Agile Data Science with R

A workflow

Edwin Thoen

Contents

Chapter 1

Working without a Workflow

When I was starting my career as a data scientist, I did not really have a workflow. Freshly out of statistics grad school I entered the arena of Dutch business, employed by a small consulting firm. Between the company, the potential clients and myself, no one knew what it meant to implement a statistical model or a machine learning method in the "real" world. But everybody was interested in this "Big Data" thing, so we quickly started to do consulting work without a clear idea what I was going to do. When we came to something that looked like a project, I plunged into it. Eager to deliver results quickly, I loaded data extracts into R and started to apply all kinds of different models and algorithms on it. Findings ended up in the code comments of the R scripts, scripts that often grew to hundreds or even thousands of lines.

The only system I had was numbering the scripts sequentially. Soon I found myself amidst dozens of scripts and data exports of intermediate results that were no longer reproducible. The R session I was running *ad infinitum* was sometimes mistakenly closed, or it crashed (which was bound to happen as the memory used grew). When this happened, I spent hours or even days to recreate the results. Deadlines were a nightmare; everything I had done up to that point had to be loaded, joined and compared at the last moment. More often than not, the model results were different from that noted earlier, with no indication if I was mistaken earlier, I was using the wrong data now, or some other source of error was introduced. Looking back, I had no clue about the importance of a good workflow for doing larger data science projects. Several times I was saved when the plug was pulled from the project for other reasons, saving me from the embarrassment of not being able to deliver.

I have learned a great deal since those days, both from the insights shared by others and from my own experiences. Writing an R package to be shipped to

CRAN enforced me to understand the basics of software engineering. Not being able to reproduce crucial results forced me to start thinking about end-to-end research and model building, controlling all the steps along the way. Last year, for the first time, I joined a Scrum team (frontend, backend, ux designer, product owner, second data scientist) to create a machine learning model that we brought to production using the Agile principles. It was an inspiring experience from which I learned a great deal. My colleagues patiently explained the principles of Agile software development and together we applied them to the data science context.

1.1 What this Text is About

All these experiences culminated in the workflow that we now adhere to at work and I think it is worthwhile to share it. It is heavily based on the principles of Agile software production, hence the title. We have explored which of the concepts from Agile did and did not work for data science and we got hands-on experience in working from these principles in an R project that actually got to production. The story of how we created *Valuecheck* can be found as an Appendix in the final chapter. This text is split into two parts. In the first we will look into the Agile philosophy and some of the methodologies that are closely related to it (chapters 2 and 3). Both will be related to the data science context, seeing what we can get from the philosophy (chapter 4) and what an Agile machine learning workflow might look like (chapter 5). The second part is hands on. We will explore how we can leverage the possibilities in the R software system to implement Agile data science.

1.2 Limitations

Data science projects can greatly differ from each other. There are so many variables that make projects unique; its goal, its type (deriving insights, machine learning, building as dashboard), the data quality, and the expertise of the data scientist(s). This implies that there are necessarily aspects of data science projects experienced by others that I am not aware of. In this text I am relating my own experiences to the theory and best practises of Agile software development to come up with a general workflow. This means that I am probably “overfitting” the workflow on the dozen or so large data science projects I have done. If you feel that what I write is not broadly applicable, or if you think there are topics overlooked, please file an issue.

This text is meant to be a living thing with the objective of documenting a workflow that yields optimal reproducibility, quick shipping of results and high quality code. The more people share their best practises, the closer we get to this objective. Please follow along on this journey and get involved! Finally, I am not a native English speaker so fixed typos and style improvements are greatly appreciated.

1.3 Intended Audience

The title of this text has four components: *Agile*, *Data Science*, *R*, and *Workflow*. If you are interested in all four, you're obviously in the right place. This text is not for you if you hope to learn about different algorithms and statistical techniques to do data science; more knowledgeable people have written many books and articles on those topics. Also it will not teach you anything about R programming. The workflow I present is completely separate from the algorithms you choose and the data preparation tools of your preference, as it focuses on code organisation and delivery. If you use python rather than R, you will still find this text valuable, especially the first part, which focuses on workflow only and is tool agnostic.

The larger data science projects I was involved with all had the objective of delivering predictions in some way, so you can file them under machine learning. I intend to present a generic workflow that is also applicable to data science projects that have a different type of delivery, such as automated reports and Shiny applications. You might find machine learning a bit overrepresented in the examples and applications. If you think there is still a misfit between your daily data science practice, please let me know.

1.4 Contributors

The following people contributed to this text, by suggesting improvements or doing pull requests. Thank you!

Colin Fay(`@ColinFay`), Dilsher Singh Dhillon(`@dshelldhillon`), Jesse Tweedle(`@tweed1e`), Lorenz Walthert(`@lorenzwalthert`), Nathan Moore(`@nmoorenz`), Nic Fox(`@foxnic`)

Part I

Agile Data Science…

Chapter 2

Agile in a Nutshell

2.1 The Origins of Agile

Agile software development is not a specific methodology, a process, or a prescription for how to do your job. Rather it is a set of beliefs, a philosophy, that should guide a software development team in making optimal decisions. Agile was not created out of thin air, of course, it was very much a reaction to the then ubiquitous approach called Waterfall. In Waterfall large software projects are divided into specific stages, each stage should be completed before the next stage can start. Subsequently these stages are *problem analysis*, *designing the project*, *writing the software*, *testing it* and finally *implementation and maintenance.* Here you can find a clear and neutral introduction into the Waterfall approach. The goal of Waterfall is delivering complete and faultless software. The of gravity of a Waterfall project is the documentation, in which every requirement and aspect of the software is written down meticulously. The underlying conviction is that the software is written faster and is of higher quality when all aspects of it are decided upon upfront.

Projects done with the Waterfall method have a major pitfall, however. They can take a very long time to be completed. The combination of sequentiality and completeness might cause projects to last many years before software is delivered. Each time an error or incompleteness is found in one of the later stages, the project moves back to the previous stage to fix it. Because of the long duration of the entire process of Waterfall it occurred often that the end result was no longer a good fit for the ever changing market. The problem analysis might be done years ago and the problem has changed in the meantime.

2.2 The Manifesto

During the nineties several reactions to the cumbersome Waterfall came to being. A number of influential thinkers in the world of software development started to explore different ways of writing software. Alternative processes were suggested, such as Scrum and Xtreme programming. Eventually, in 2001 a group of seventeen came together in Utah and drew up the *Manifesto for Agile Software Development*. Their short 68 word statement is:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> Individuals and interactions over processes and tools
>
> Working software over comprehensive documentation
>
> Customer collaboration over contract negotiation
>
> Responding to change over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

If you think “well, that is rather vague”, it is on purpose. It is not a process you should follow or a methodology that prescribes how you should approach software development. Rather, they are core values that guide the development team with the many choices it makes along the way. At every crossroads the option that is most in line with these values should be selected.

Agile has a radically different approach to software design, by recognising a number of flaws in Waterfall thinking. First, it is impossible to think through all the aspