pragmatic Architecture [12]

The clean pragmatic architecture follows the same design goals. Decouple the domain from the application's facades and the infrastructure.

The ideas are similar to the clean, onion, and hexagonal architecture. Partition the system into core and plugins. The application interacts with the users and other systems. The facades are built to separate the core domain logic from the implementation details. This uses data transfer objects instead of boundary objects. It also uses mappers and validators in the application layer to deal with conversions and catching mishaps in the input interactions. This group includes any CLI, API, Scheduler, Web front end, or "ML Pipeline" component that drives the application. The domain is

kept in the dark and does not know anything about what is driving the application. Note that all the dependencies are from the application to the core domain service.

The external supporting infrastructure is also hidden from the core domain using adapters. Using the dependency inversion principle, the infra becomes a plugin to the code domain. This group includes any supporting database, APIs, file storage, remote supporting services, and anything the core domain needs to do its work. Notice how the domain communicates with the infrastructure strictly through the Interfaces such as IExtSrv Adapter, and IRepo. Again the implementations of the infrastructure depend on the higher-level policy of the domain service.

Friction and Boundary Conditions

"*In mathematics, in the field of differential equations, a boundary value problem is a differential equation together with a set of additional constraints, called the boundary conditions. A solution to a boundary value problem is a solution to the differential equation which also satisfies the boundary conditions. Boundary value problems arise in several branches of physics as any physical differential equation will have them. Problems involving the wave equation, such as the determination of normal modes, are often stated as boundary value problems.*" [14]

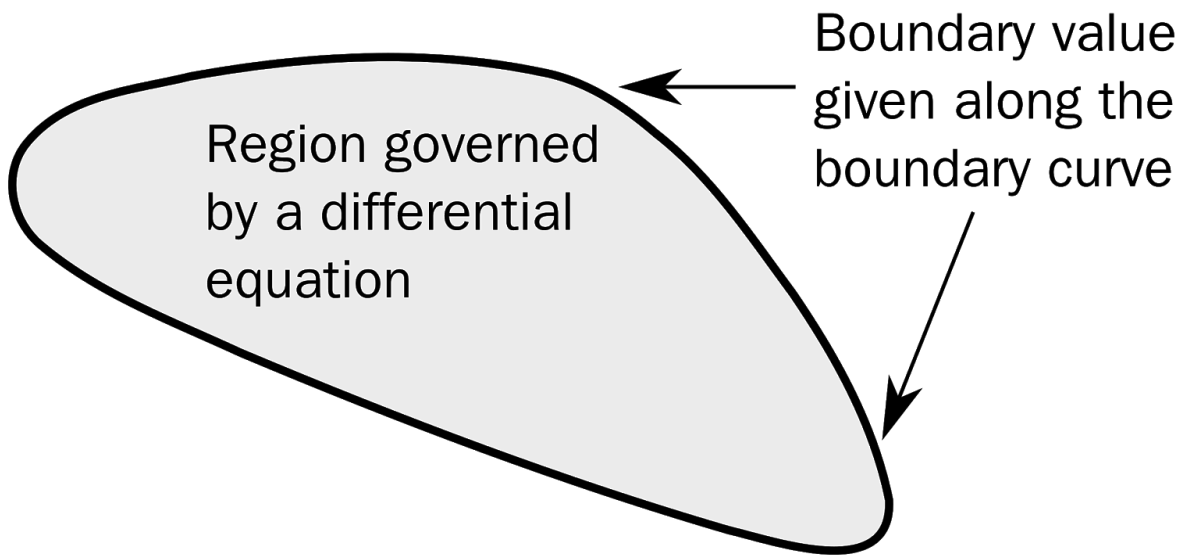


Figure 6.19

The architecture of a system is defined by a group of components and the boundaries that divide and unite them. Many of the architectural headaches come from designing sustainable boundaries.

In software applications, crossing a boundary is composed of a function on one side of the boundary sending a message to a function to the other side of the boundary. In its simplest form, the message is a function call that passes some data to the other side of the boundary.

The key to creating good boundary crossings is to manage the source code dependencies. When one module is modified, other modules might need to be adjusted, reconfigured, and redeployed.

The first boundary type that developers learn is the segregation of functions and data in a single address space. In this mode, there are no visible boundaries. The application looks like one large monolith with deeply interconnected function calls. When the number of functions that do similar things grows, newly minted devs sometimes start creating dictionaries of functions. These dictionaries are then used to select which behavior is needed. This makes the boundary configurable by passing a key and getting back a function. This manual polymorphism can be risky, and any hope of component partitioning is blocked.

```
# Definition
data_loader_funcs = {
   "s3": load_from_s3,
   "fs": load_from_fs,
   "db": load_from_db
}
# Usage. Here comes trouble.
data_loader_funcs['db']()
```

The next step that devs usually go with is object-oriented concepts to manage the boundary. So we end up with something like this:

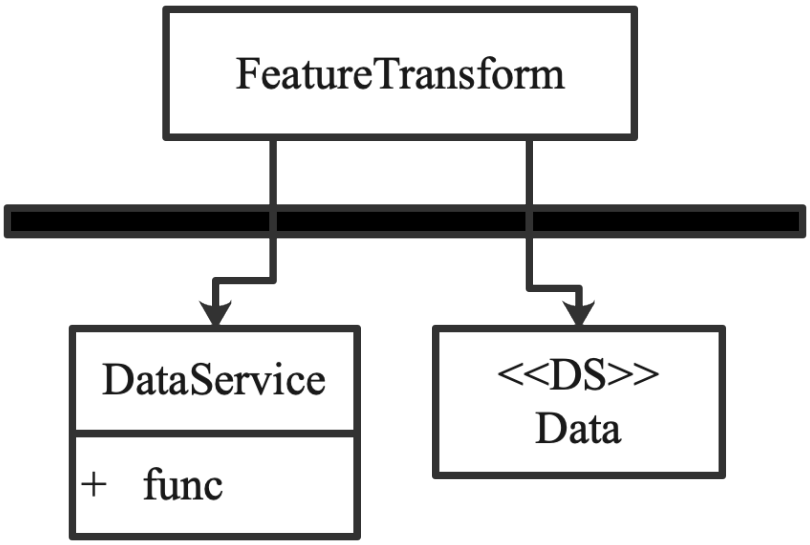


Figure 6.20: High-level policy depending on the low-level policy

The code is grouped in nice little boxes of functionality. Then the time comes to make the boxes interact. In the figure, the arrows represent the flow of control. The flow of control crosses the boundary from the high-level policy client to the low-level policy concrete service. The client passes some data to the service. This can take the form of a function argument or more complex methods.

The subtle, but flawed, part that devs do not realize immediately is that the **definition** of the "data" is wrongfully located on the **service** side. This creates a dependency from a higher-level policy on a low-level detail.

The primary method for handling this dependency is to implement dynamic polymorphism to invert the dependency.

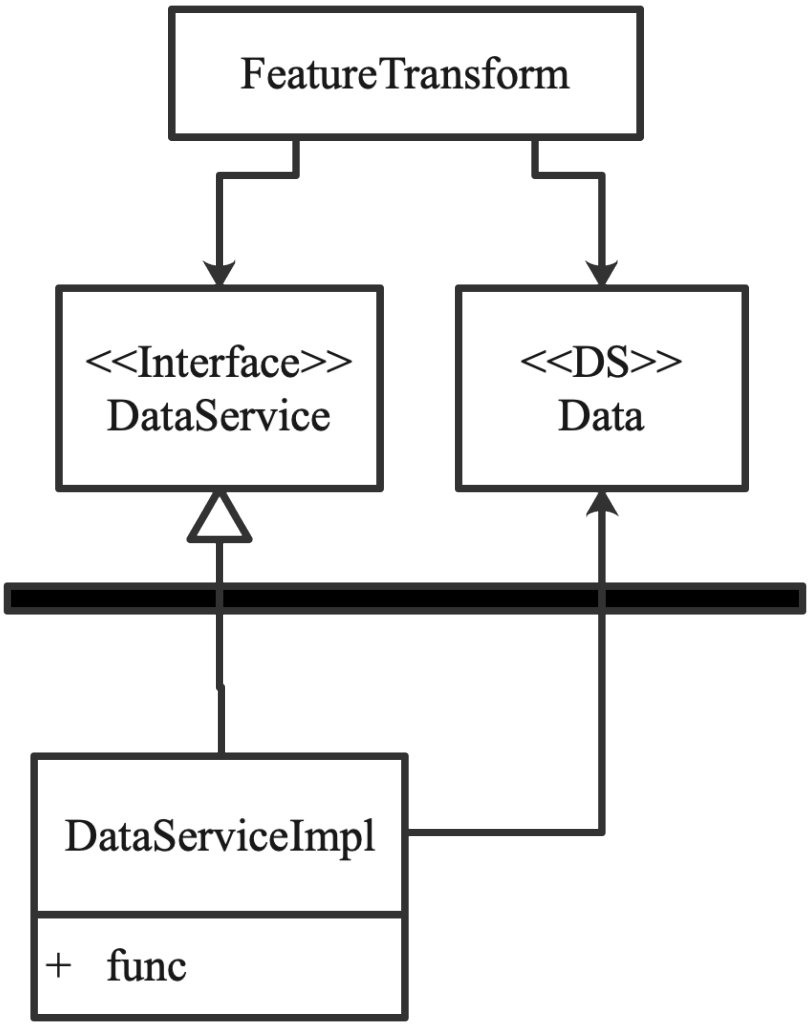


Figure 6.21: Low-level policy depending on the high-level policy

The flow of control still goes from the high-level policy to the low-level details. However, in this case, the client calls the service functionality through the service interface. The client does not know which service implementation will be ultimately used. This means that the implementation can evolve independently from the high-level policy. Now in the figure, we see how the source code dependencies run opposite to the flow of control. Note that the service interface and the data transfer structure are both on the core application side of the boundary. **This means that the service implementation that talks to the external service now depends on the interface that is defined in the high-level policy side of the boundary**.

This kind of partitioning is a great starting point to aid independent deployability, developability

and testability. The service developers can evolve their service without interfering with other devs. The high-level policy is independent of the low-level details.

Taming the Recsys Beast

Let's go back to the original story we started at the beginning of this chapter. Where did we leave this off? Ah yes, Dan, the PM wanted to deprioritize the architecture/refactoring time by saying, "*So a code quality issue then?*".

"*Nah..*" you respond indignantly, "*more like a deep architectural issue.*"

"*Can we take this offline? I don't want to derail the conversation about deliverables.*" he asks you.

"*Offline?*" you think, feeling offended. "*This is not an issue that can be handled offline. I mean, even planes need servicing after a certain number of flying miles, right?*"

"*Yes, but we didn't even release the first version? How could it have gone that bad that fast?*" says the PM.

"*Well, you know, ML eng is not like building a car in an assembly line….this kind of work… It is more like a soup. The more you add to this codebase, the more its flavor and smell change. I can tell you it is not good at this point.*"

"*Did you look at the architecture diagram lately?*" you venture bravely.

"*Well, I looked at the initial plans.*" Dan says, "*This one, right?*"

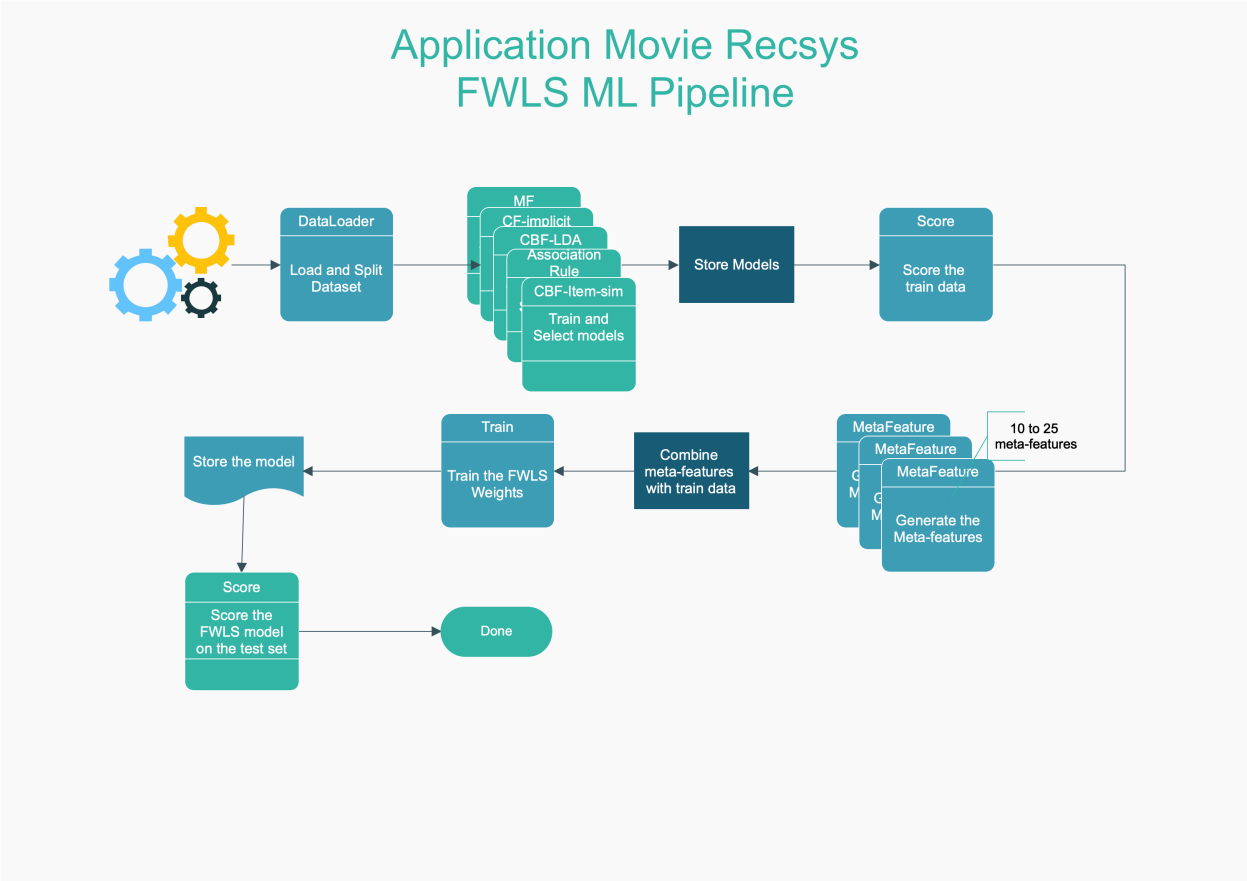


Figure 6.22

"Yeah… no, that evolved quite a bit. Here is a snapshot of the recommendations builder modules." you say.

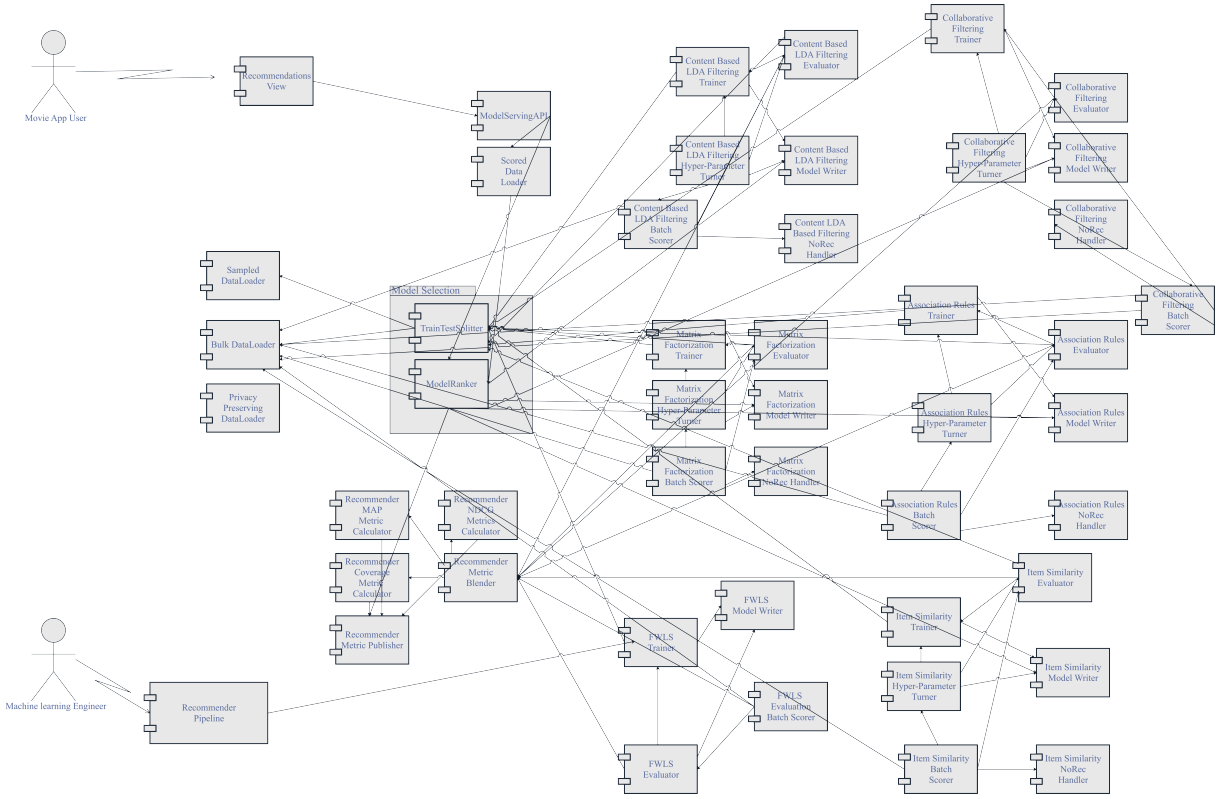


Figure 6.23: Current Big Ball of Mud state of the codebase

"*Ayy, this does not look good, do we absolutely need all this complexity? I thought it was a single linear pipeline... Just how are we going to deal with removing the dependency on the browsing data? The data eng team said they would have it ready for us in 2.5 months, but we promised the client to show them the product 3 weeks from today.*"

"*We can't remove that dependency that easily. We are using FWLS, a Feature Weighted Linear Stacking method. That requires us to train five models and combine them using 25 meta-features. That's how we got that MAP@20 lift that the client liked*"

"*FWLS?! Like the Netflix prize paper we discussed two weeks ago? I remember that list of meta-features, it looked a bit overkill...*"

Table 1: Meta-Features used for Netflix Prize model blending

1	A constant 1 voting feature (this allows the original predictors to be regressed against in addition to their interaction with the voting features)
2	A binary variable indicating whether the user rated more than 3 movies on this particular date
3	The log of the number of times the movie has been rated
4	The log of the number of distinct dates on which a user has rated movies
5	A bayesian estimate of the mean rating of the movie after having subtracted out the user's bayesian-estimated mean
6	The log of the number of user ratings
7	The mean rating of the user, shrunk in a standard bayesian way towards the mean over all users of the simple averages of the users
8	The norm of the SVD factor vector of the user from a 10-factor SVD trained on the residuals of global effects
9	The norm of the SVD factor vector of the movie from a 10-factor SVD trained on the residuals of global effects
10	The log of the sum of the positive correlations of movies the user has already rated with the movie to be predicted
11	The standard deviation of the prediction of a 60-factor ordinal SVD
12	Log of the average number of user ratings for those users who rated the movie
13	The log of the standard deviation of the dates on which the movie was rated. Multiple ratings on the same date are represented multiple times in this calculation
14	The percentage of the correlation sum in feature 10 accounted for by the top 20 percent of the most correlated movies the user has rated.
15	The standard deviation of the date-specific user means from a model which has separate user means (a.k.a biases) for each date
16	The standard deviation of the user ratings
17	The standard deviation of the movie ratings
18	The log of (rating date - first user rating date + 1)
19	The log of the number of user ratings on the date + 1
20	The maximum correlation of the movie with any other movie, regardless of whether the other movies have been rated by the user or not
21	Feature 3 times Feature 6, i.e., the log of the number of user ratings times the log of the number of movie ratings
22	Among pairs of users who rated the movie, the average overlap in the sets of movies the two users rated, where overlap is defined as the percentage of movies in the smaller of the two sets which are also in the larger of the two sets.
23	The percentage of ratings of the movie which were the only rating of the day for the user
24	The (regularized) average number of movie ratings for the movies rated by the user.
25	First, a movie-movie matrix was created with entries containing the probability that the pair of movies was rated on the same day conditional on a user having rated both movies. Then, for each movie, the correlation between this probability vector over all movies and the vector of ratings correlations with all movies was computed

Figure 6.24: Meta-feature from the FWLS paper [15]

"*Yes, it is a lot, but we wanted to squeeze as many gains from the sample dataset the client sent us. We might have gone a little overboard to be ready for the pitch.*"

"*That's a real problem....and there is no way we can fix this mess quickly? Some new framework, maybe? I hear good things about that TensorFlow extended platform, is that something we could use*

to clean this up?"

"*That's more of a deployment tool if I understand correctly. And I am not sure we need more dependencies at the moment. We can look into using it once we get there. The problem is that the ML pipeline's core logic is completely intertwined with the data sources and all the other supporting components. We need at least two weeks to fix this and refactor this mess to remove the missing browsing data dependencies.*"

"*2 weeks!? That would leave us with a single week to run the full end to end acceptance tests with the API! Is there no other way? We are going to lose the client and maybe even our jobs if we miss more deadlines! Are you sure there is no backdoor we could fudge to save what can be saved, and plan the refactoring later?*"

You think deeply about gambling your reputation, but you remember that they depend on you for good estimates. They don't know what the code looks like. They rely on you to give them honest estimates. Pretending that it will take less than two weeks would be madness. "*Yes, two weeks. On the good side, at least we have some characterization tests that we put in place recently. That should help guide our design. But yes, I think two weeks is just enough to tame this beast.*"

"*And you think we can get rid of that data dependency in time?*"

"*Yes, I'll do my best to get it done.*"

"*Well, I mean, we do trust your judgment after all....*"

The product manager looks at his calendar one last time and sends an invite to the tech leads in two weeks to look at the progress.

"*Alright then, I am fine with two weeks. I think I can pull some ropes to get this accepted by sales.*"

"*Sounds good!*" you say. You leave the meeting room feeling good about standing up for the team and the codebase's health. Now, how the hell can this pipeline be refactored in time.

Let's take a look at how the clean architecture concepts we covered can help.

Clean ML Architecture

One of the best things that came out of the clean architecture is the Dependency Rule. Let's use it. We position our modules, so that source dependencies point towards the center of the application. As we move from outer to inner layers, the level of abstraction increases. Towards the inner layers, we deal with the high-level policies that encode the application's value-generating component. As we go towards the outer layers, we deal with the more pragmatic details.
The center layer, in contrast, has the most general components that sit at the highest level of abstraction of the application at hand.

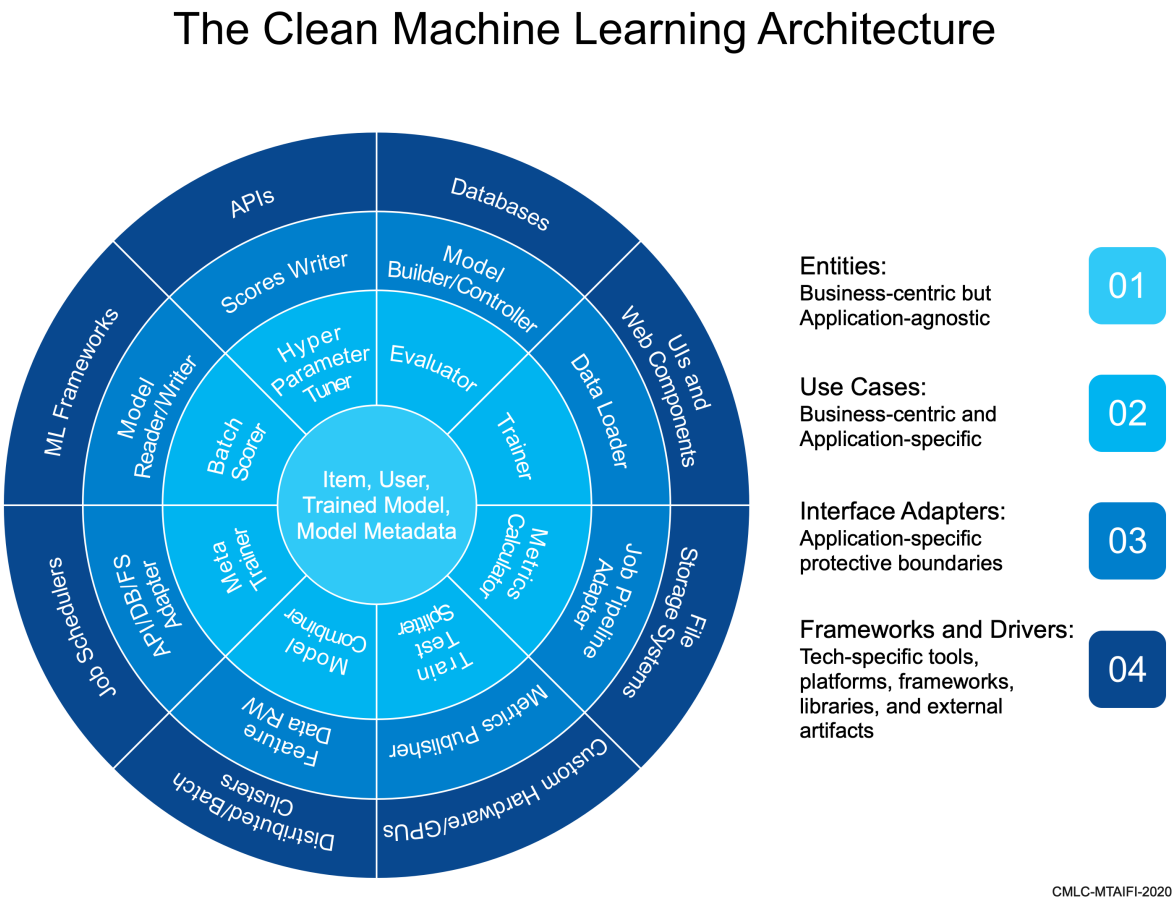


Figure 6.26: Clean ML Architecture

Re-architecting the ML Pipeline

Software architecture is a living creature. Software is less like a building and more like a garden. It needs constant watchful eyes to keep it clean, organized, and maintainable. However, sometimes the team is under pressure to deliver and ends up just piling new code on old code without any concern for handling future changes. And how could they not be? They can't see the future. It seems that to be able to handle uncertainty in software development, the feedback cycle needs to be as short as possible.

The client needs to be able to constantly steer the software they need by trying it out, seeing demos, playing with fake workflows, so that the clients explain to themselves what they want. This constant contact gives the developers just enough information into what is important to the customer, what can change, and the business invariants, if any.

The first step to entangle this architecture is to identify the actors and the use cases.

Use Case Analysis

After examining the interactions of the current components, four actors take center stage. The Single Responsibility Principle tells us that the primary sources of change will come from these four actors. When the system needs new functionality, this will serve one of these actors. The reasonable thing to do is to partition the system according to these actors/roles. The goal is that if we add functionality that serves one actor, the other actors are not impacted.

The use cases we have in our case study above are not the complete set of use cases that we need to examine. For example, we are omitting the model serving component, the data validation, the feature transformations, the metrics publishing, and so on. This is all for managing the size of the problem for the demonstration.

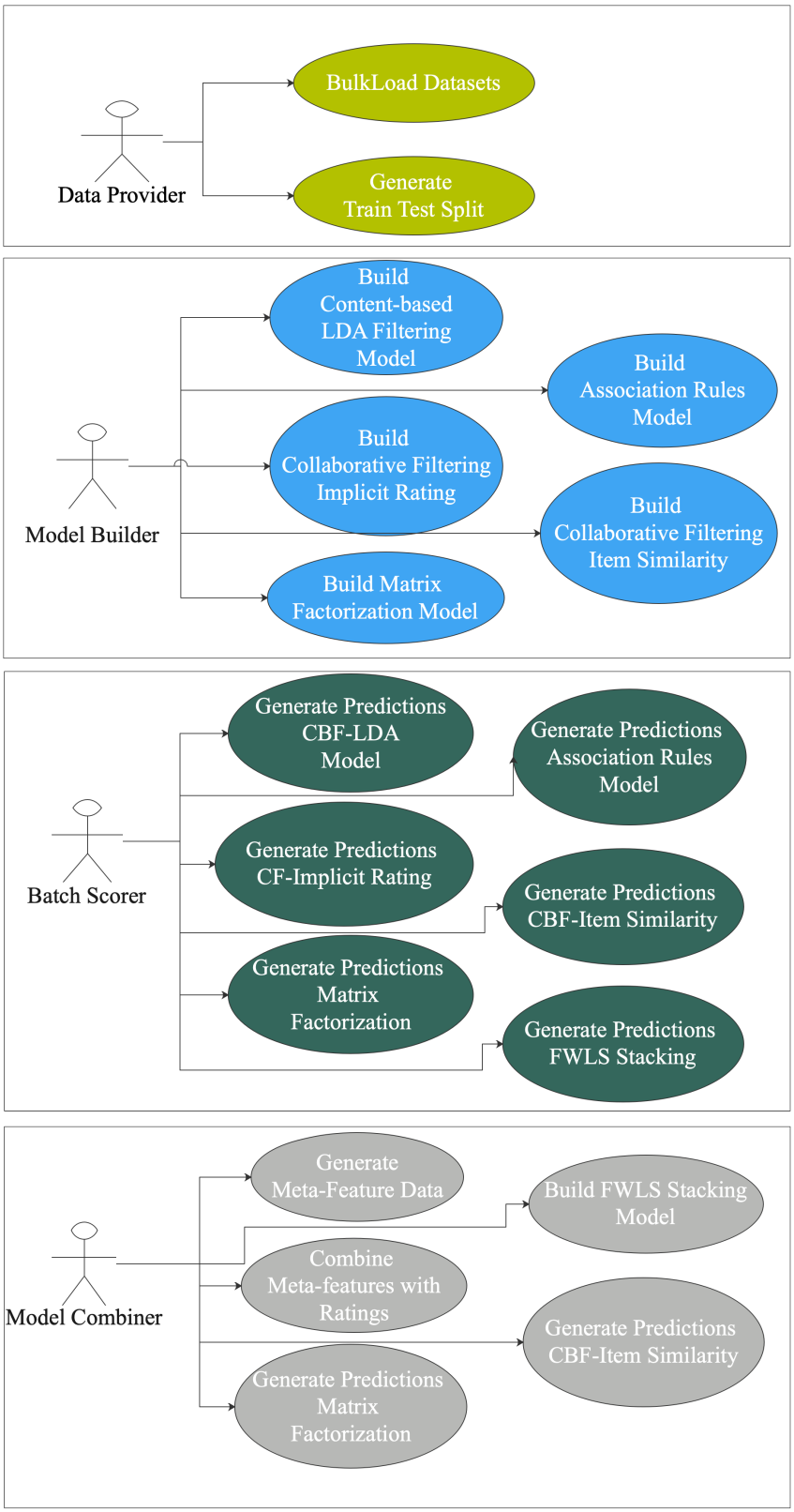


Figure 6.25: ML use cases with actors

The next step is to decide to stick with the clean architecture guidelines. In the figure 6.26, I set up four layers, but it is only one or many possibilities. The recsys in this case-study will probably need more complex configurations than the ones presented here.

Component View

Once we have the data flow, the actors, the use cases, and the goals of clean architecture in mind, we can group the diagram above into components.

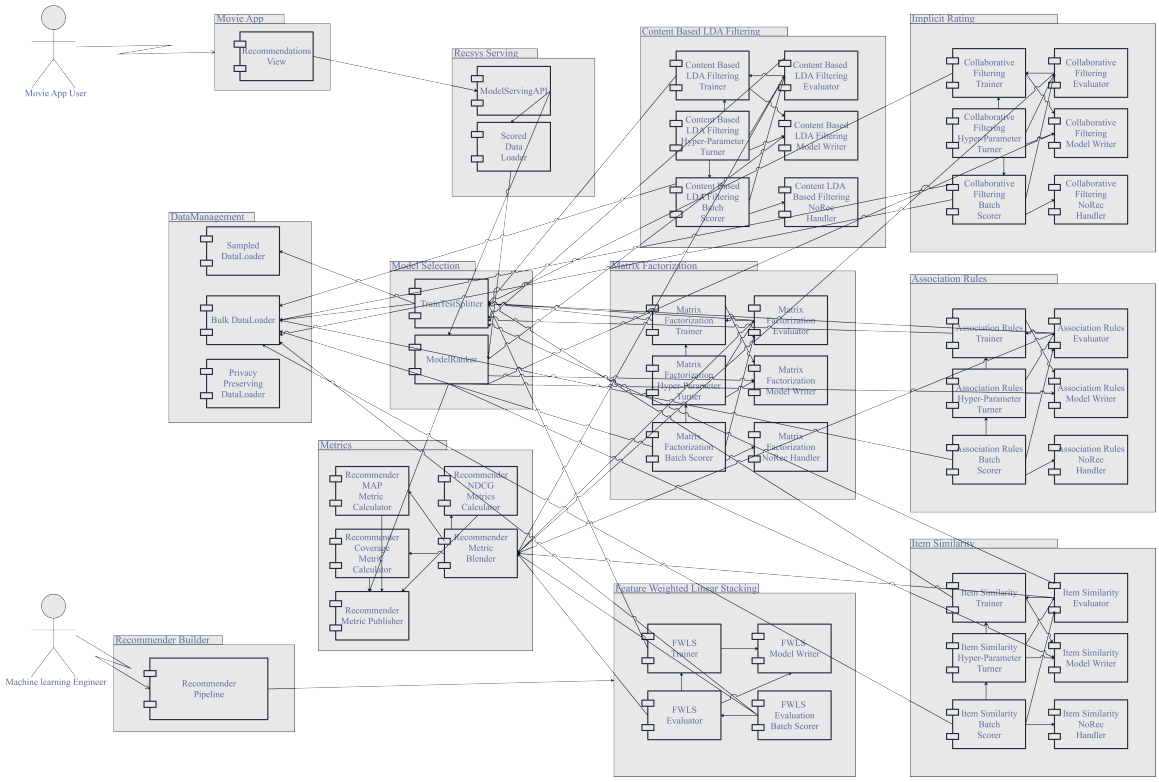


Figure 6.27:Small Ball of Mud Component View

This view starts giving us some idea about the various components that compose the system. Following the dependencies arrows, we see that the ML training components depend on the data loading, metrics, and model selection components. Inside the individual training components, we find that the higher-level module, such as the model trainer, depends on the low-level model writer and batch scorer modules. Also, we notice that the code is ripe with redundant modules.

The current system architecture is very reminiscent of the monolith architecture style, where all the components are bound with each other in a single large application, commonly called “Big Ball of Mud”[16]. By adding the components labels, we end up with a “Small Ball of Mud”[16] because we can at least group modules under common components.

Useful Abstractions

The standard guideline from the clean architecture principles is to push the low-level details to the system's outer-bands. Unfortunately, the publicly available examples about "how" to achieve that goal are mostly web-application-specific. Secondly, similar to other traditional software engineering challenges, there are tradeoffs with each possible architectural style that serves that decoupling. Let's take a couple of strategies to reduce the coupling that exists in this system and learn how to apply common architectural patterns to this problem.

Following the clean architecture pattern, we want to abstract away the low-level details behind clean interfaces. The figure below shows one possible functional decomposition of the system. This figure shows a pseudo-class diagram for one of the model training use cases (matrix factorization in this case). Notice the direction of the arrows. The source code dependencies cross the architectural boundaries in single direction, pointing towards the application. The idea is to drive the low-level implementation details to the periphery of the application. Here we see how a controller/driver depends on the use case, which in turn depends on higher-level abstractions such as the Hyper-Parameter Tuner, the Trainer, and the Evaluator concepts.

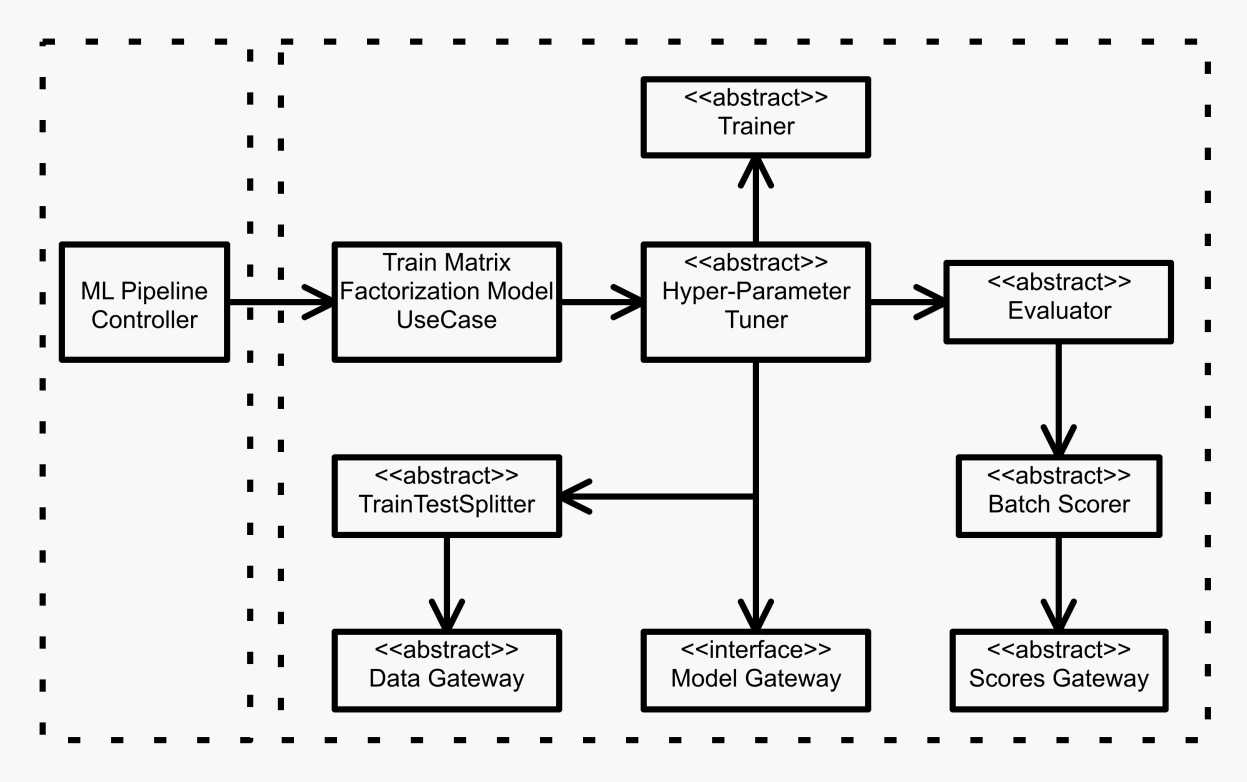


Figure 6.28: Initial ML abstractions

How do we include the concrete training algorithms in this picture? Check the figure below:

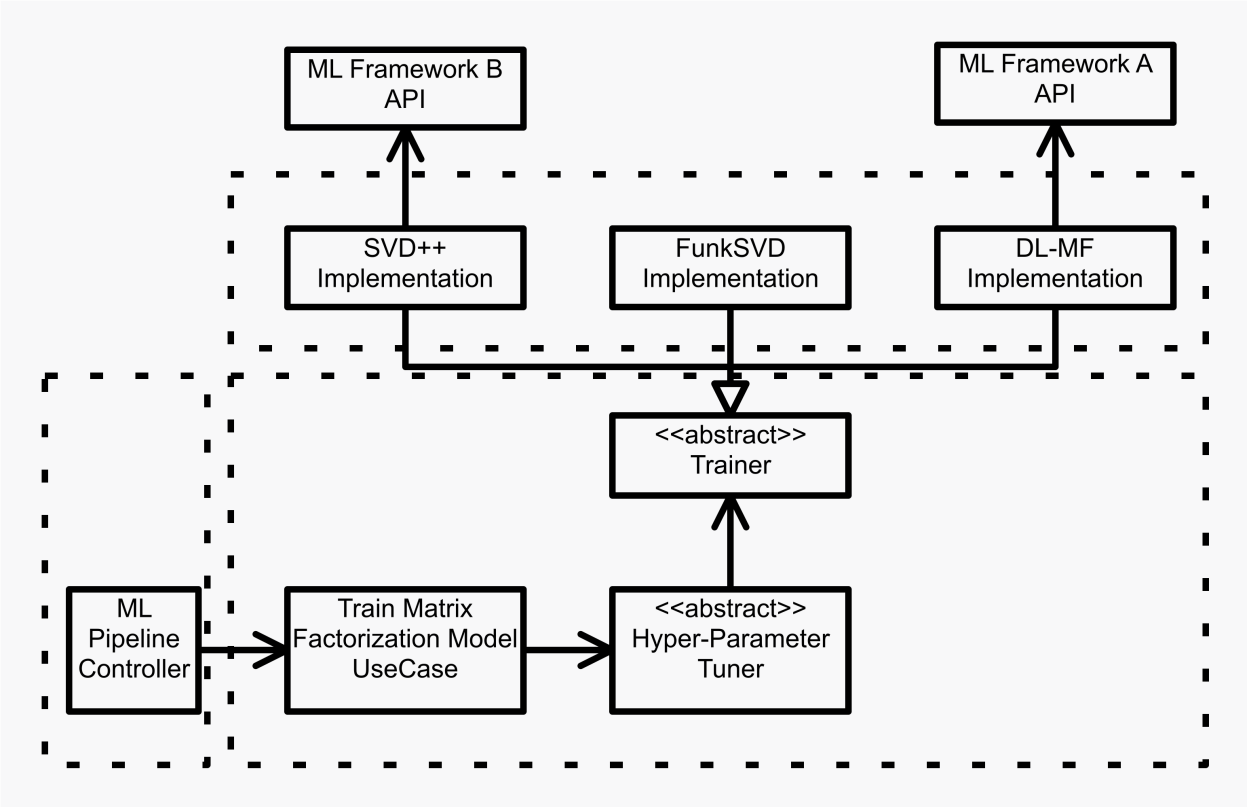


Figure 6.29: Concrete ML trainer strategies

The Trainer interface can be implemented by a select set of matrix factorization algorithms. At runtime, we do not want the use case to depend on the concrete implementations. We don't want the use case to be even aware that there are multiple types of trainers. In this solution, it is the responsibility of the low-level controller to create the concrete trainer instances and pass them to the use case. The use case and the hyper-parameter tuner are only aware of the trainer interface. This forces the low-level details of the training algos classes to respect and implement the standard Trainer interface.

The other use cases such as the collaborative filtering, the association rules, and the content based models can also use this strategy. This helps the use cases and the tuner to be decoupled from the concrete ML algos implementations.

What about the interaction with the outside world? We do need to read/write data, models, and scores, from/to external systems for persistence. How can this conceptual framework deal with that? Check out the next diagram:

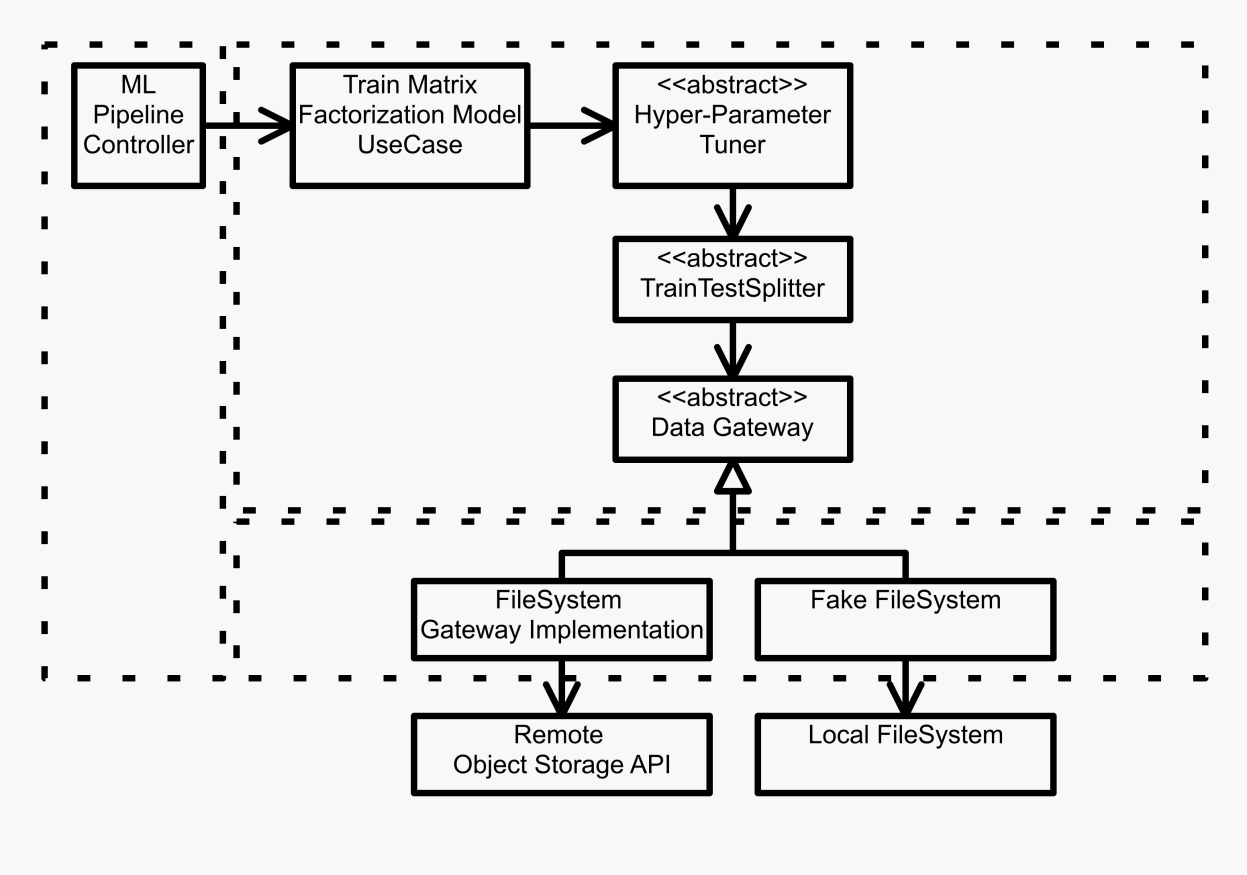


Figure 6.30: Concrete data access strategies

This decoupling is one of the core transformations that the clean architecture promotes. The data gateway acts as a boundary between the core application logic and the external world. The gateway interface can be implemented by multiple concrete strategies.

This has two direct benefits. First, in terms of testing the pipeline, where the use case can be tested with small local files that can fit on a developer's machine. Second, in terms of keeping a plugin architecture, data sources can be added and removed without having to modify the core of the application.

One of the challenges that our use-case/story reported was that we were unable to remove a data source dependency from the code base with such short notice. We could not remove the browsing data source from the ML pipeline without having to apply Shotgun Surgery [17] to the whole application core. With this decoupling, it will be much easier to remove the data dependency if an upstream data engineering team takes too many vacation days two months before the launch of our flagship product. But who can blame them, the weather is beautiful, they work hard the rest of the time, and it was just happenstance that a bunch of crucial senior data engineers were hired at the same time five years ago, and they all became eligible for a sabbatical on the exact same date.

With the initial overly interconnected architecture, we would be completely paralyzed by the unavailability of the browsing data source. However, we can mitigate this by following the

guidelines of the Dependency rule.

Adding the supplementary components, we get the following figure that shows how the other functionalities can be integrated as plugins:

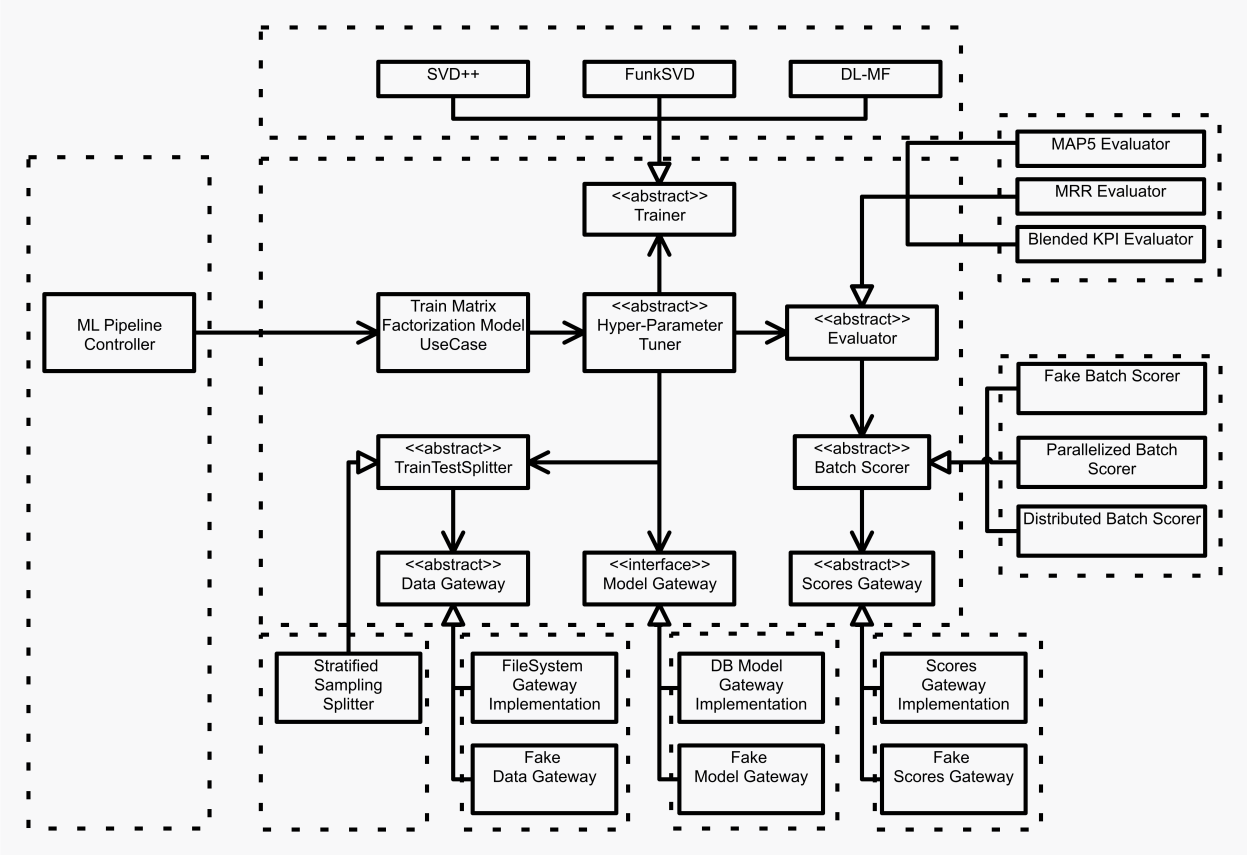


Figure 6.31: Adding the plugins to the core application

Living with a Main

In every ML system, we can always find at a minimum of one component responsible for creating, orchestrating, and monitoring the others. This is usually called the Main component/function.

This component is special because of the level of chaos that is allowed in it. It is the lowest-level policy in the system. It is the entry point of the ML system. None of the other components depend on it. The operating system itself is the only thing that depends on it.

The main component is responsible for creating all the other components, with their use cases, factories, strategies, adapters, proxies, builders, and any other global modules. Then it thanks the audience for coming out tonight, bows down on stage, and hands control over to the other sections of the system with the high-level policies.

To fulfill this role, Main depends on many other components, but no component depends on Main. This makes Main naturally unstable and makes it the filthiest of all the filthy components we have to deal with.

Here is a diagram of the level of dependencies that the ML pipeline main component would have to deal with:

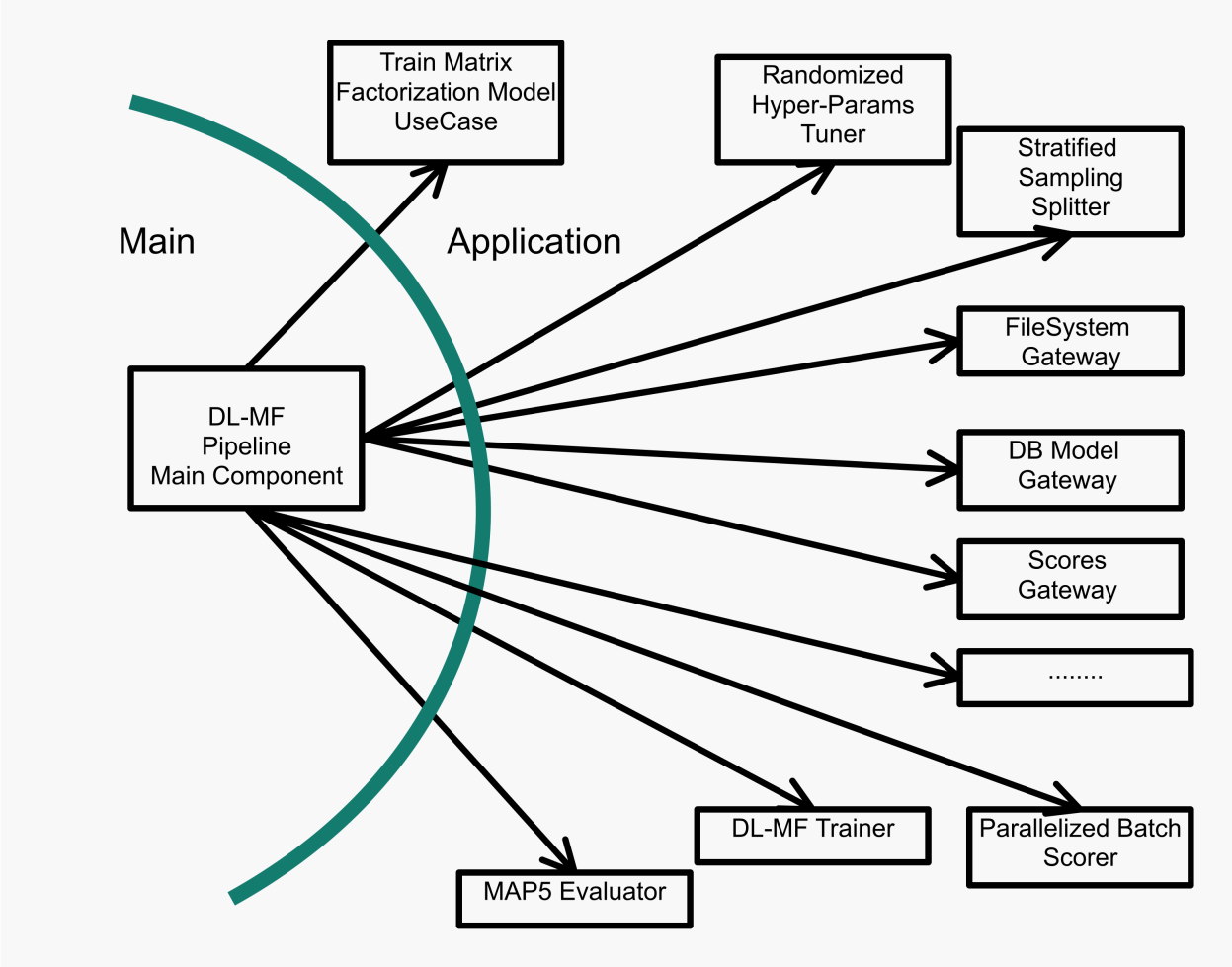


Figure 6.32: Main component connections to the core application

The Main component lives on the outskirts of the clean ML architecture. It bootstraps all the components needed for the high-level system and transfers control to it.

In this architectural framework, Main is a plugin for the rest of the application. It builds and composes the initial state of the application and transfers the control to the high-level policy. This means that it is possible to have multiple plugins that would act differently for development, testing, and production/deployment. This also means that we could have separate plugins for each stage of the product. If there are delays in terms of data sources, we can build a different Main component that would load everything except the components that deal with the missing data source and its related ML components.

Main, as a plugin to the rest of the application, presents the ML development team with a single location for change. All the configuration, wiring, and plumbing can be done here behind an outer architectural boundary. This makes configuration much more flexible and accessible for the team.

Conclusion

Well, this was one hell of a chapter. We started with a story about an ML pipeline gone wrong. We described how unexpected delays can hit ML software projects due to non-ready dependencies. We then dived into the aspects that ML software architecture should focus on, and what it should not focus on initially. We covered a couple of common mistakes that ML teams make by building their application around low-level details, such as the database, the ML frameworks, and the like.

We then explored how to think of ML applications in terms of use cases, and how these use-cases can help ML teams separate high-level policy from low-level details. We investigated the traditional clean architecture and described its major components. We also performed a quick tour of similar conceptual frameworks to help you search for solutions using external resources during your ML journey. We covered the differences between the layered architecture and the concentrically layered architecture. We described the hexagonal, the onion, and the pragmatic architecture ideas that expose similar sets of goals and tradeoffs: **How to manage dependencies without obscuring the business goals of the application**. We then talked about the reason behind our obsession around boundaries and the crucial role they play in building architectures we can reason about.

Finally, we went back to the application of our movie recommender, where we sketched out the general architectural directions that will help deal with the pesky and late data dependencies. We described how abstraction can help reason about complex systems, and we showed how to bound use cases with these abstractions while being very careful about managing the directions of our dependencies. We covered how Main is the one place where messy code can live to isolate the dirty parts from the clean parts of the system.

Hopefully, you found this chapter useful. Obviously, there is a lot more to be said about software architecture. If you are interested in this topic (and you should, for the sake of your sanity), here are some additional references that can help with your journey:

- **Fundamentals of Software Architecture by Mark Richard and Neal Ford**: This book gives a great deep dive into many of the non-functional requirements that are implicitly required for software systems. The micro-kernel architecture they describe in that book is close to the clean ML architecture. However, they give an excellent overview of other methods for partitioning systems when the demands and responsibilities of the systems grow and evolve.
- **Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**: This classic contains a lot more content around how to deal with creating objects, structuring interactions between objects, and how to modify the behavior of objects. This book, also known as the Gang of Four book, needs no introduction. However, if you are looking for a gentler introduction, check **Robert Martin's Design Patterns**

(Clean Coders Video Series) for a fun and engaging way to drive home the usefulness of design patterns in software engineering.

References

[1] Clean Code: A Handbook of Agile Software Craftsmanship. Robert C. Martin. 2008. Prentice Hall PTR, Upper Saddle River, NJ, United States

[2] I think I heard it here: RailsConf 2014 - All the Little Things by Sandi Metz
or here: Sandi Metz - Talk Session: Polly Want a Message

[3] Architecture, Use Cases, and High Level Design, Robert Martin https://cleancoders.com/episode/clean-code-episode-7

[4] Ridiculous Math Problems: http://www.dam.brown.edu/people/mumford/blog/2020/Ridiculous.html
[5] Object Oriented Software Engineering: A Use Case Driven Approach by Ivar Jacobson

[6] Writing Effective Use Cases by Alistair Cockburn

[7] User Stories Applied: For Agile Software Development by Mike Cohn

[8] Use Cases: Patterns and Blueprints by Gunnar Overgaard and Karin Palmkvist

[9] Requirements: The Masterclass By James Robertson / Suzanne Robertson

[10] Clean Architecture Blog Post, Robert Martin, https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[11] Hexagonal architecture, Alistair Cockburn https://alistair.cockburn.us/hexagonal-architecture/

[12] The Onion Architecture, Jeffrey Palermo, https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
[13] Victor Rentea - Evolving a Clean, Pragmatic Architecture - A Software Crafter's Guide
[14] Boundary Value Problem: https://en.wikipedia.org/wiki/Boundary_value_problem

[15] Feature-Weighted Linear Stacking, Joseph Sill, Gabor Takacs, Lester Mackey, and David Lin, https://arxiv.org/pdf/0911.0460.pdf

[16] Fundamentals of Software Architecture by Mark Richard and Neal Ford

[17] RailsConf 2016 - Get a Whiff of This by Sandi Metz

Chapter 7 - Test Driven Machine Learning

Making Your Life Harder in the Short Term

This chapter will probably make your life harder. I don't have much proof, apart from anecdotal, that what I claim in this chapter actually works. The principles and techniques I describe in this chapter will definitely slow you down in the short term. They will probably make you ask for a refund on this book. I am willing to take this risk. Our nascent ML industry needs this.

"How could you possibly say this?! Isn't TDD the holy grail of software engineering?"

It is possibly one of the best techniques we have to build software, yes.

"So why are you saying that it will slow me down? I already expensed this book by telling my boss how good the content is!"

Thank you for the nice words. But some things rub some people the wrong way when talking about testing. Some people even get headaches, yes headaches, when they contemplate writing their tests BEFORE the implementation. It just feels so unnatural at first that the resistance gets so strong that they go and write position posts such as "TDD is dead"[1] to make their point.

"I am pretty new at this, so I don't know the history of this debate. What's the TLDR here?"

I wish I had a concise TLDR summarize the debate. But maybe we can spend a little time getting to know the tradeoffs of Test-Driven Machine Learning (TDML), and we can build a TLDR as we go?

"Sure thing. Let's do it."

60 Minutes to Save Lives

Trigger warning: This section might be too much to handle if you have been involved in a boat or cruise accident. Beware.

Imagine you are in a commercial navy center in 1912, and you get a signal from a boat in distress in the North Atlantic. The Titanic, the largest ship ever built so far, just hit an iceberg and is sinking fast. Excuse the inconsistent time-travel, but your role is to run the current version of the "cruise_survival.py" machine learning pipeline to predict who will most probably drown during this catastrophe. Your results will determine where the crew should go to help as many passengers as possible.

Imagine that the ML pipeline is a standard binary classification task. It takes as input a list of previous passengers in similar cruise disasters. It builds a model of who will survive. Using the current passenger list predicts who is most prone to drowning in this recent disaster.

Your task is to edit the feature engineering pipeline enough to accept the new data passenger list format. It is slightly different from the original data the pipeline was trained on.

We have 30 mins to fix the pipeline to handle the new dataset.

You can find the ML pipeline and the directions for the code changes necessary to save lives here: https://github.com/moutai/tdml-example.

Note that there are two versions. The two versions produce the same output except that one was built with testing in mind, and the second one wasn't.

I would highly encourage you to check out this challenge after you are done with this chapter. **And time yourself** from the moment you are done setting up your development environment to passing the characterization tests with the unseen dataset.

Which version was more comfortable to work with? By how much?

"Why all this urgency?!"

I attempt to compress the psychological pressure that developers feel when faced with changing existing large codebases in a single example. Classic space-time tradeoff. We can't address a large problem-space in one chapter, so we compress time for the demonstration.

The two versions have the same characterization test [2], which should help a bit.

My prediction is that it will take you much longer to work with the code that does not have tests. There are a couple of corner cases and couplings that are hard to disentangle without a test suite's help.

Does ML Code Rot?

Sometimes it feels like ML Code is like peaches. They are sweet and juicy, exciting and plump, soft to the touch, and vibrant to the taste buds. However, if you leave peaches unchecked for 2 days in semi-humid weather, they turn into mush, and no one wants to eat them.

You probably saw ML and DS code rot. Let's take that Jupyter notebook you wrote 6 weeks ago and left unattended. It didn't take long for it to become so big and convoluted that you'd instead start from fresh the analysis that took you 20 hours to do. That code is completely un-reusable. That is one example of code rot. It started with a good design and relatively clean cells and started degrading over time.

You said to yourself, "This time, it will be different!" and with all the greatest of intentions, you met the deadline for the POC ML pipeline feature importance analysis. Now 6 weeks later, you are asked to perform the same analysis but for a different set of features. They are asking us to give them more information on the composition of those embedding layers.

The same notebook was worked on by Andrea and Philip. They did their best to meet the deadlines, which extended the contract with the client. Still, they had to do some severe shotgun surgery[3] to the notebook to get the analysis out in time.

From your perspective, that notebook/script you used is unrecoverable. It became rigid, fragile, and immobile. You give it a chance. You make a copy of that notebook to check, but as soon as you touch one cell/line, the program breaks in multiple places. You attempt to debug it by uncommenting all the commented out debugging print statements that litter the code. You start getting annoyed because your time estimates are now growing, and you have this feeling in the pit of your stomach. Could it be fear? Maybe it is fear? Yes, it definitely feels like fear. Fear of what? Fear that you might get sucked in by the quicksands of the code.

Why do we allow such code rot to persist? We don't clean it because we are afraid.

After opening that script, your first thought was probably "Wow! this is some ugly a** code, I should fix it up". Then the very next thought in your head is "I am not touching it because you know if you touch it you'll break it and if you break it will become yours", [R.C.M Does an immensely better job at explaining this guttural feeling of fear in this video [4] where he describes fearless competence]. So you put your tail between your legs and close that notebook tab. It is usually fear that prevents ML engineers from cleaning rotting code. So you leave it there in its current state as it gradually descends into chaos. You email your product manager and explain to them that it will take another 2 weeks to revive the old code and perform enough surgery and resusitations to get the code back in order.

You end up wanting to fix up the code so bad that you contemplate setting up a concerted effort to do a big clean up. We may set aside a day, a week, or even more to clean the code.

Managers might even end up agreeing with us. In the end, they are hopeless and support the cleaning effort. They also agree that cleaning the code will raise our productivity for future endeavors. Then we can catch up on the roadmap estimates.

The problem is that clean code is a minefield. We can refactor, move code, reorganize the code, but we have no way of knowing that the whole pipeline works. Every time we change something, we need to run the entire dataset through the tests and see if we broke something. We are not even sure that all the branches are exercised. We would have no idea if we introduced a condition that would trigger a side effect somewhere in the code.

Who knows what the impact of re-ordering the features in the dataframes is? Do we have a quick way to check that replacing the static learning rate with a function might break the gradient descent code? What about the evaluation metrics? How do we know that replacing the negative sampling strategy would not harm the rest of the code?

To be sure, we start building an ad hoc, manual, slow test suite. As we change the code, the analytic pipeline breaks in strange ways we can't easily explain. We start digging into the intermediary outputs, and this debugging eats into our estimates. We re-surface from cleaning the code when we run out of time.

Finally, we attempt to run the analysis on 100% of the dataset. Unfortunately, after running for 2

hours, the system crashes with a huge pile of defects that our simple ad-hoc manual testing didn't catch. Some of these defects change the results of the analysis entirely and are easy to spot, regardless of how painful that is. But some of the results are off just a little too much, that your trained eye catches a pattern in the output that you decide to investigate.

You study the list of warnings and failing code paths, but you are left with inexplicable behavior.

After much brooding, sulking, and injunction spewing, we conclude that our saving grace is to put the "cleaned" notebook aside and use the previous working state.

Don't get me wrong, this does not happen every time. Usually, I see developers starting to seek solutions, techniques, and disciplines as soon as we face one instance of these disaster scenarios. Once they are involved in such a chaotic development environment, they get it. They tell themselves, "never attempt a big clean up again.", "If this is how it is done around here, there is no way I can change this codebase.". And that's how you get demoralized devs working on a mess they are too afraid to clean. They collectively raise the delivery estimates, and productivity takes a nosedive.

R.C.M said it best [4]: "We can't clean code until we eliminate the fear of change."

Tests Let You Clean Your Code

Eliminating our fear of change is a fascinating idea. On the other end of the spectrum, away from the disaster scenario, you can imagine a codebase with a suite of tests that is so complete that no bug could hide undetected. This suite of tests would be so quick to run that developers would consider trivial the act of running hundreds of tests in minutes by clicking a single big red button.

Any developer could check out the code, run the tests, and fix/delete/add functionality without fear of breaking some unknown code path that only runs during full moons. What if this test suite could never get out of sync with the system under test. Try to imagine what a suite of tests like these would make to your team's mental well-being and productivity.

"What a waste of time! This kind of test suite is completely infeasible!" is the usual response to such a grand challenge.

Self-testing ML Code

Martin Fowler introduced the idea of self-testing code in 2014 [5]. He describes an attitude that has helped many teams reduce their fear of change. "Our attitude is to assume that any non-trivial code without tests is broken." This quote is worth examining in the light of the self-testing ML code.

You have self-testing ML code when you can be confident that if your tests pass, then the code is free of any significant defects. The idea is that while you write your code, you write a bug detector simultaneously. This detector catches any defect that was accidentally introduced in the codebase. This detector is small and fast and can run multiple times per day. If a bug is found, finding the culprit stands out because the distance between a working state and a deficient state is short.

Did I break something?

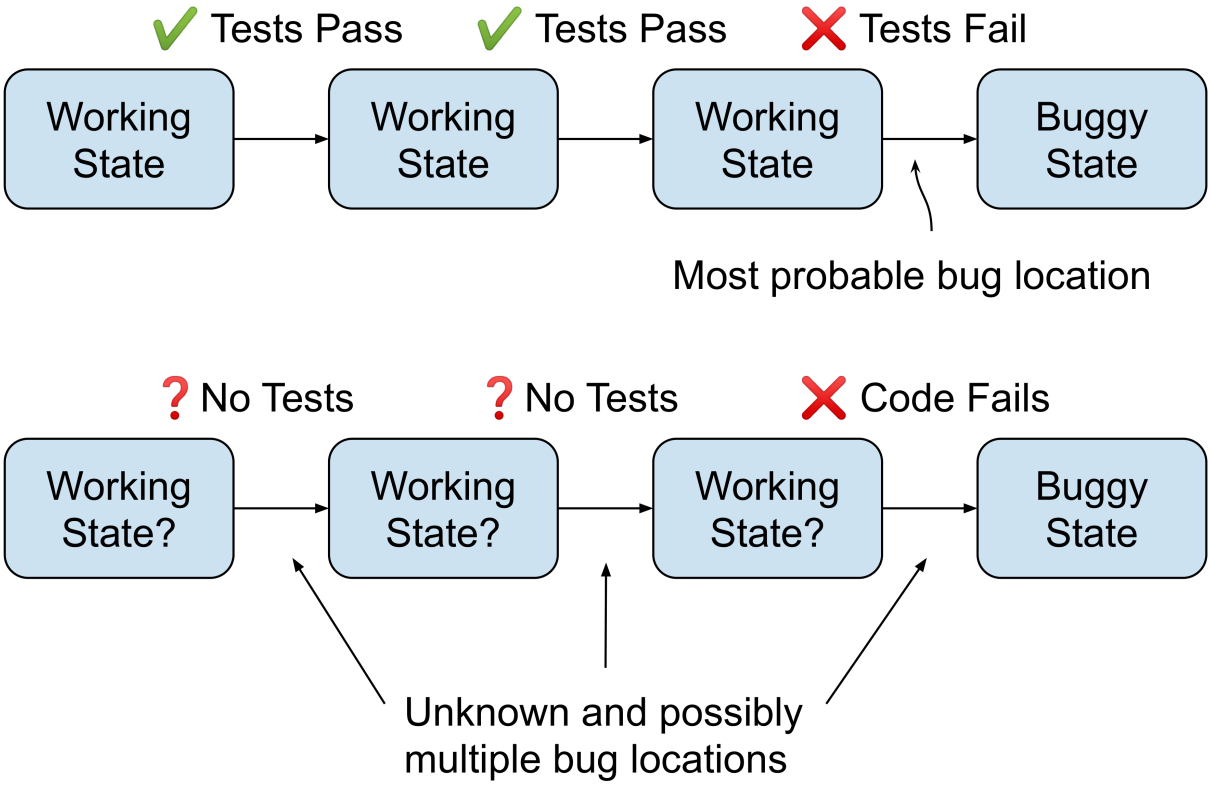


Figure 7.1: Without automated tests between working states it is hard to detect when a bug was introduced.

Developers usually get to know automated testing through the need to reduce production bugs. But from my experience, they stay for the fearless competence and confidence to make changes to the system.

Instead of agonizing over the impact of fixing a bug because you might create more bugs, devs get in the habit of relying on the test suite to guide them in unknown code territories. M.F says it best: **"With that safety net, you can spend time keeping the code in good shape, and end up in a virtuous spiral where you get steadily faster at adding new features."**

One key differentiator for the concept of self-testing code is to remember that: **What is important is not how you got to the tests, but rather that you have tests.** Many disciplines can bring tests to live alongside software projects. Still, sometimes devs have an adverse reaction to automated testing because they conflate self-testing code and test-driven development (TDD).

One way to regularly check on the health of a team's self-testing attitude is to watch ML engineers' reaction to a production bug. **Cultures that promote self-testing code write a test to exercise the buggy code**. On the other hand, teams that prioritize fixing the bug rush towards repairing the bug instance. In a self-testing code culture, the reaction to a production defect is taken as a sign of the failure of the production code and the failure of the tests themselves.

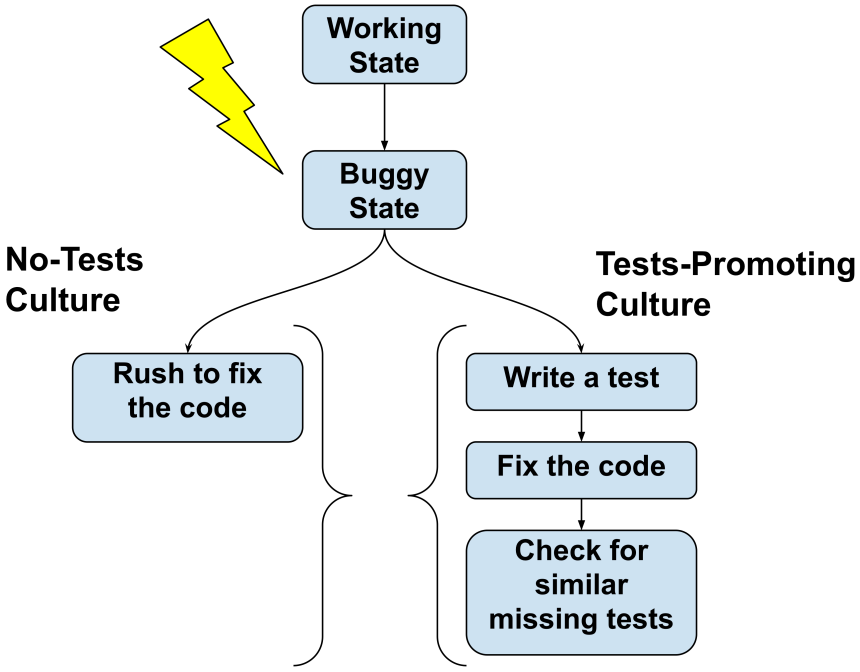


Figure 7.2: The reaction to bugs can tell you if your team values testing or not.

What is this TDD you are talking about?

TDD stands for Test-Driven Development. It is a discipline, attitude, and a technique for developing software guided by tests. Kent Beck came up with this in his excellent book [30].

The TDD steps are:

- Write a test for the next bit of functionality you want to add.
- Write the functional code until the test passes.
- Refactor both new and old code to make it well structured.

TDD is more than a technique. It is a discipline. Being a discipline, it has a set of rules. My favorite rendition of the TDD flow is described by R.C.M. [6] in his Three Laws of TDD:

The Three Laws of TDD:

- First Law: You may not write production code until you have written a failing unit test.
- Second Law: You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
- Third Law: You may not write more production code than is sufficient to pass the currently failing test.

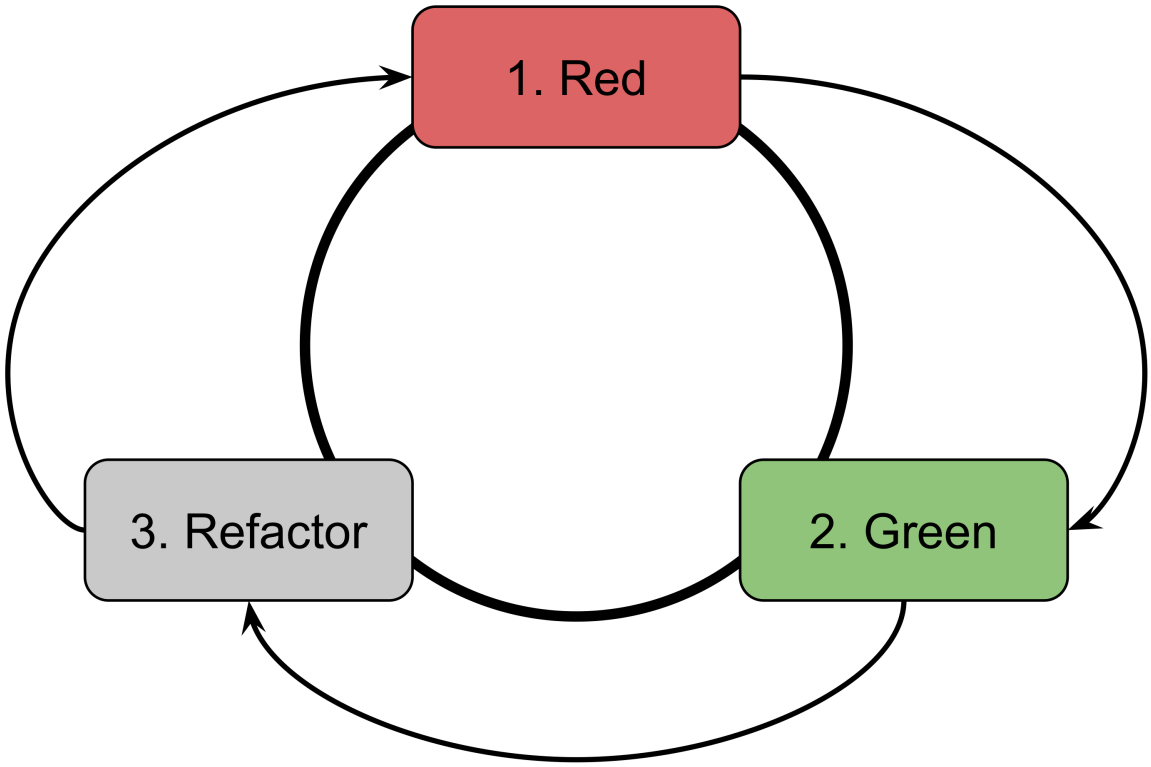


Figure 7.3: The classic red-green-refactor development cycle.

The basic idea is that you get into a development cycle that repeats these three steps, and the system grows over time.

When I first encountered this discipline, it seemed utterly counter-intuitive. Writing tests before the functionality? What a crazy idea.

Unfortunately, I tried and failed many times in implementing this level of discipline in ML/DS code. However, after each failure, I started noticing three main benefits of following these rules:

- First, the more obvious one is that I ended up with more self-testing code, with a growing list of tests ready to be run on every change.
- Second, a more subtle effect is that writing tests first forces you to examine the interfaces of the code. This helps in separating what needs to be done from how to do it. It forces you to think from the user's perspective rather than from the perspective of the builder.
- Third, when applying TDD, I feel a lot more confident in changing the code's organization through small refactorings.

Which ML Code Tests Do You Need?

Once you start working with tests, you notice that they start having different responsibilities. One of the most popular ways to examine your set of tests is through the Testing Pyramid metaphor [7].

The central argument of the Testing Pyramid is that you should have few high-level tests running through the UI of the application, but many more low level unit tests.

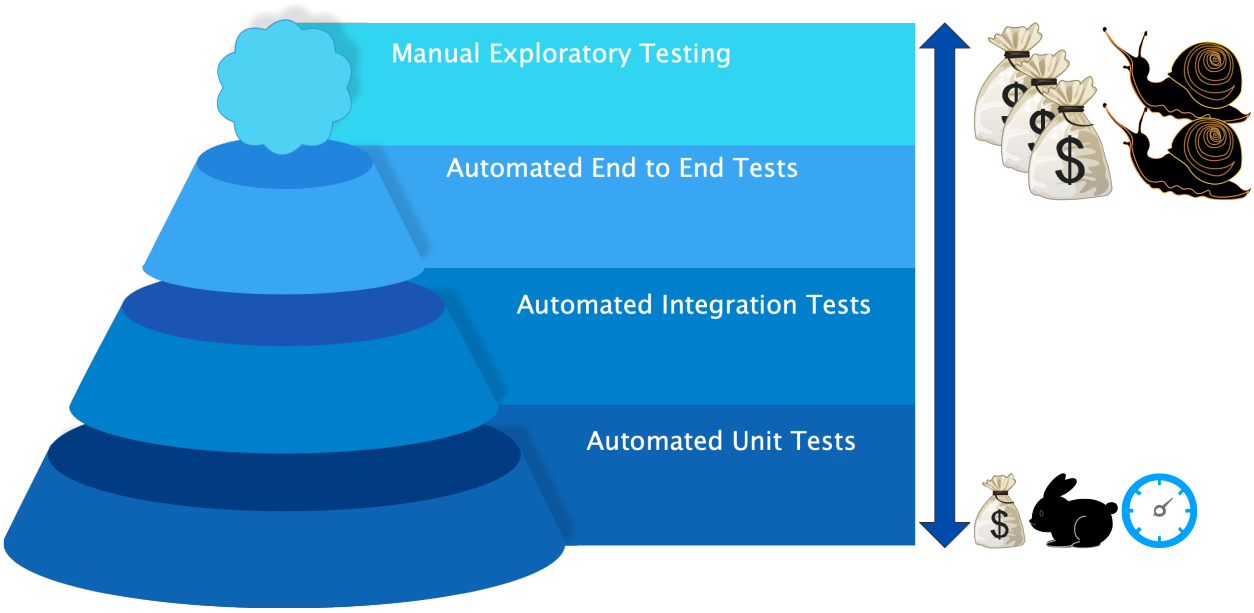


Figure 7.4: Idealized Testing Pyramid

A worrying trend is that much of the training of data scientists and ML engineers in the industry, as of 2020, is performed using throwaway scripts and experimental notebooks. This means that the "testing" is done traditionally by running the whole ML/DS application end-to-end with the full dataset. This is equivalent to testing a web application through its UI by clicking on every possible button every time. This method is the most basic testing method, and if it works for you, please carry on and come back once it stops working on your project.

Yes, yes, I know, I know, I understand that you reduce the data size to speed up the testing. And I know that you restart the kernel and rerun all the cells in your notebook to get a fresh state. If that is what gets the job done, carry on with it, there is nothing to see here.

However, similar to what happened in the world of non-ML applications, this approach degrades quickly. It drags the testing pyramid to become an ice-cream cone [8].

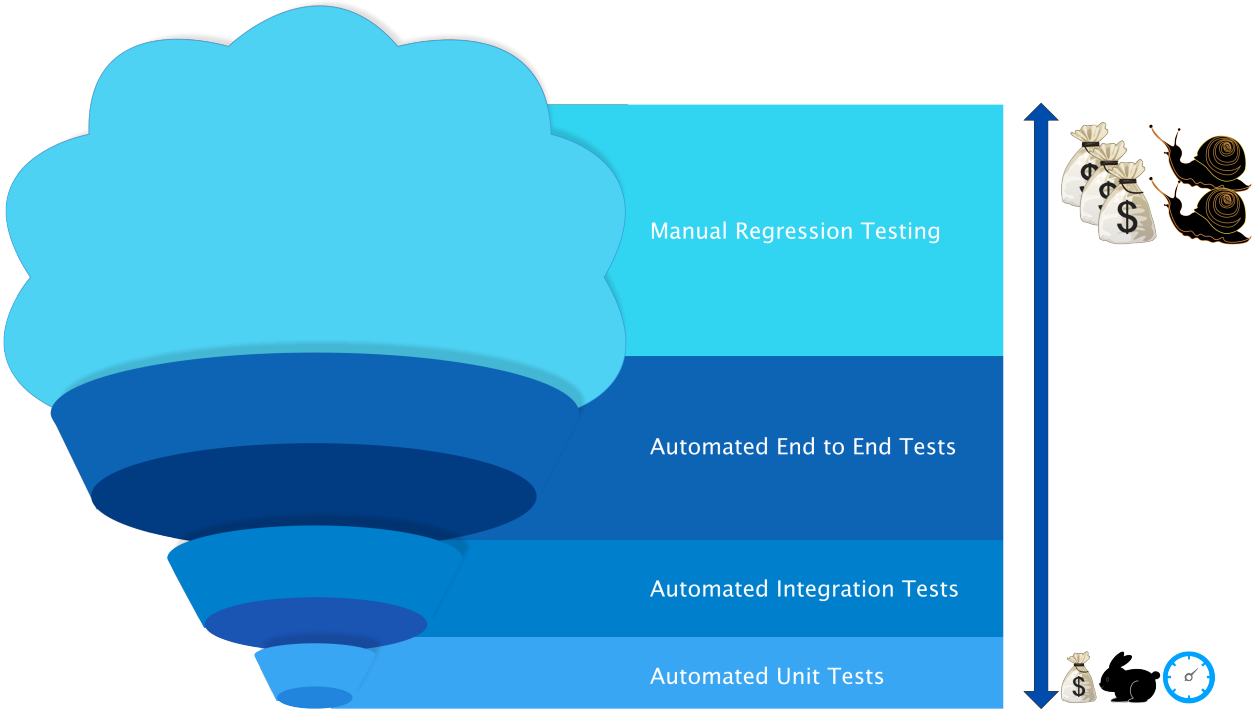


Figure 7.5: The Unfortunate Testing Ice Cream Cone

Testing through the "UI" of the ML applications is equivalent to testing the end-to-end ML pipeline. Similarly, it is slow, fragile, and costly. It requires accessing various external datasets. Munging and re-munging the full dataset every time the tests run. Possibly even calling external databases and APIs that might be down for maintenance at any point in time.

Usually, these e2e "UI" tests need expensive machines to run on where there is enough memory for the whole dataset, and possibly even a couple of GPUs to run the tests. Not to mention that they might need a fully distributed computing cluster. There a distributed job, with startup times in the minutes and runtimes in the tens of minutes, might need to be triggered every time we change a couple of lines.

From experience, watching a resource manager telling me that a testing job is fourth in line behind other jobs is pure joy <sarcasm>. Time to check my Slack messages and read some email, only to find out, 30 mins later, that I forgot to handle escaped double quotes in the string parsing code. This is probably fine if you are fixing a single bug, but you must agree that it is not healthy in the long term.

Above all, these semi-manual tests end up being very fragile. A single change to a feature engineering function can break the whole end-to-end test. Not the least, e2e tests suffer from extensive **non-determinism** problems that make successes and failures alike suspicious to the observer. Basically, end-to-end tests performed from the "UI" of ML applications are fragile, hard to write right, and expensive to run. The pyramid metaphor encourages developers to shift most of their automated testing to the broad base of unit-tests.

The pyramid metaphor relies on the assumption that your e2e tests are "sloths" riding a Tesla car without insurance: They are slow, fragile, and expensive. If your scripts and notebooks run in less than a minute, have less than 25 lines of code, and cannot really damage your project, please ignore most of what I mention in this section. You can come back later once you work on more business-critical stuff. Or is it that business doesn't trust your output because they don't trust the code's quality? First, I encourage you to work on the more critical business stuff. Second, read-on to get an idea of what to do once you get there.

GridSearch for ML Code Tests

Unit Tests

What Is A Unit?

The concept of "unit tests" is pretty vague. Try asking three of your colleagues what a unit test is, and you surely will get four different answers.

Regardless of the difference, you most probably will get answers with some common concepts in them. They will probably begin by telling you that unit-tests are low-level. They focus on relatively small chunks of the system. Second, you'll notice that they mention that the devs are themselves responsible, not some distant QA team, for writing these tests with the help of some library for assertions, fixtures, test discovery, and the like. Third, they will probably emphasize the high speed at which these tests run and the fast feedback they provide.

On the other side of the fence, we have the definition of a "unit." This not to be confused with the r/absoluteunits subreddit with the classic gems: "Oh Lawd They Comin." We might consider a class a unit, while a function could be a unit in other cases. This is highly contextual. In the end, it is the developer's responsibility to determine the size of the unit to be tested, given their understanding of the system.

In the end, they should be fast, isolated as much as possible, require the least amount of setup, and give enough feedback about the system as possible. This could be a single function that does a single feature transformation. It could be a method that calculates some evaluation metrics. But it could even be a whole class that builds a multi-layer model and tests that the inputs and outputs' shapes align with the expected dimensions of the ML data.

To Call Or Not To Call, That Is The Question

An interesting distinction for unit tests is the **Solitary vs. Sociable** preferences [9], popularized in [10]. Take an ML Model Evaluator as an example. Imagine you are testing a custom F1 metric. The calculation method needs to call some functions on the Model and Metric Publisher classes. Devs that like their unit tests **solitary** will do what they can to avoid using the real Model and Metric

Publisher. This is because if there is a bug in the Model class, then the tests of the ML Model Evaluator will fail. The devs will remove this unnecessary coupling using TestDoubles as replacements for the unit's collaborators under test.

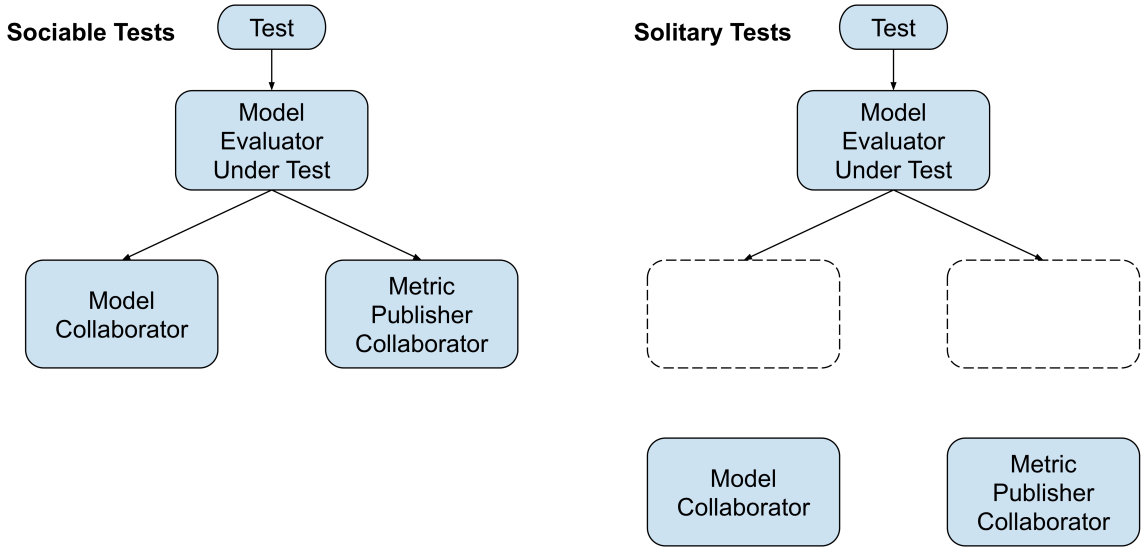


Figure 7.6: Sociable tests use their collaborates while Solitary tests isolate themselves with Test Doubles

However, you might find that not all your teammates like these solitary unit tests. They might say that if communicating with a collaborator is not too much work, like loading a tiny dataset from disk or creating a local collaborator in memory, then it is fine to allow the tests to be **sociable**.

This distinction is sometimes at the root of the confusion around unit tests. Even sociable tests can be considered "unit tests" as long as the tests are written, assuming all the collaborators are working correctly.

You can detect this distinction when you encounter systems that make use of Mocking libraries. Fully air-gapped unit tests are invaluable when dealing with non-determinism, slow collaborators, and any external uncertainty that we genuinely don't control.

Being aware of the distinction is what matters here. Projects can benefit from either method without going all-in on a single absolute rule. If the collaborators are stable and fast enough for the system you are working on, go for the friendly tests. On the other hand, if you need to control flaky external resources' behavior, use solitary tests.

The Need For Speed

One thing that people agree on is how fast unit tests should be. Just kidding, even that is open for debate. But at least there are some guidelines on how long a unit test should take.

Unit tests are meant to be run very frequently when developing the system—every 1 to 5 lines of code. Several times per minute, anytime, a meaningful code block is changed, added, or removed.

The unit tests are there to give feedback to the developer as quickly and as frequently as possible. The idea is that if the ML dev breaks the feature transformations that bucket the ages because of a misunderstanding of the input features, we want to know as quickly as possible. It is much easier to detect a bug in 1 to 5 lines of new code than digging through the 100 lines of code we added in the past hour. Hearing things like "I guess I just have to comment out everything I added in this feature transformation and uncomment one line at a time until I find the problem" is not uncommon when dealing with complex ML pipeline logic with the help of unit tests.

As we add more unit tests to the system and run these tests very frequently, it is ok to run a subset of tests. We might want to run the tests related to the piece of code we are dealing with at the moment. A good technique popularized here [10] is differentiating between compile suite and commit suite. The compile suite trades off the depth of the testing for the speed of testing. It is a subset of tests, and they run fast to give the dev feedback as quickly as mentally-processable as possible.

On the other hand, we have the commit suite, which contains all the unit tests and a few slower system tests. In a team with many devs, each team member should prioritize running the full commit suite before committing – get it? Committing – any code to remote repos. Regardless of which suite you are running next, one thing is constant, the faster the suites, the more frequently the devs will be encouraged to run them to check their work.

But what is this speed we are speaking of? There is a sense that there is an upper bound of a few seconds for the smaller, more targeted suite of tests and a few minutes for a full suite of tests. Here are my heuristics: The smaller suite of tests should probably take less than the time it takes you to take a sip of water or tea. The more extensive suite of tests should take less than the amount of time it takes you to go to the kitchen to get a small snack.

The critical message here is put best by Martin Fowler [10]: "The real point is that your test suites should run fast enough that you're not discouraged from running them frequently enough. And frequently enough is so that when they detect a bug, there's a sufficiently small amount of work to look through that you can find it quickly."

Integration Tests

Integration tests are built to ensure that independently developed units of software can collaborate with each other. There are many definitions for this kind of test. A primary dimension of confusion is the scope of integration tests. When tasked with testing the integration of their systems, many developers go for "broad scopes." In contrast, integration tests can just as well work in "narrow scopes" [31].

After running the individual unit tests, the time comes to combine the individual modules into a full system or at least into a set of meaningful sub-systems. The goal of the integration exercise centers around checking that module built by different developers at different speeds, and different times, work together as expected. But this creates problems for the developers.

Unfortunately, this "broad" perspective merges two responsibilities. First, it attempts to test that

multiple independently developed modules can collaborate. Second, it attempts to test that a system composed of multiple modules works.

It is useful to be aware of these two separate goals to be able to disentangle them. Ultimately, how will we test that a data loader module and a model builder module work together without running both of them into a single system and watching them interact under a set of tests?

It took some time for the software industry to realize that the "broad" tests do not have to be the daunting tests that they used to be. To remediate this issue the industry was introduced to the idea of **Tests Doubles**.

The model builder's integration can be done by running the portion of the builder that interacts with the data loader module, using a test double. Using this test double, we can check the correctness of the collaboration of the builder without having a complete data loader instance. This might sound overwork for two simple Python classes that live inside a monolith. Still, if the data loader interacts with an out-of-process service across a network boundary, it would be a lot of work to get that data loader service to initialize. We would need to involve all the infrastructure, tooling, and internal deployment workflows that the data loader needs to start. Keeping the Test Double that can play the data loader's role in-process can reduce the cost and simplify running the "narrow" integration tests.

Keeping the test doubles in sync with the real thing is the subject of a subsequent section. Still, the current functionality synchronization is done mainly through contract tests.

The level of complexity drops when teams use these guidelines in their integration tests. The "narrow" integration tests combined with the contract tests let the ML devs test their integration without ever interacting with a possibly bulky and expensive instance of that service.

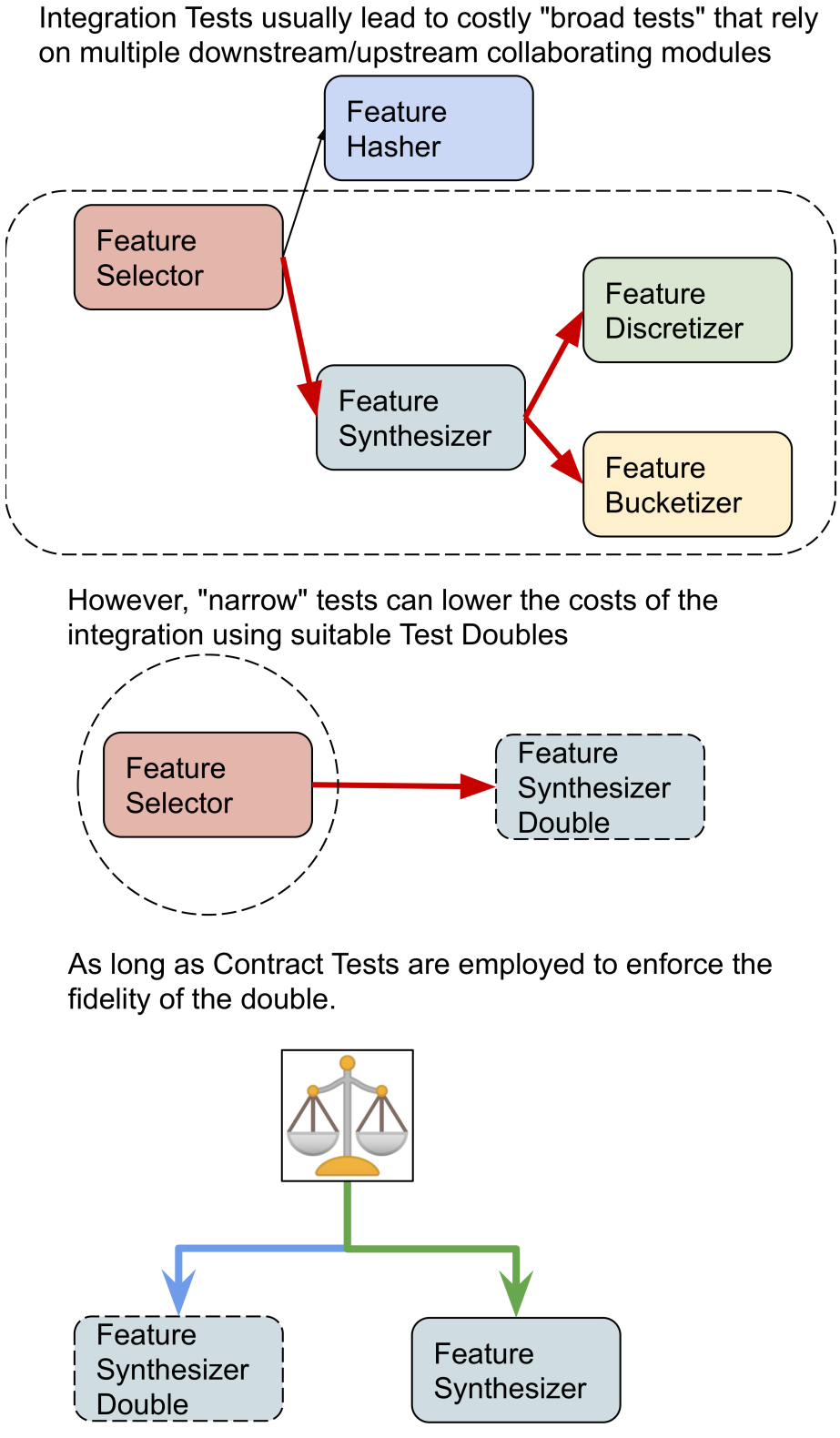


Figure 7.7: Integration Tests

The next time you attempt to write integration tests, ask yourself if you need a broad or narrow test. Suppose you can run a narrow integration test of the evaluation metrics calculator using a test-double instead of a remote black-box optimization hyperparameter tuner service. In that case, that will simplify your development. After fixing all the narrow bugs, you can cover the rest of the cases using a broad integration test by requiring a live version of the tuner service and exercising all the code paths in the tuner service.

One last hidden benefit of narrow tests that ML devs don't realize initially is how these sorts of tests can speed up the early stages of continuous delivery pipelines. By early stopping the testing pipeline before reaching the more expensive broad tests, we could save us some time and money by not running costly end-to-end tests that might take a long time to converge.

If you are working in a monolithic ML application, and you know everyone involved in the development, then you can probably skip this section.

If you are working in a neo micro-services world, or your application is dependent on a couple of other team's services, or your module is a dependency of the production app, then read on.

The reality is that as soon as you start depending on external services, you have to contend with slow and unreliable services running on slow and unreliable network paths. How can your testing escape that slowness? As we saw previously, test doubles are handy in this case. However, **how can you keep your test doubles as good representatives of the actual real external service? What can you do when the external service changes its contract?** You know things like those deprecation notices from critical data sources, dates changing formats, IDs being passed as strings instead of numerics, floats changing their precisions, and what are all these fields I am receiving and not using? Should I worry about that too?

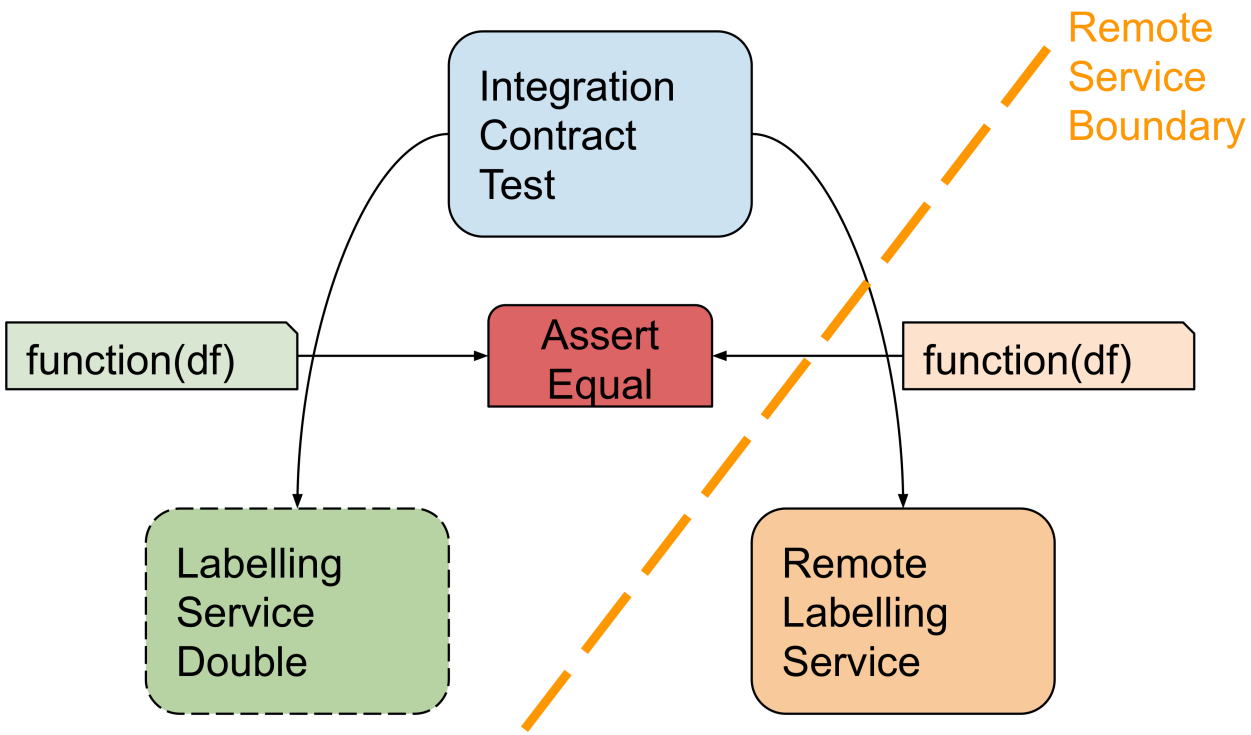


Figure 7.8: Contract testing in a nutshell

The software industry is still solving this problem, circa 2020, but the current guidance is to work at two different scales. First, you run your tests against the doubles to unblock your development so you can discover the requirements of the service you are building. Then you run periodically a different set of **contract tests** [12] that call the external service. You use that to compare the results to your test doubles. If they fail, you have to modify your test double and code to match the external service's new version.

Two parallel rhythms need to be taken into account here. There is your internal application's change rhythm, which you partially control. And there is the external service's rhythm that might change faster than your application. The second rhythm is outside of your control. Still, a daily poll of the external service is usually enough to inform you about breaking changes in the external service.

As chaotic as software teams go, external APIs change in unpredictable ways. However, at the minimum, versioning has been taken seriously by most teams at this point. The versioning allows the contract tests to fail without blocking the build and inform the application owner that something changed. "The data API v2.x has been deprecated. The new data_api v3.4 has been released. You are encouraged to upgrade to the v3.x releases before the end of life of the data API v2.x which will occur on 2020-08-31". These automated checks will trigger a couple things. First, a new task will be created to bring the test doubles and relevant code in sync with the application. Second, it will start a conversation with the owners of the data_api about the impact of this change.

If this API version upgrade falls through the cracks, then unexpected breaches of contract will

probably escalate to a production incident where we served the wrong recommendations to the front end team for a couple of hours. No wonder the forecasting pipeline was triggering false-positive ratios alerts last night.

A useful technique for dealing with the above is consumer driven contracts where you share your contract tests with the provider of the service.

The external service owner can then run the contract tests to check that they can release their next version without breaking your recommendation service and probably many other related services. Go here if you want to learn more about contract testing:

- Martin Fowler's Gems:
  * Consumer-driven Contracts: https://martinfowler.com/articles/consumerDrivenContracts.html
  * Self-initializing Fakes: https://martinfowler.com/bliki/ContractTest.html
- An example framework that specializes in contract tests: https://docs.pact.io/

Component Tests

Component Tests [11] are tests that target a portion of the system. They are slightly smaller than end-to-end tests. A Component test might include the FeatureExtractor+DataLoader+DataAugmenter or ModelTrainer+ModelEvaluator+Metrics Publisher. It is smaller than the whole ML pipeline, which has seven or who knows how many modules.

The key idea here is that we are deliberately removing parts of the system.

These tests are more manageable than e2e tests, run faster than the whole pipeline, and speed up the early stopping of a continuous deployment pipeline before the full e2e tests are kicked off.

This is, once again, a classic Test Pyramid benefit. A large number of faster component tests can clear the buggy code paths and reduce the number of expensive e2e tests needed.

End-to-End Tests

End to end tests, a.k.a. e2e, are there to help you make use of the majority of the parts of your application. It stands in contrast to the component tests, which selectively test sub-groups of modules of the overall application.

End to end ML tests often take the perspective of the user by exercising the application from the UI. This means, for example, exercising, the training, model publishing, and model serving components in a single orchestrated test.

However, e2e can be expensive, slow, flaky, and hard to maintain. The Test Pyramid puts the ends to end tests at the top of the pyramid, where few of these tests should live. To be effective with e2e tests, the testing community encourages the devs to reduce the number of e2e tests and make use of

faster and smaller tests that can give feedback a lot faster. Instead of wasting a full training job just to find that there is a bug in the early stopping code when checkpointing is enabled.

End-to-end tests have the precious advantage of checking the ML application with all its modules wired together. Thus, it helps find bugs that lurk in the interaction between components, which are out of reach for component tests. But e2e tests costs are not insignificant since the tests rely on loading, transforming, and training on large portions, if not all, of the datasets available to train/validate/test/predict on with the additional costs of launching custom hardware, GPU accelerators, or distributed computing jobs to reach a solid confidence level in the quality of the code. This can get expensive and resource-intensive very fast.

** For e2e tests, the guidance is to use the Testing Pyramid as a guide to push as many tests down the pyramid towards the faster, cheaper, and quick feedback tests we described above.**

Threshold Tests

Threshold tests monitor a measurable metric and compare it to a known value. If the threshold is crossed, the tests fail, the build is halted, and the deployment pipeline is stopped.

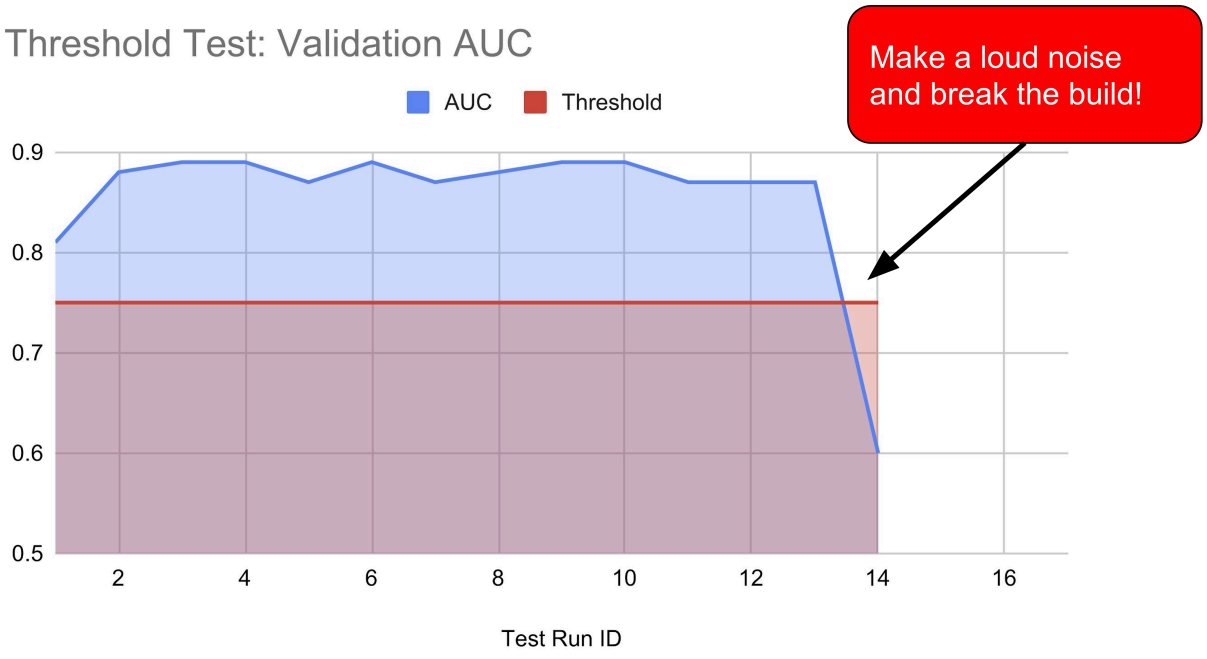


Figure 7.9: Threshold test example

Most commonly, in ML applications, threshold tests target performance metrics. This includes model predictive performance metrics such as accuracy, precision, recall, and the like. If the model precision drops below an acceptable value, fail the tests.

Another use of threshold tests is related to time-based metrics. Did the queries take too long to run? For data loading and writing, poorly written queries can be the culprit. Did the model converge within the allotted time-bound? Maybe someone changed the default learning rate, making it too small for the model to converge in the time allowed.

Threshold tests are the bread and butter of ML applications due to the predictive inputs and outputs' probabilistic nature. This includes:

- Validating data
  - Does data meet the expected ranges?
  - Does data contain the expected values?
- Validating model quality
  - Does the model reach the required performance?
- Validating the inference performance
  - Does the model return predictions under an acceptable time bound?
- Validating model bias and fairness
  - Is the distribution of the sensitive attributes data balanced?
  - Are the fairness criteria of the model predictions met?

Having threshold tests can bring some sanity and process to debugging ML model behavior. By having these automated tests, you can spot the exact set of commits that caused the breach. This is priceless when searching for the reason why our model is misbehaving in production. If the threshold tests are not run frequently, there is a natural accumulation of new code commits that are harder to disentangle. Was it the new SQL filtering that messed the predictions data ranges, or was it the change in the way we search for optimal embedding sizes?

Another good natural consequence of this type of test is using them for **ratcheting** [13]. As the modeling pipeline performance/quality improves, you can adjust the threshold to tighter bounds. After each commit that raises the test F1 score by 0.06 to 0.87, you can increase the threshold to be 0.85 from the previous 0.80 targets. This is incredibly helpful if you start a refactoring effort that might also be touching behavior. It is up to the team to relax the threshold as they encounter various obstacles during the refactoring. The threshold tests acts as a reminder that once the system is functional again we can raise the threshold once again to its previous recorded value to continue the code improvements using a deliberate and previously-achieved model quality goal.

Regression Tests

Imagine we have a deployed ML pipeline that does sentiment analysis as a step before spam detection. We get a bug report that the pipeline breaks because we don't handle greek characters well, such as λαγουδάκια είναι υπέροχα. The feature transformation throws a warning and outputs default values, which hurts the model performance.

To build regression tests, your reaction to this should be to first write a set of tests that expose the problem and then write the code that will handle these characters.

The next week, you get a new bug report that Arabic characters are not supported, such as ▯▯▯▯▯▯“ .”▯▯▯▯ You write the failing tests, you write the code to handle the Arabic characters, and you run the full regression test suite. Boom! The greek characters’ regression tests blow up, pointing out that our current solution for handling both cases is in conflict. That’s where a good regression test suite helps catch bugs that are accidentally introduced and ensure that previously fixed bugs continue to be dead. This is priceless in a team setting. Various developers might not know about the corner cases of a specific feature transformation.

A good regression test suite is one of the best protective measures against the random resurfacing of old problems.

Test Implementation techniques

Test Doubles

One of the confusing parts that I noticed myself and new machine learning engineers struggle with is the inconsistent naming for the test doubles. The initial interaction with testing workflows leads many folks towards learning about excessive usage of Mocks. Unfortunately, this is an incomplete picture that drives many ML practitioners to abandon unit testing because of pure confusion.

In this excellent book about Testing Patterns [14], we can find a fine-grained naming and definition for the generic term Test Double. In general, when we replace some object, function, or other components to enable testing that thing that is used is called a Test Double. However, test doubles can take many forms:

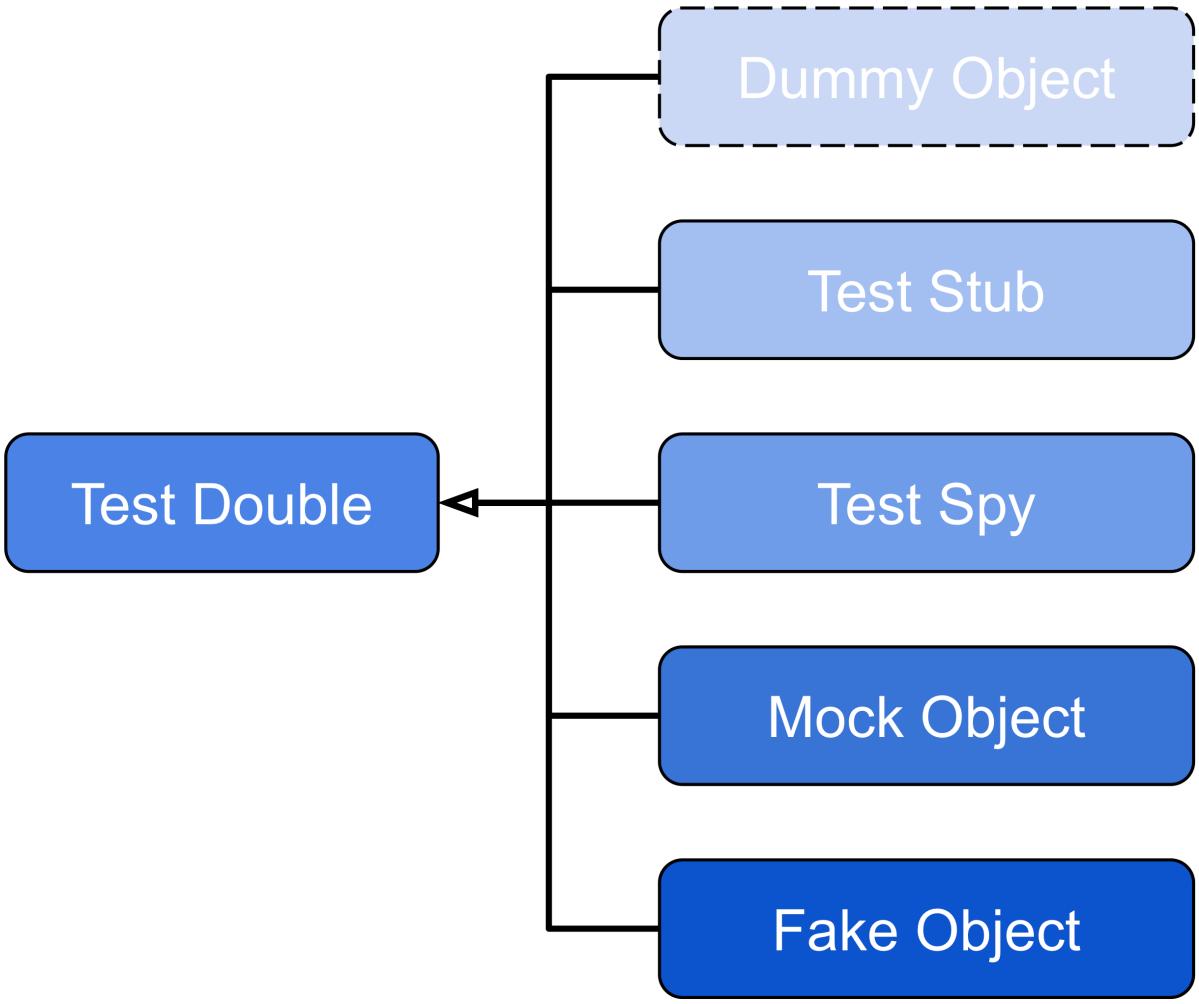


Figure 7.10: Types of test doubles

- **Dummy** objects are used to fill in a blank but are not actually used in the tests. For example, this can be a function argument that is needed to run the tests but does not have any impact on the processing.
- **Stubs** are objects that are pre-programmed to return specific answers during the tests. This can take the form of a TrainedModelStub that would return a set of canned predictions regardless of the inference inputs. They are usually used to go down specific logic branches, such as an EvaluationMetricCalculator, that checks if the model is trained before using it in the evaluation calculation.
- **Spies** are a cousin of stubs but with some memory of the interaction with whoever called them. This can take the form of a FeatureTransfomer that records how many rows were passed as input or how many rows were dropped because they did not pass some missing values condition. Another example is publishing metrics tests where a MetricsPublisher spy might record how many data points would be sent to an external ML metrics service.

- **Mocks** are pre-set with a set of calls they are expected to interact with. They can reproduce side-effects such as exceptions to test failure conditions. They record the various interactions they saw during their lifecycle and can be probed to confirm that they were called as expected. This can take the form of a mocked MockModelValidator that would return canned validation metrics or throw an expected exception if the model passed does not comply as expected. Mocks test for behavior
- **Fake** objects are built to take the place of a production object. Still, they have an almost full implementation of the object we need to replace. For example, objects such as an InMemoryModelRegistry would speed up the tests that need to interact with a FileSystemModelRegistry. This object would have the basic CRUD functionality without using the network for most unit and integration tests.

As you can tell, there is a lot of overlap between the various types of test doubles. A Mock is like a spy but moves the behavior assertion as part of itself. A spy is a stub with some memory of what happened. A stub is a dummy that returns some predefined output. A fake is different from the other doubles because it contains actual logic that temporarily replaces the real thing. This usually requires tests for the fake itself.

The level of coupling also increases as we move along dummy, stub, spy, and mocks. Each increases the amount of knowledge that the double has to know about the system's inner workings. The fake is usually the most coupled method since it redefines the core functionality of an external service.

Cost Effective Tests

When new ML engineers start their careers, it seems like they go from writing no tests to writing too many tests. Tests start piling up, and they become out of sync with the codebase. Following that, the testing suite's value drops, which pushes the devs to ignore the oversized test suite that might eat up their whole morning. Fear of touching the code settles in, and productivity drops. A straightforward way of avoiding this is to write fewer tests.

When designing the code under test, using good OO design provides an escape hatch. One of OO's best things is how it helps with managing dependencies by providing as much encapsulation as possible. Putting strong boundaries around your objects is key to good OO design.

Imagine that your ML pipeline is a series of messages going between a set of black boxes. By being a black box, each part of your pipeline has strong constraints around what other black boxes are allowed to know and what is internal only.

In [15], the author shares a great metaphor of imagining that each box is a space capsule. Outerspace is one hell of an environment. Nothing on the outside should see inside, and nothing inside the capsule should see outside. Only a few well-protected doors and windows are put in place to allow for maneuvering, fueling, and landing the capsule. I am sure you interacted with objects like that. They are a delight to work with. They maximize their own replaceability.

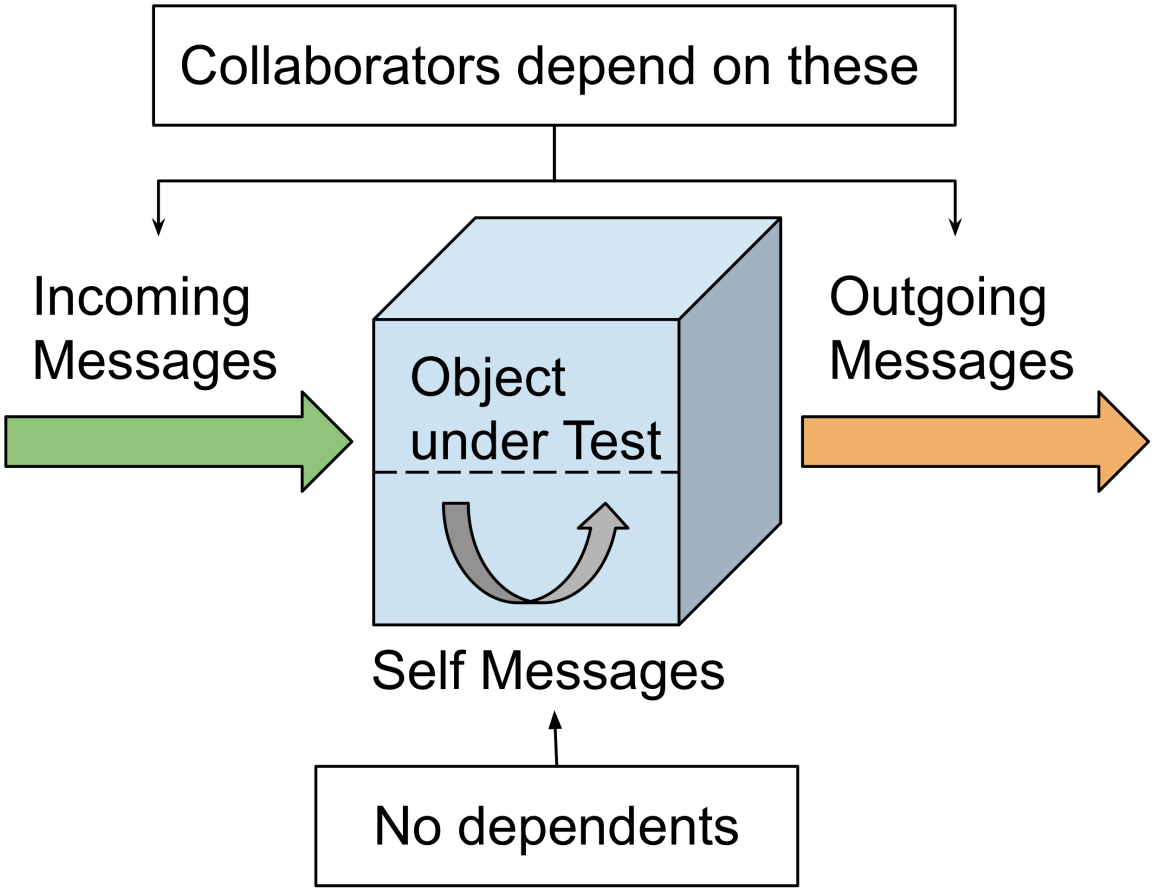


Figure 7.11: Viewing objects as space capsules in the merciless outer space encourages encapsulation

Tests are but another component of your application, and they can get entangled with the rest of your system. They become more vulnerable to change as you add more coupling between them and the application code.

The idea is, as much as possible, to only couple your tests to stable things.

The most expensive-to-maintain tests are the ones that drill a hole in the space capsule to get access to internal details of the object under test.

Say you are testing an iterative reinforcement algorithm that does repeated trials and regrets calculations. If your test has to access a private list of gradients to check that the learning loop is working, you coupled your test to the learning loop implementation. Any refactoring of the DQN learning loop will break that test.

We don't want that. We want to concentrate on the messages that go in and out of our object boundary. The incoming messages are the public interface of the receiving object. The outgoing messages are going to be the incoming messages of a collaborating object.

We already covered the distinction between Queries vs. Commands but let's look at their impact on

testing.

Queries have no side effects. A query returns something you care about, but it doesn't do anything that any other part of your app can see. Think feature transformer; It takes a single column categorical feature and returns its one-hot encoded representation.

Commands are the opposite. A Command has a side effect but returns nothing you depend on. Think model metric publisher; It takes a metric and saves it to a remote service.

It is easy to get burned by Queries that hide Commands inside of them. A developer runs a *.describe_model()* function that looks like a query, but ends up overwriting existing model metadata in prod DBs.

Now since we can classify incoming and outgoing message as queries or commands, here are the guidelines:

- Incoming Messages:
  - Queries: Make assertions about what gets sent back.
  - Commands: Make assertions about direct public side effects.
- Messages to self:
  - Queries: Do not make assertions about their results.
  - Commands: Do not expect to send them.
  - If you really need to test internal private functions while developing, add tests but DELETE them when done.
- Outgoing Messages:
  - Queries: Do not make assertions about their results and do not expect to send them.
  - Commands: Expect to send outgoing command messages.

This table, adapted from [15], summarizes this selection process best:

Message Type/Origin	Query message	Command message
Incoming Message	Assert the results	Assert the direct public side effects
Self Message	Ignore	Ignore
Outgoing Message	Ignore	Expect to send the outgoing message

Figure 7.12: What you can safely avoid testing

Property-based testing

Testing can become tedious work. You probably are going to get bored and ineffective after building the 10th dataframe from scratch. Crafting each column of each dataframe by hand gets old pretty quickly, so you throw in a couple of data generators in there. Your start building code that builds dataframes for your tests to run on. You store test CSV files around the codebase.

But then you notice that you don't have tests for that test data generators, and you start questioning your choices. "Should I be really generating this much data?" you ask yourself. Also, you start noticing that the data generation process does a lot of rework. It re-tests working scenarios mechanically instead of helping you focus on that new bug we found last week. In that incident, we had to deal with UTF-8 conversions in the feature transformation pipeline.

So you add more code to prioritize the hard cases. More test code means more maintenance of the test code. You say enough is enough. You throw the towel and assume that the feature transformation works for the small part of the input space you tested. Then you pray that the new to-be-scored data don't blow up in your face like last time.

The software community came up with a method to reduce this pain a bit: **Property-based testing(PBT)** .

Property-based testing is a generative test methodology that expands the input space of tests. Instead of defining specific inputs and outputs, this method needs you to "describe" the inputs you would like to send to your tests. Then you specify standard assertions of the expected behavior. The tools that implement property-based testing generate test cases for you.

Property-based testing libraries, such as hypothesis[16], provide a nice concise way of defining data generation strategies such as values, collections, maps, filters, and more.

Here is an example from [17]:

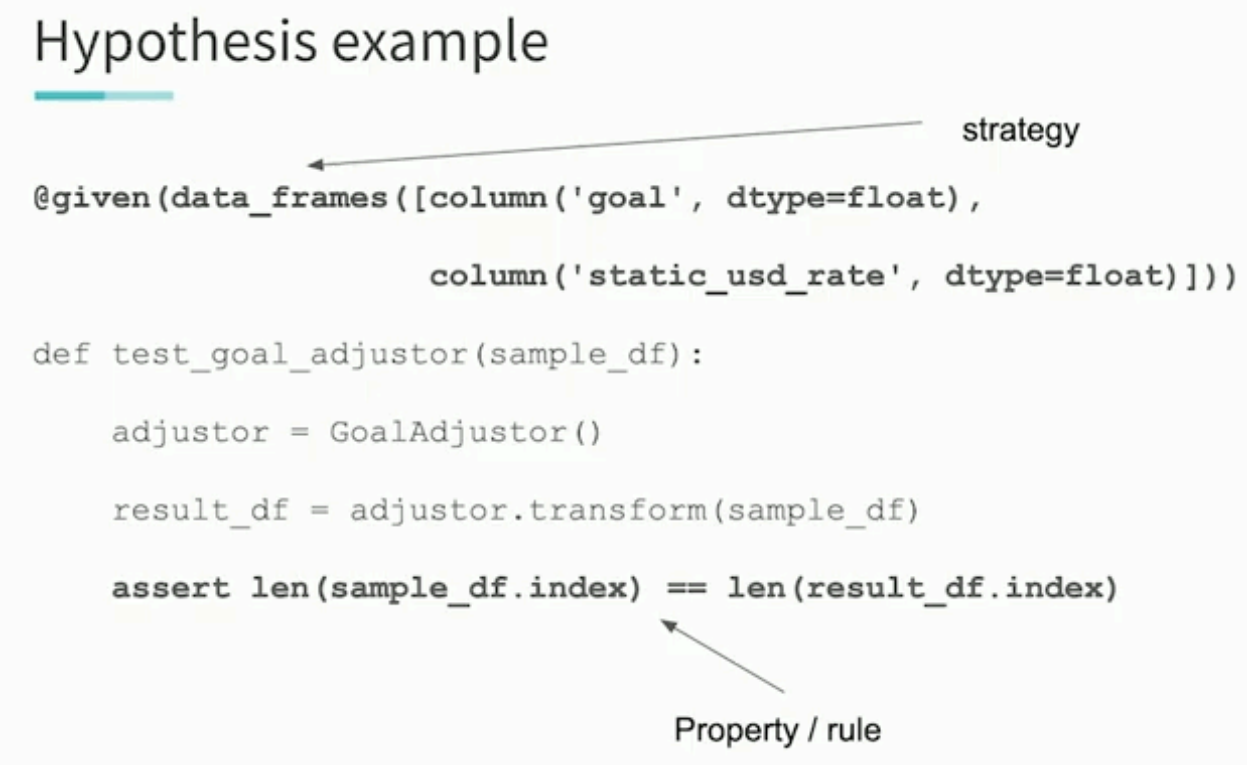


Figure 7.13: Hypothesis library example usage

```
---------- Hypothesis ----------
Falsifying example: test_goal_adjustor(sample_df=   goal  static_usd_rate
0   0.0             inf)

You can reproduce this example by temporarily adding @reproduce_failure('4.26.3', b'AXicY2RABf8/QBkAEKkBBQ==') as a dec
orator on your test case
---------- Hypothesis Statistics ----------
tests/test_transformers_hypothesis.py::test_goal_adjustor:

  - 26 passing examples, 30 failing examples, 29 invalid examples
  - Typical runtimes: 3-8 ms
  - Fraction of time spent in data generation: ~ 34%
  - Stopped because nothing left to do
```

Figure 7.14: Hypothesis library example output

In addition to finding falsifying examples [18], PBT tools also provide "**Shrinking**" capabilities. Shrinking is the process of producing human-readable examples of failure-inducing samples. The process takes complex examples and reduces them to simpler ones. Basically, instead of telling you

that a dataframe column composed of 100 URLs broke your feature engineering pipeline, it narrows down the problem to 2 or 3 URLs that trigger the same feature engineering error but in a more readable way.

In terms of what to test, the PBT workflow usually encourages the users to describe their outputs rules in one of the following test types:

- **Invariants**: This includes the shape of dataframes, the size of lists/set/dicts, thresholds on the values of the output data (all positive values, valid probabilities, valid classes), etc. [18]

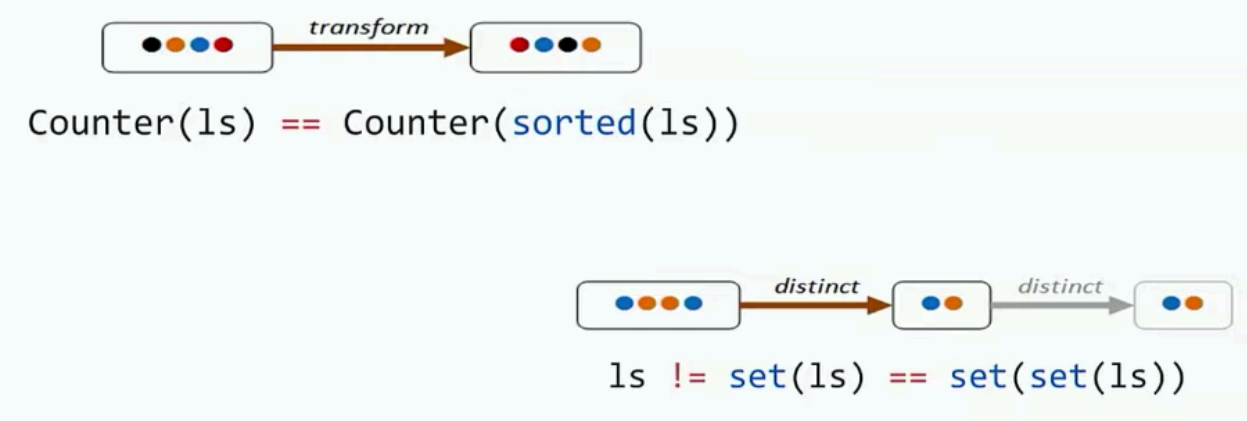


Figure 7.15: PBT invariant test

- **Round trips**: This includes any operation with a dual such as put/get, encode/decode, and the like. This can include deterministic functions that reverse a feature transformation or functions that encode and decode data between formats to feed into different modeling libraries. [18]

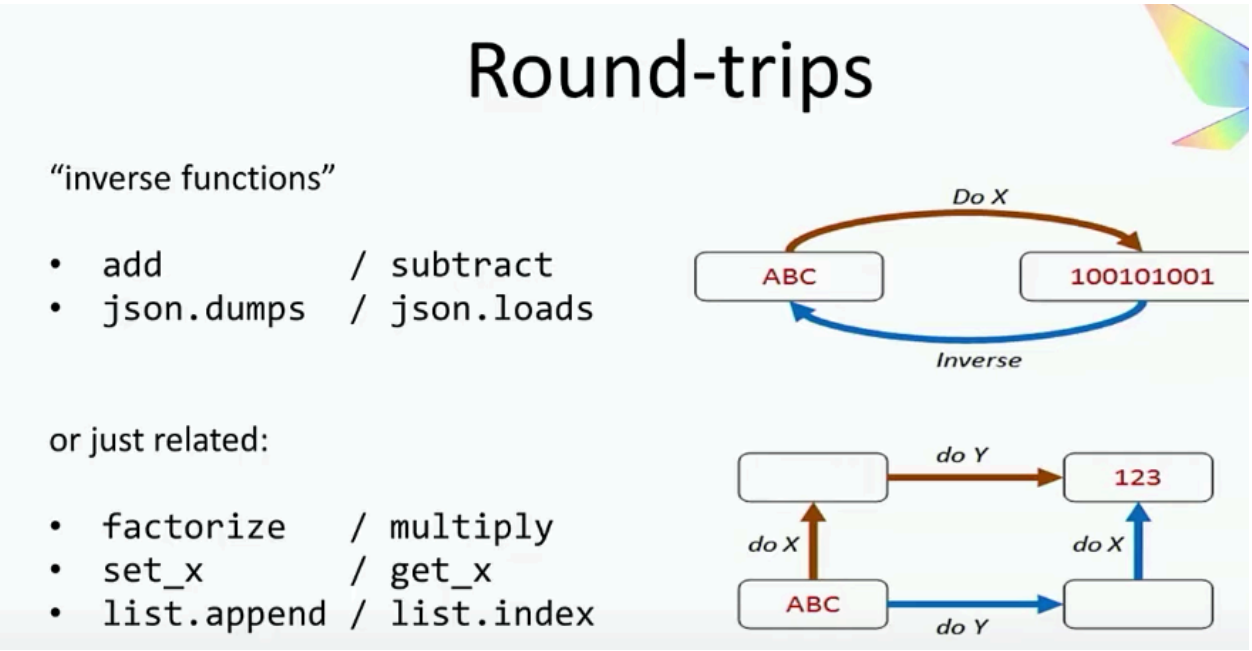


Figure 7.16: PBT round-trip test

- **Does Not Crash**: This one can seem silly. The goal here is to throw a bunch of data at a functionality without defining any output rules except that it should not go belly up. This is extremely useful for finding corner cases. For example, it can be used to determine that a feature transformation does not support strings with spaces in them. This can avoid the usual blame game of "Well, we thought that the ingestion pipeline took care of sanitizing the strings and removing spaces." Even if this rule type can feel like cheating, it is useful in practice.[18]

```
@given(lists(integers()))
def test_fuzz_max(xs):
    max(xs) # no assertions in the test!
```

Figure 7.17: PBT does-not-crash test

- **Oracle tests**: An oracle can predict the future. It relates to the situation where we have an existing function that works great and produces correct outputs, but that we need to extend. The suggested process is to write a new function with the new functionality and compare the old vs. new function's output. Basically, we are using the original function as an oracle to compare against. This pops up in many forms in ML code. For example, say you are building a specialized logistic regression library with custom-built handling of categorical data. The Scikit-learn library has a standard and highly performant logistic regression implementation.

The idea is to train the two libraries on multiple but identical generated datasets and compare the resulting post-convergence loss value and model parameters.

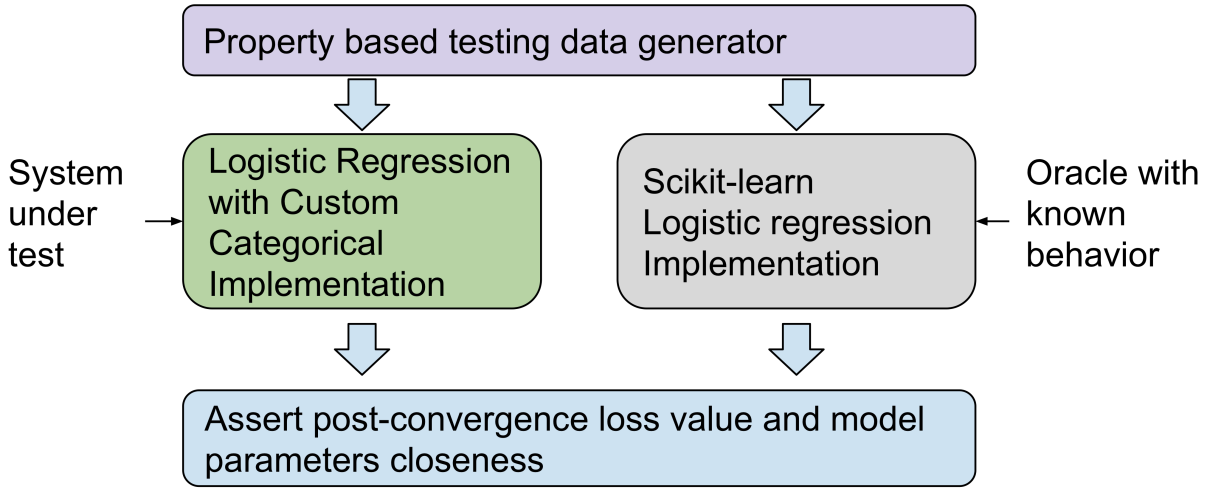


Figure 7.18: PBT oracle test

One last point to touch on is **stateful testing**. Instead of writing the tests yourself, PBT tools generate the tests and the data in one package. The tools require you to provide a list of basic actions that can be combined together. Then the testing tool tries to find falsifying sequences of actions that would cause a failure. This kind of testing is also called sometimes "model-based testing." The key idea is that you don't really need to care about the "state" part of the stateful testing, only that your code has a set of primitive actions that can be combined together to produce tests.

Exterminate Non-Determinism in ML Tests

There is no doubt that an automated regression suite can save your mental life.
It reduces the cognitive load of modifying existing code and prevents old bugs from reappearing.

The issue with ML code and generic code is that we deal with a lot of non-determinism.

This shows up as tests that pass most of the time but fail for no apparent reason.
When MLEs don't tackle this problem head-on, the value of the tests is dramatically discounted by the team.

The Basics

The first thing you want to do is set all the seeds for all the random number generators used in your ML pipeline components.

```
random.seed(42) // python
numpy.random.seed(42) // numpy
torch.manual_seed(42) // pytorch
tf.random.set_seed(1234) // tensorflow
```

However, ML libraries will routinely have no way to entirely control randomness. Quality packages usually declare their limitations with sentences like "There is currently no simple way of avoiding nondeterminism in these functions."[19]

In this case, you need to resort to threshold and range-based tests if you are testing the internals of your ML algorithms that depend on third party packages.

Social Distancing

The first thing to do when you are faced with non-determinism in your test suite is to throw them away. There could be 15 tests that are non-deterministic out of the 200 tests you have in your codebase. When these tests fail, initially, the team looks at the failures and remembers that they are non-deterministic. They at least initially pay attention to the deterministic tests and ignore the other ones.

However, let's not kid ourselves! Attention is a scarce resource.

They will inevitably start ignoring the failures in the deterministic tests because that is the path of least resistance. Over time the whole test suite gets ignored. You might as well just throw the whole test suite away.

The first mitigation technique is **quarantining**[37] the non-deterministic tests.
Social distancing is another way of looking at it. Don't let them infect the rest of the healthy tests.

The benefit you get is the feedback from the deterministic part of the test suite. The danger is that you'll forget about these quarantined tests and let them rot in the dark. Place a limit on the number of tests that are in that area. 10 tests seem to be a reasonable limit. So no more than 10 +/-5 tests in the quarantine area.

An easy way to set tests in the quarantine area is to put something like @pytest.mark.skip. The nice thing about that is the statistics you get at the bottom of your test report. "**10 tests skipped**, **190 passed**" should act as a wake-up call to heal these sickly tests.

Isolation And Co-mingling

Getting a handle on the environment in which the tests run is the first step towards a sane testing suite. For example, suppose a feature engineering integration generates data in a DB table and leaves it there. In that case, a second test can get corrupted because of the leftover crumbs.

Test isolation is a vital characteristic of a good test suite. It starts by being easy when the number of broad scope tests is small. But it gets harder when multiple component level tests start messing with the same resources.

Enforcing isolation can be done either by actively rebuilding the initial state from square one or carefully cleaning up any remains after a test runs. The guidance is to follow the first option because it simplifies debugging failing tests. By relying on tests' goodwill to clean up after themselves, bugs' sources get harder to spot.

In the case of unit tests, favoring the clean slate approach is easy to start with. For more extensive component tests and e2e tests, it gets harder to load a full dataset every time they are run. In that case, you can rely on reversing the side-effects of the tests to speed up the inter-test frequency.

One strategy for cleaning up between tests in Python is to rely on context managers [20] that offer some insurance against random crashes in the middle of a test. Building a custom context manager with an **enter** and **exit** functionality or a decorator/generator combination is surprisingly accessible.

As a counterpoint to isolation, you might be tempted to provide a strict order to your tests instead of isolating them. This strategy can be useful for waiting for certain events before launching the rest of the tests. However, beyond a couple of smoke tests, managing this complexity level might be too much to bear, short of using a full workflow manager to run tests in order. Isolating the tests has the double benefit that they can be run in subsets and run in parallel to speed up the testing process.

The Brave New Async World

Asynchronous tooling allows MLEs to reach high levels of concurrency to speed up their workflows. Async feature transformations that rely on lots of IO benefit from staggering operations to do something useful instead of waiting for a slow data loading to finish.

Say we are generating features from three different data sources. Using an async method to process the three data sources would save us from having to suffer from the slowness of the slowest of the three data sources.

The same applies to multi-processing, where a set of futures can be created to process segments of your data in parallel, and only when they are all done processing do we merge the results. This has traditionally improved the runtime of many algorithms and data preparations when we can partition them into multiple cooperating units of work.

However, when testing, asynchrony can be a tough nut to crack. The common blunder for dealing with async calls is to use a period of sleep:

```
async_build_feature1()
async_build_feature2()
async_build_feature3()
sleep(x_seconds)
check_the_results()
```

This usually causes problems for devs because either they set the sleep too short or too long. When the sleep is too short, we get many false positives; the tests fail because the process didn't fail but is not done yet. When the sleep is too long, the tests are slowed down dramatically. Finally, we may find the perfect sleep time interval, only to be hit by a change in the environment that changes the ideal sleep interval to a different value.

The alternatives to using this sleep anti-pattern are to use **callbacks or polling**.

Callbacks are functions that you, as the consumer of an API, provide so that when the API you are calling finishes, it can invoke your callback functions. Usually, we see callbacks in ML code where on each end of a training epoch, we would like to calculate and record metrics to keep a history of the training evolution. This strategy can be used in your tests as well as long as we have control over the API we are testing, and we can add the ability to use callbacks. Using the feature transform example, we can embed assertions as callbacks in the transform function. As soon as the async transform finishes, the assertion conditions are checked, without having to guess when the transform will finish.

The second strategy is to poll the results. Instead of checking the results once, we can check multiple times at regular intervals.

```
async_build_feature1()
start_time=now()
while(not_done_with_transform):
    if current_time - start_time > time_limit:
        raise FeatureTransformTimeoutException
    sleep(polling_interval)
read_results()
```

The significant advantage here is that you can set the polling interval to a configurable value to raise the frequency of the checks and minimize the total wait time.

During the checks, it is beneficial to use a formal Exception type to give you enough information about the reason for the test failure. Using a simple return would lead to false negatives because the subsequent assertions will be skipped. Testing frameworks usually can use that information to notify you that the failure was the async waiting time rather than bugs in the transform's logic.

As always, it is advisable to have all the timings and conditions fully configurable from a single place. You will surely need to change those timings many times as you explore the behavior of the async calls.

Another piece of advice is to add initial smoke tests to check that an async dependency is alive. This is especially useful when the async dependency experiences a failure, in which case waiting for the timeouts will slow down the test suite considerably. Running a quick check that the dependency is available can save you some time.

In the end, you probably should not even have to deal with these problems. If you find yourself dealing with some business logic that is trapped in an async environment, the method to use there is to isolate the logic and test it separately in a synchronous manner. Basically, the goal is to minimize the places you actually have to deal with async behavior.

Working around Remote Services

Many a time, MLEs find themselves downstream of gigantic data streams that are managed by external teams that have their own schedules. Integrating with these systems can be a huge headache because test systems might not be in place. In many cases, ML apps might need to rely on touching prod data systems directly.

In this case, you must reach for isolation and determinism as much as possible. Wrappers and Test doubles should rank high on your list of things to use to ensure reproducible tests.

We already covered this point in this book many times. Still, suppose you interact with an external data source with a non-deterministic behavior (e.g., changing datasets, varying response times, fluctuating cluster statuses, etc.). In that case, you should save yourself some future-time and isolate your code from those dependencies.

Test doubles in ML and DS code usually get a bad reputation because teams are used to run code directly against production datasets and production APIs. Using a Test double to hide a remote service feels empty of meaning. *"I can't possibly fake all the possible data combinations that we get from the upstream data feed"* is a common argument for avoiding full isolation of the tests.

This position is, unfortunately, in opposition to the fact that you can throw away the benefits of automated tests if they are non-deterministic. But who am I to prescribe determinism. In the end, it is up to the team to choose what works best for them.

Clocks

Simply put, if you are dealing with time, you have to have a way to wrap the source of time and replace it with your values. You will also want to sync your test datasets DateTime fields with the clock wrapper and have a process to move the time wrapper and data in tandem. Besides, you probably will want to avoid having random ID generators that are based on the system clock. The joys of dealing with daylight savings during a leap year on a UTC+30min boundary, anyone?… anyone?…

It Only Fails During Business Hours

You are probably running your integration and system-level tests on shared resources. You have this DB query that runs fine after 5pm EST. Still, during the business hours, you get things like "connection pool is full" and "process was evicted due to memory pressure." This can cause a headache because of the non-determinism involved.

Integration, component, and system tests routinely ask and receive resources such as CPU/GPU/memory/connections/filehandles. These resources are usually shared across all the tests, which will cause processes to compete with each other instead of coordinating their access patterns. The standard solution to this is to set resource pools that constraint the tests to a small number of resources. If any tests ask for unavailable resources, the test should make a loud noise. This will help you find the issues at testing time rather than figure out which pipeline component caused the production issue.

Test Coverage

Once MLEs get a hold of the testing framework, hooks, and workflows, they usually come across the coverage metric (also known as test coverage or code coverage). The next natural step is to drive towards optimizing this metric like they optimize the rest of their log losses, MAPs, BLEU score, etc. This can be held as a medal of pride in the quality of the codebase, but it misses the goal of coverage.

Coverage is intended to help the developer find untested sections of their code. It is not, I repeat, it is NOT to be used as a metric of how comprehensive the test suite is.

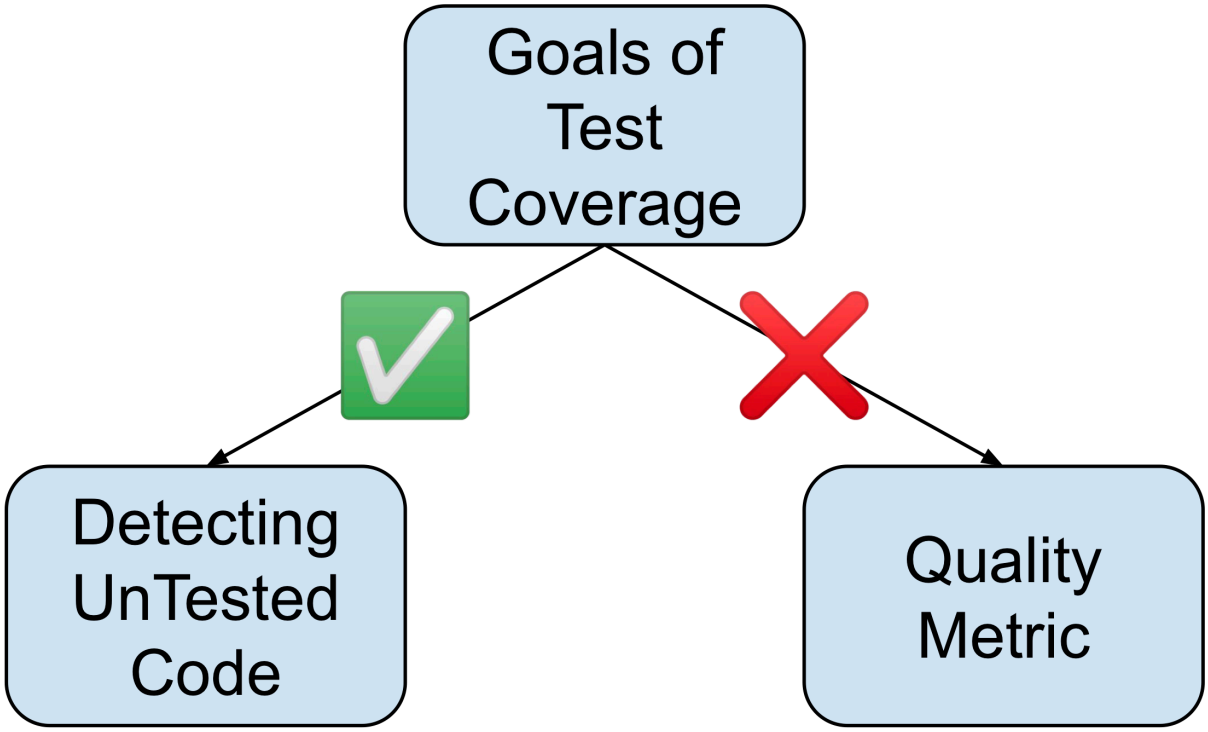


Figure 7.19: The use of Coverage metrics

The primary issue I see with coverage is that it encourages us to write lots of tests for things that don't matter just to move the needle from 85% to 86% and add it to our performance reviews. This is a distraction that takes time away from testing lines and sections in your code that actually matter.

Coverage is usually used to know by the dev to know if their testing is sufficient. Unfortunately, there is not a hard and fast rule for that. 50% coverage is not good that we can agree with. Still, above 80%, it becomes harder to use coverage as a metric to evaluate the testing quality.

A better measure for "having enough tests" is to see that your team rarely releases bugs in production. You are rarely afraid or scared to modify code because you are unsure if it will create bugs in production.

Note the dual goals here: **no prod bugs + no fear of modifying existing code**.

The value of the coverage is in finding sections of your code that are not currently tested or not tested enough. The code coverage tools can point to low-risk portions of your code like that script you use to start the application locally vs. high-risk portions of your code like the new code you added that does feature transformations on the timeseries of the customers' bank account balances. Which of these worries you more? The coverage tools will point both of them out, but ultimately it is up to you to decide which test you should include next in your suite.

What To Do If You Are Giving Up on Testing

What if testing is just not for you.

You tried it. But it is just too much overhead.

Management does not care if you write automated tests or not.

Your product contribution is low risk. For example, it adds a field to some table that adjusts a downstream score for a mortgage retailer. You are responsible for predicting the median price of houses by county. This is used selectively by a downstream process that provides recommendations to users based on their search filters on a real estate application.

If your API goes down or the table does not get populated, they have a historical average that they can use instead of your predictions.

Fine. No problem. I am not here to make you even less productive. You have your own problems: The data sources upstream of your app have been mutated so many times you forget what queries work anymore. The deployment process changed 3 times in the past 3 quarters. Besides, the product manager is not even sure yet if they want to include that predicted median house price in the recommendations data pipeline feature set.

You need speed, and you need it now. What do you do? What are the components that you might need to keep the lights on even if you don't have any tests in your code?

Basically, how can you reduce future suffering from the lack of testing?

P.S.A: I am not advocating for skipping testing, but I don't know your life. All I am saying is that if you find yourself in a bind, with mounting pressures to deliver new versions of untested ML software, you can at least benefit from a disciplined way of testing in production using the following tips and tricks. Beware the techniques below might be useful even if you **are** writing tests for your ML application.

Testing Expeditions a.k.a. Exploratory Testing

You are probably already doing this in your notebooks. You get some dataset and some vague business requirements, and you poke at the code until it confesses. The exploration is useful in many ways, where scripted testing is too much for the developer to put in place, or the requirements are just too vague to build test cases ahead of adding the code.

Nevertheless, in [21], the authors explore exploration testing and report ways to instill discipline. The two main ways I liked are time-boxing and slicing the part of the software we are experimenting with per testing session.

In an ML context, the exploration sessions should aim at learning about the business requirements and how the current code at hand handles subsets of that. The sessions should generate hypotheses about what the code does and more test scenarios to understand the data, feature transformations,

and modeling algorithms used. The key is to split the sessions into charters that have bounded focus areas and record any questions that arise from the exploration for future sessions. While exploring the behavior of one of the buggy feature transforms, it is easy to get dragged into unrelated code paths. Exploring the impact of changing bucket sizes on downstream model predictive performance might drag the dev into exploring the expected range of values allowed in upstream data sources. The time-boxing combined with a focused charter, can help with deliberately cutting unnecessary distraction.

Exploratory testing is usually carried at the beginning of an ML product lifecycle. Still, it can also be beneficial after the first bugs trickle down from the production deployments. The manual checking of why the feature synthesis does not accept Asian characters in a single, bounded, and focused session can provide valuable feedback to the ML engineers.

Synthetic Monitoring

If you are not writing tests for your ML products, then you are probably already doing some type of synthetic monitoring. You probably have some scripts and code that runs against the live version of your ML scoring service to check its health and the range of values it is returning. This method piggybacks on the existing production infrastructure to find your requests recorded in various monitoring services by default.
You also probably use a separate user account or a fake customer id in your system to poke at the live version of your service. You create new items, run fake searches, run user journey tests, and the like to make sure that we are not returning garbage to the callers.

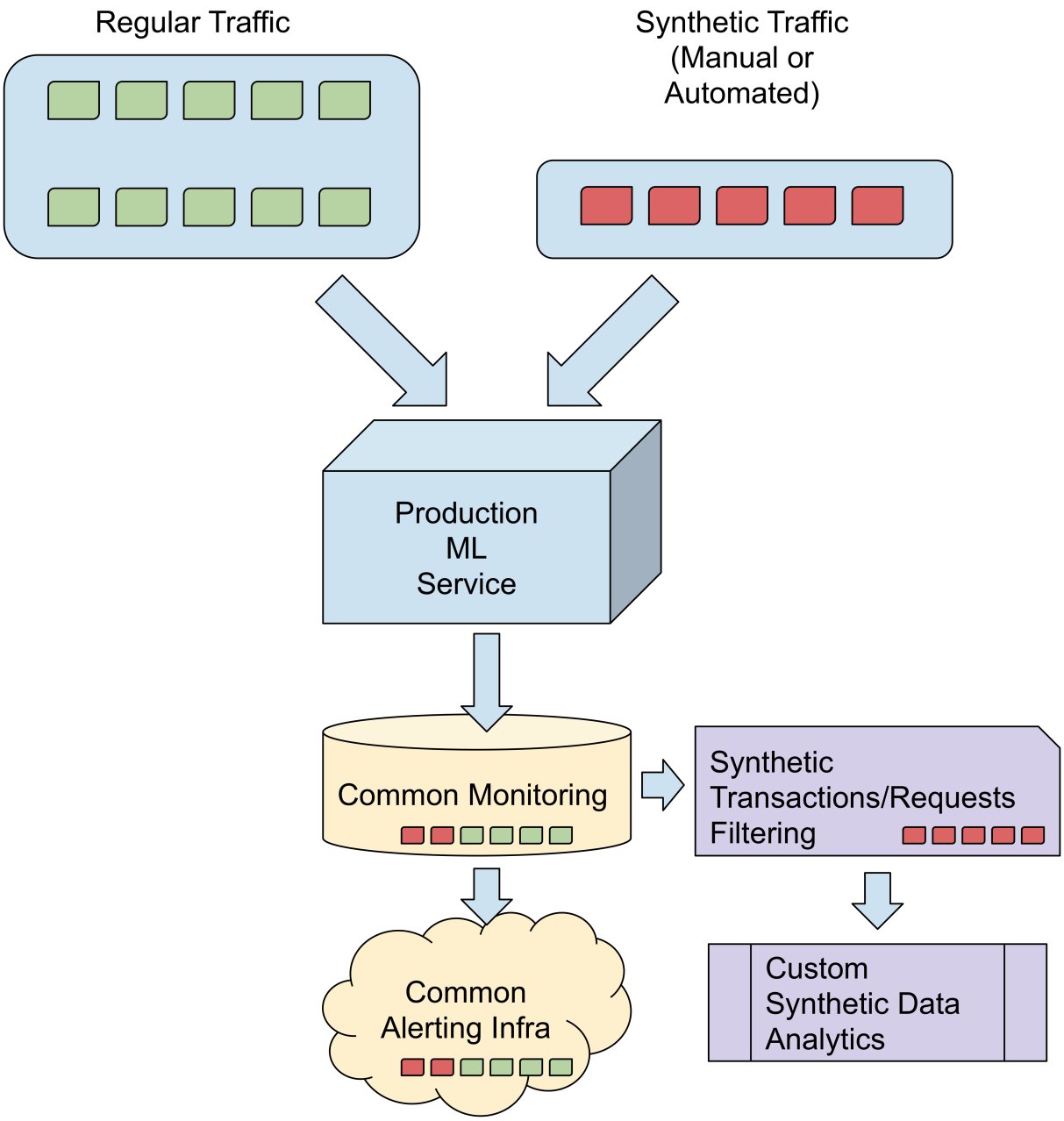


Figure 7.20: Synthetic monitoring boils down to dropping pebbles on the live system and trying to collect them at the other end

Synthetic monitoring springs from the age-old tension between optimizing the MTBF (mean time between failures) and the MTTR (mean time to recovery)[22]. You can find a nice write up about this technique here[23].

Instead of focusing on preventing bugs, you instead focus on fixing problems as quickly as possible. It goes without saying that this strategy would benefit significantly from a robust continuous delivery infrastructure. Suppose you find a problem with the insurance premiums predictions at 11AM on a

Tuesday because the learning rate was set to 0.001 instead of 0.01. In that case, you can release a fix later that day at 4PM with a single git commit and a release button.

If you start relying on this technique extensively, then it is advisable to start passing custom HTTP headers, URL parameters, or synthetic transaction identifiers to allow the downstream services and batch jobs to differentiate between the synthetic tests and the real customer traffic/data. In addition, these synthetic tests can be automated to hit production services, nicely, without taking them down, and feed the results to the regular monitoring backends and potentially trigger the regular pager duties of the team. If a synthetic test triggers an unknown bug in the prediction pipeline, it might be better to raise a non-synthetic alarm before the real customers get to experience the bug.

Feature Toggles

Given that you might have a vague idea that the end-to-end ML pipeline works, on the extensively curated test dataset, you might be tempted or pressured to release it to production ASAP.

The scenario I am talking about is where we have an already working version of the prediction service. However, we want to check if the new without-tests version of the modeling stack can generate more clicks, views, conversions, purchases, revenue, or profits. This means that we want to bolt the new ml code to the existing "stable" code. Basically, we want to change the predictive behavior without making a big bang release because the risks are unknown without the guiding lights of tests.

Feature toggles play precisely this role[32]. They allow ML teams to modify which model component gets run at scoring time without having to do a full release or modify existing code.

"Wait a minute!" you say. Isn't that what A/B testing is? Yes, Feature Toggles are the mechanical part of A/B testing. From a traditional software perspective, devs start off using feature toggles and realize that they can use the same infrastructure to activate latent code paths to perform canary releases and even A/B tests. However, from the ML dev perspective, it usually starts off as being infrastructure and code modifications to enable A/B testing only to be recognized as a means to fix bugs, improve system performance, or extending capabilities that have little to do with predictive performance A/B testing. So basically, you probably already know what Feature Toggles are, and I bet you are already using them.

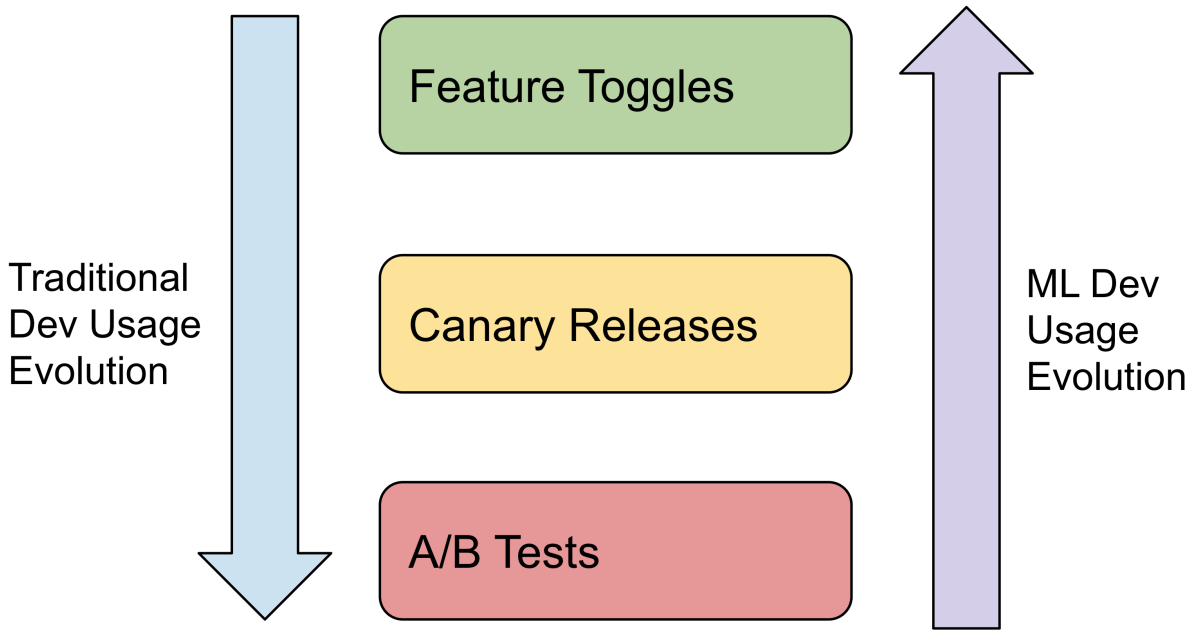


Figure 7.21: Differing points of view on Feature Toggles

Over the years, the industry came up with a structured way of talking about feature toggles. There are four main components that are usually put in place: The toggle point, router, config, and context. The toggle point is where the decision is enacted. The router reads a configuration and a context to determine which code path to enable.

In the example below, we have two prediction consumers that share a common code base. Say we want to enable a bug fix in the search-model serving component. The feature toggle point would live in the serving service, but choosing which code path to choose would be delegated to the toggle router. The toggle router, in turn, can be used to not only provide a decision but to examine the current context, such as which customers are searching. If they are high-paying enterprise customers, we might want to continue serving the stable predictions. However, if the current search comes from a casual user of our search service, the router can decide to enable the ML model bug fix A. Finally, if it is an insignificant user, we could use that traffic to test-in-prod the ML model with the untested code path C. Once we see that nothing bad happens, we can gradually expand the traffic that is served by the untested code (or is it tested now? since we tested it in prod with real traffic? I'll leave that up to you and your priorities).

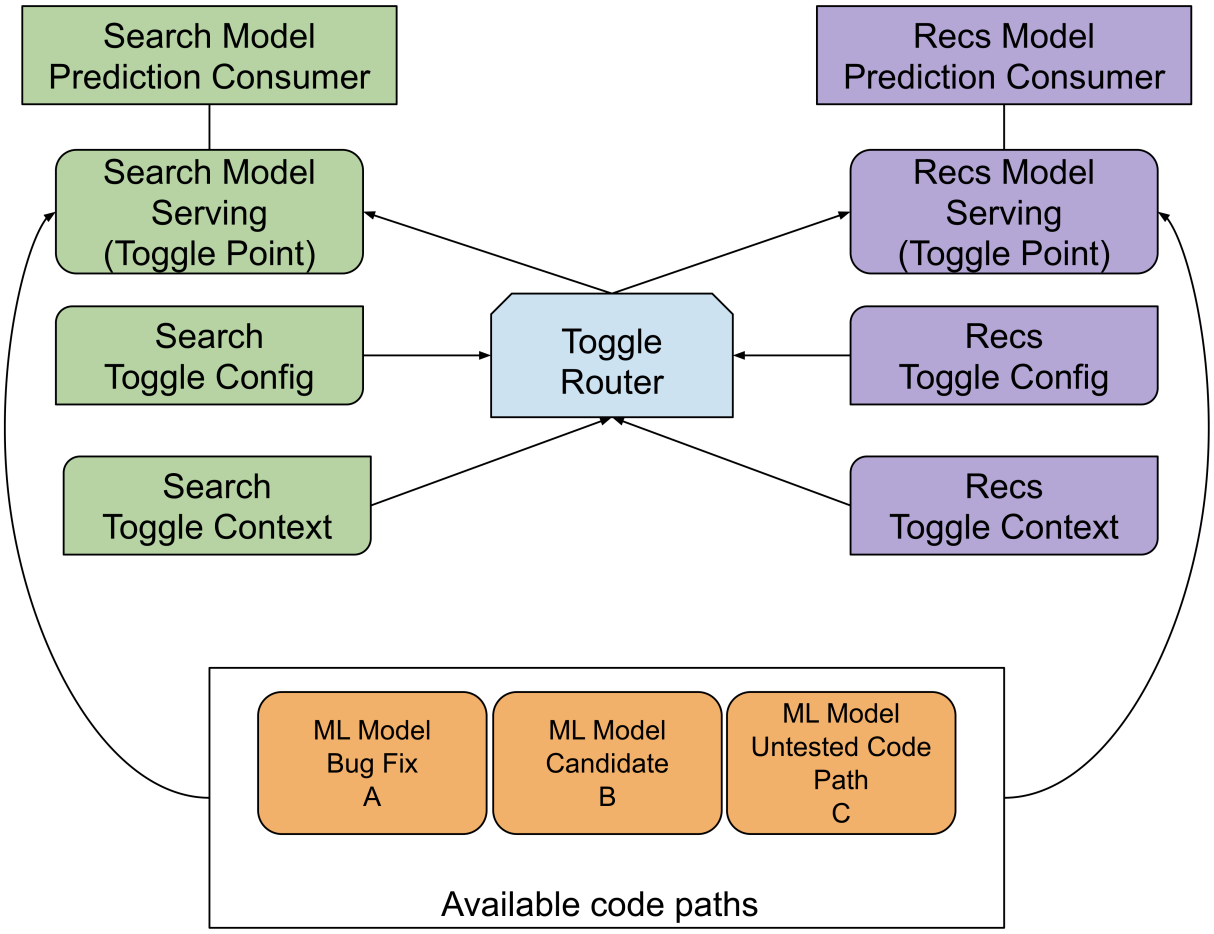


Figure 7.22: Putting Feature Toggles' infrastructure in perspective

We can see how this can gradually evolve from a simple code path switch, to a canary release, to a full-blown A/B testing infrastructure.

While feature toggles add complexity, there are multiple ways to reduce the complexity, but mainly:

- **Configure feature toggles in a single location (maybe two)**.
  - Don't go around copying the same if/elif/else all around the codebase.
  - Use a toggle config with a centralized context-aware toggle router.
- **Remove the toggles when done migrating to the new version**.
  - You can keep the feature toggle infrastructure, but make sure to remove old configurations and the old code that was activated by these configurations.
  - Don't be the next Knight Capital[33].
- **Read** this fantastic writeup about Feature toggles and how they complement trunk-based development [24].

An additional variation on this idea is Dark Launching [25]. For ML apps, the idea follows the same conceptual framework where an enhanced ML service is deployed but stays invisible to the end-user. This one works best when we want to enhance an existing user workflow. The key is to have both the stable and the new services running side by side and send the same traffic to both instances.

The difference is that the customer only sees the results from the stable service, and not from the new service. Instead of serving the predictions to the end-user, the outputs can be funneled to a data collection component that will hopefully provide ample feedback to the owner of the new ML service. Once the new ML service is acceptable, the feature toggle infra can be switched to fully utilize the secondary path and actually return the outputs to the end-user.

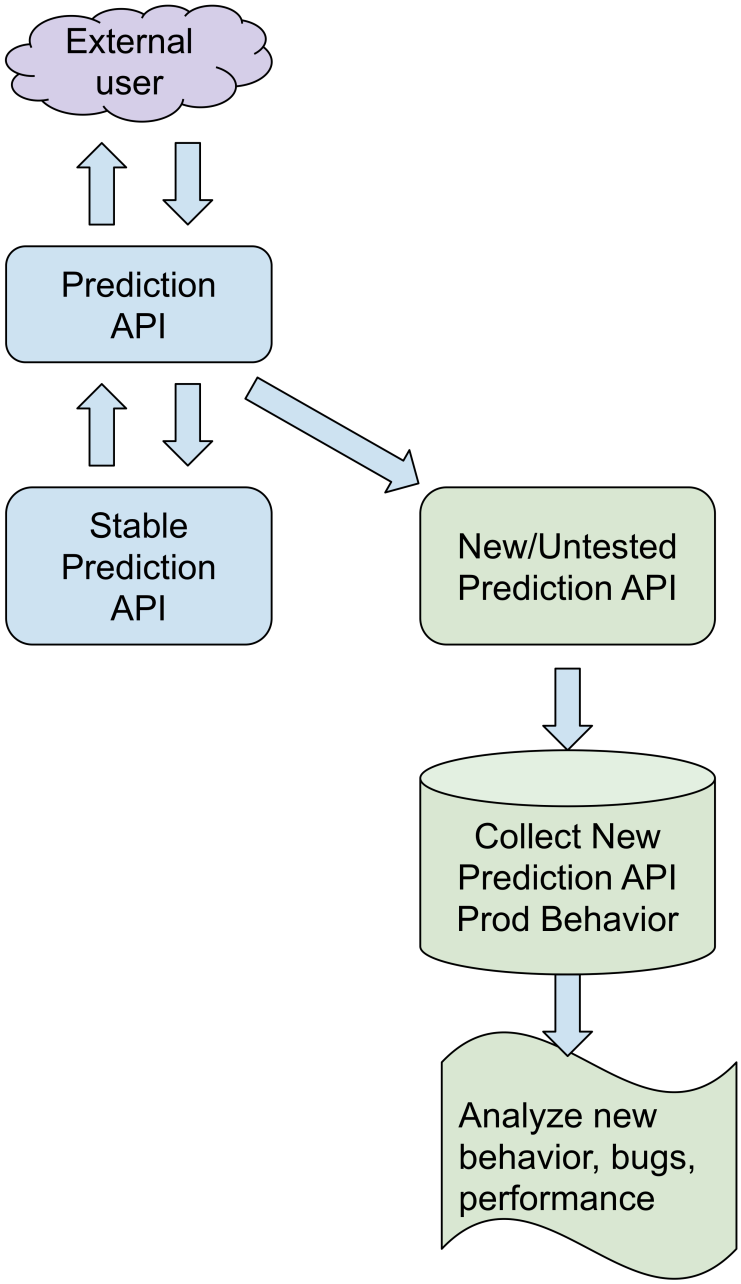


Figure 7.23: Dark Launching to exercise a new/untested API

It is important to note that dark launching is suitable for workflows that do not require the users to choose between options. For example, if we are enhancing an existing ranking API, dark launching applies. If we add new functionality that is not already supported in the currently deployed workflow, then the new functionality will not be exercised in the darkly launched service unless the user decides to choose that option. For example if there is a dropdown that specifies the ranking type that the user would like to employ, then that would require the user to click on the new ranking option, which becomes a prerequisite to sending traffic to the darkly launched service. In

the cases where users must choose among various options of behaviors, canary deployments are to be preferred.

Approaches From Around The ML Community

Software 2.0

The software industry welcomed the redefinition of the software 2.0 metaphor by the chief data scientist at Tesla motors circa 2017[26].

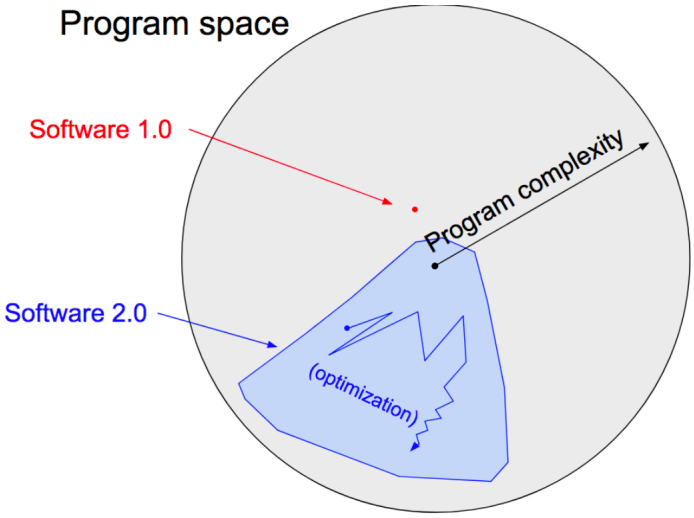


Figure 7.24: Comparing Software 1.0 to 2.0

The argument was that contrary to the software 1.0 method of carefully crafting manual solutions to problems, software 2.0 works as follows:

- Specify the behavior we would like to see by specifying a dataset of input and output pairs of examples.
- Specify the search space by writing a skeleton or template of the software with an untrained or sub-trained model with a train eval loop and a target metric.
- Use computational resources to explore the search space efficiently.

The ML Test Score

The ML test score is a checklist like a document that provides a rubric for rating ML systems in production. This seminal paper [27] emphasizes the production level concerns that are different

from offline experiments. They provide a set of 28 testing guidelines for ML systems. They focus on the challenge that it is hard to know a priori the predictive systems' behavior.

Like the software 2.0 ideas, the ML test score surfaces because we have a completely different set of challenges when dealing with predictive systems, as seen in the figure below.

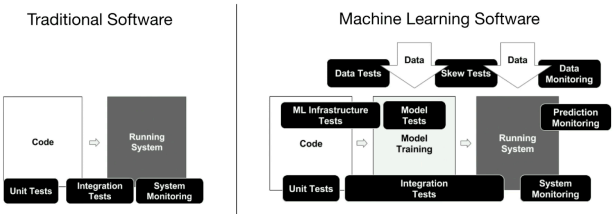


Figure 7.25: Extra components to handle in MLSE vs TSW

The suggested tests are split into 4 categories: Data, Model, ML Infra, Monitoring.

Data tests

1	Feature expectations are captured in a schema.
2	All features are beneficial.
3	No feature's cost is too much.
4	Features adhere to meta-level requirements.
5	The data pipeline has appropriate privacy controls.
6	New features can be added quickly.
7	All input feature code is tested.

Figure 7.26: Data tests

Model tests

1	Feature expectations are captured in a schema.
2	All features are beneficial.
3	No feature's cost is too much.
4	Features adhere to meta-level requirements.
5	The data pipeline has appropriate privacy controls.
6	New features can be added quickly.
7	All input feature code is tested.

Figure 7.27: Model tests

ML Infra tests

1	Training is reproducible.
2	Model specs are unit tested.
3	The ML pipeline is Integration tested.
4	Model quality is validated before serving.
5	The model is debuggable.
6	Models are canaried before serving.
7	Serving models can be rolled back.

Figure 7.28: ML Infra tests

Monitoring tests

1	Dependency changes result in notification.
2	Data invariants hold for inputs.
3	Training and serving are not skewed.
4	Models are not too stale.
5	Models are numerically stable.
6	Computing performance has not regressed.
7	Prediction quality has not regressed.

Figure 7.29: Monitoring tests

ML Score Checklist Visualized

A more approachable explanation of this checklist was developed here [28]. In that set of informative videos, the concepts of the ML score are mapped on the following ML testing framework:

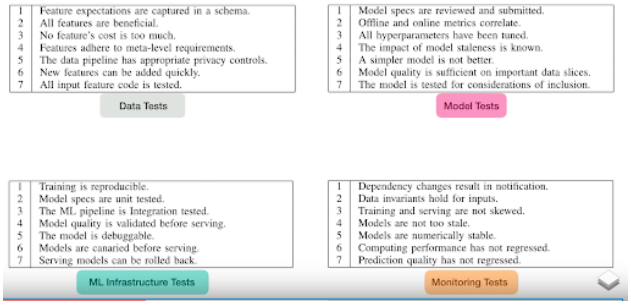


Figure 7.30: ML test score by category

Which color codes the various ML tests, as well as mapping them to the ML score machine learning software process diagram:

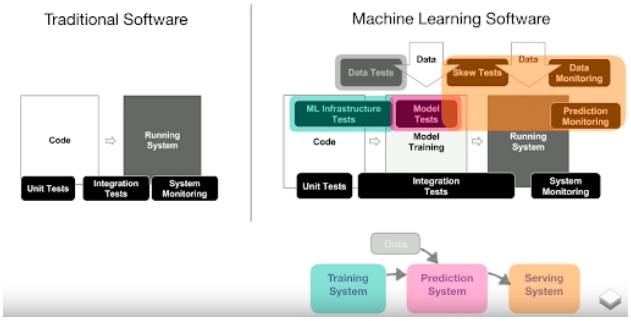


Figure 7.31: ML test score over pipeline

The work splits the process into 4 parts: Training, Data, Prediction, and Serving Systems. This helps map the content of the checklist over the individual parts of the system.

Furthermore, this helps differentiate the testing across the various phases of the ML pipeline:

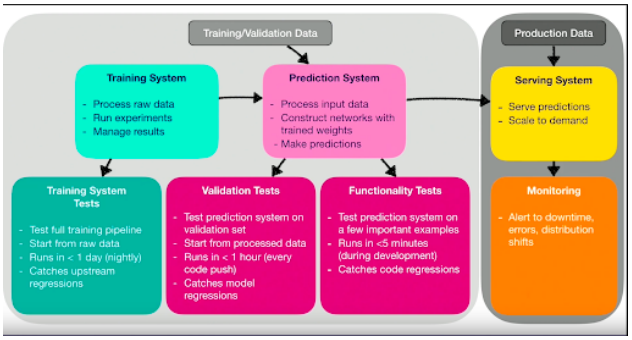


Figure 7.32: ML test score over phases

The work exposes the need for solid Continuous Integration (CI) tooling when it comes to enforcing tests. Also, they provide an interesting view of where CI tooling becomes useful for ML workloads:

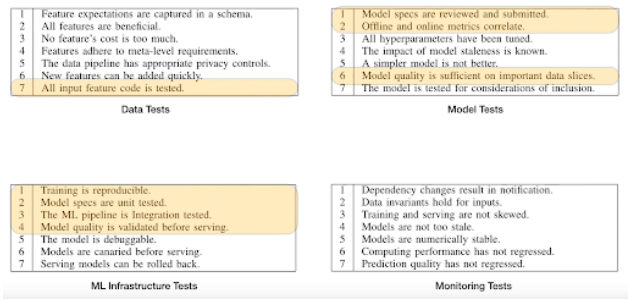


Figure 7.33: ML test score over CI

Coding Habits for Data Scientists

In this article [29], the author uses a useful analogy to the Law of Flat surfaces that says that "Any surface within the home or office tends to accumulate clutter." The authors point out that Jupyter notebooks are the flat surfaces of the ML world.

The author points out that notebooks are excellent tools for exploration during the initial phases of a new dataset or a new business problem. However, notebooks routinely lack structure and grow organically until they reach a tipping point where it gets harder to change the code in the interrelated code cells.

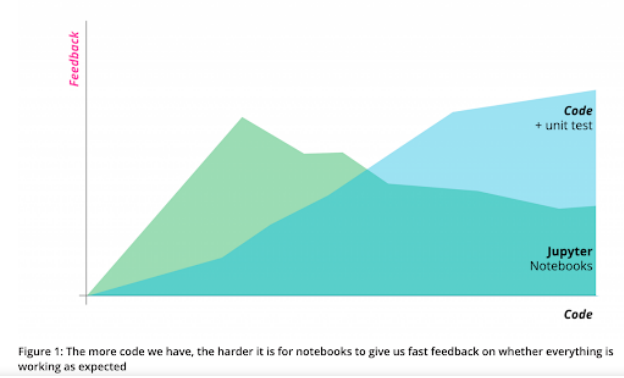


Figure 7.34: Feedback of Code vs Notebooks over Code amount

The author provides a visual interpretation of the problem at hand. As we have more and more code to manage, notebooks peak quickly on the feedback scale. The format of the notebooks alone does allow for large codebases. On the other hand, constantly extracting functionality from notebooks and isolating them for testability is one way to extend notebooks' usefulness.

The author also provides a handy refactoring workflow to complement notebooks with more structured and testable python code.

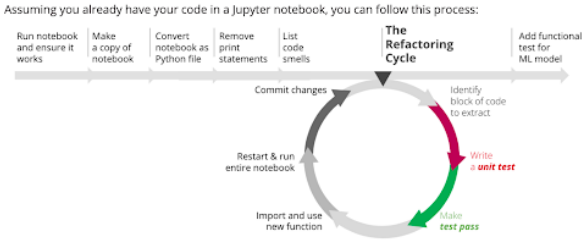


Figure 7.35: Refactoring cycle for notebooks

Continuous Delivery

"Continuous Delivery is the ability to get changes of all types—including new features, configuration changes, bug fixes and experiments—into production, or into the hands of users, safely and quickly in a sustainable way." [38]

In the context of ML applications, continuous delivery (CD) is all about making releasing your predictive pipeline a standard and ordinary activity that can be performed by a single action. The mindset required for this that our ML pipeline or predictive service is permanently in a "ready to deploy" state. No matter when the business wants to release the latest product version, the CD pipeline is ready to ship whatever is on master to production at the click of a button.

The leading version of the CD movement emphasises five principles:

- **Build quality in**:

  - This one will not make you happy. It requires tests. But not all hope is lost.
  - If you have at least a working e2e test that you run with the full dataset, that counts.
  - The idea here is to embed automated testing as upstream in the development process as possible, as well as building feedback loops that touch the customer to know what the customers like and dislike in our work.
- **Work in small batches**:
  - This principle is achievable even without testing.
  - The idea here is to have the ability to push every tiny change to the ML pipeline all the way to production everytime.
  - The less changes we have to push to prod, the less stressful it is.
  - It literally changes the economics of development, because it avoids having large fixed costs due to handing over the pipeline to a different team.
- **Computers perform repetitive tasks**, **people solve problems**:
  - If you have a dance that you perform every time you need to ship a piece of code to prod then automate all of it.
- **Relentlessly pursue continuous improvement**:
  - The idea here is that improvement is not a one and done project.
  - Everybody should be encouraged to treat continuous improvement as a vital part of their responsibilities.
- **Everyone is responsible**:
  - This last principle encourages ownership of the whole product, and to avoid local optimization.
  - The best way to motivate teams is to build fast feedback loops that report on the experience of the end-user.

Continuous Delivery is a rich topic that is actively being explored and refined in various locations on the web. CD4ML is a conceptual framework put forward here [34], here [35], and here [36]. Highly recommended.

Conclusions

Testing machine learning code is harder than it looks.

We went over the reasons why you should test your next ML pipeline. You probably didn't try out the TDML example. That's ok. You can take a look later. But, as it stands, to build a gut feeling of what to look out for, there is nothing better than trying to fix a bug in a pretend-prod ML pipeline.

We covered how ML code rots and how the rotting is accelerated because of the high dependency on data rather than just code. We talked about the benefits that tests bring to your productivity, mental health and how it can enable you to express fearless competence during your development activities.

We then explored the concept of self-testing code as a primary bug detection mechanism. We then linked the concepts of self-testing code with the TDD discipline that forces us to use the age-old double-entry bookkeeping method of writing tests before writing code.

Following that, we talked about the pyramid and the ice cream view of testing. This concept alone is worth the effort. The fact that we are able to catch the majority of the bugs in our software using an army of unit tests and only when they all pass do we spend the time and money on costly and slow system end-to-end tests is priceless.

We then did a deep dive into adapting to the ML domain, concepts such as unit tests, integration tests, contract tests, component tests, end-to-end tests, threshold tests, and regression tests.

We then looked at implementation techniques, such as test doubles and what tests to write and what tests to skip. Then we looked at property-based testing that allows you to throw a whole bunch of data at your code and see where it breaks. Then we dug into the sources of non-determinism in our code and how to handle some of the main randomness sources. We also looked at the goals of coverage, and we confirmed that it is not a quality metric but an untested code detector.

Subsequently, we looked at techniques to use to suffer less pain when external or internal forces are conspiring to make you skip writing tests. We covered how test expeditions allow for disciplined explorations. Then we looked at the benefits of synthetic monitoring to run QA in production. We then explored Feature Toggles and their relationship with A/B tests. Then we outlined the role of continuous delivery in helping you release your fixes faster.

In the last section, we looked at a brief sample of the solutions that great minds on the web have come up with to cope with this relatively brand new challenge of testing and building maintainable and testable ML products.

GOoD LuCk! I feel for you. I hope we keep evolving the practice without giving up on testing for ML code.

References

[1] David Heinemeier Hansson https://dhh.dk/2014/tdd-is-dead-long-live-testing.html
[2] Feathers, Michael. “Working effectively with legacy code.” Object Mentor, Inc. Available online at http://www. objectmentor. com (2002).
[3] https://en.wikipedia.org/wiki/Shotgun_surgery
[4] Robert C. Martin. Voxxed CERN 2019 • Keynote • Robert “Uncle Bob” Martin, https://www.youtube.com/watch?v
RkPU
[5] Martin Fowler. https://www.martinfowler.com/bliki/SelfTestingCode.html
[6] Clean Code: A Handbook of Agile Software Craftsmanship. Robert C. Martin. 2008. Prentice Hall PTR, Upper Saddle River, NJ, United States
[7] Cohn, Mike. Succeeding with agile: software development using Scrum. Pearson Education, 2010.
[8] Scott B. Alister, Testing Pyramids, https://alisterbscott.com/kb/testing-pyramids/
[9] Fields, Jay, et al. Refactoring: Ruby Edition. Pearson Education, 2009.

[10] Martin Fowler, https://www.martinfowler.com/bliki/UnitTest.html
[11] Martin Fowler, https://martinfowler.com/bliki/ComponentTest.html
[12] Martin Fowler, https://www.martinfowler.com/bliki/ContractTest.html
[13] Martin Fowler, https://www.martinfowler.com/bliki/ThresholdTest.html
[14] Meszaros, Gerard. xUnit test patterns: Refactoring test code. Pearson Education, 2007.
[15] Rails Conf 2013 The Magic Tricks of Testing by Sandi Metz, https://www.youtube.com/watch?v=URSWYvyc42M
[16] Hypothesis, Python package, https://hypothesis.readthedocs.io/en/latest/
[17] Raoul-Gabriel Urma, Kevin Lemagnen: Adv. Software Testing for Data Scientists, PyData London 2019, https://www.youtube.com/watch?v=WTj6T0QdHHM&t=2574s
[18] Escape from auto-manual testing with Hypothesis!, https://www.youtube.com/watch?v=U_-KhEi2vRT8
[19] Pytorch Authors, 2020, Pytorch Docs, Reproducibility, https://pytorch.org/docs/stable/notes/randomness.html
[20] Python docs, 2020, https://docs.python.org/3/library/contextlib.html
[21] Hendrickson, Elisabeth. Explore it!: reduce risk and increase confidence with exploratory testing. Pragmatic Bookshelf, 2013.
[22] Focus on Mean Time To Recovery, 2015, https://www.thoughtworks.com/radar/techniques/focus-on-mean-time-to-recovery
[23] Flávia Falé, Serge Gebhardt, 2020, Synthetic Monitoring, https://www.martinfowler.com/bliki/SyntheticMonito
[24] Pete Hodgson, 2017, https://martinfowler.com/articles/feature-toggles.html
[25] Martin Fowler, 2020, https://martinfowler.com/bliki/DarkLaunching.html
[26] Andrej Karpathy, Software 2.0 https://medium.com/@karpathy/software-2-0-a64152b37c35
[27] Breck, Eric, et al. "The ml test score: A rubric for ml production readiness and technical debt reduction." 2017 IEEE International Conference on Big Data (Big Data). IEEE, 2017.
[28] Testing and Deployment - Full Stack Deep Learning - March 2019 https://www.youtube.com/watch?v=nu7h1zdl
[29] David Tan, 2020, Coding habits for data scientists, https://www.thoughtworks.com/insights/blog/coding-habits-data-scientists
[30] Beck, Kent. Test-driven development: by example. Addison-Wesley Professional, 2003.
[31] Martin Fowler, 2018, https://martinfowler.com/bliki/IntegrationTest.html
[32] Sam Newman, GOTO 2017, Feature Branches and Toggles in a Post-GitHub World. https://www.youtube.com/watch?v=lqRQYEHAtpk
[33]Knight Capital. 2012. https://en.wikipedia.org/wiki/Knight_Capital_Group#2012_stock_trading_-disruption
[34] Danilo Sato, Arif Wider, Christoph Windheuser, Continuous Delivery for Machine Learning, 2019, https://martinfowler.com/articles/cd4ml.html
[35] Continuous Delivery for Machine Learning, 2020, https://www.slideshare.net/ThoughtWorks/continuous-delivery-for-machine-learning-198815316
[36] Continuous Delivery for Machine Learning Applications with Open Source Tools, 2019 https://www.youtube.com/watch?v=ub9XIgcUMAQ
[37] Martin Fowler, 2011, Eradicating Non-Determinism in Tests, https://martinfowler.com/articles/nonDeterminism
[38] Humble, Jez, and David Farley. Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education, 2010.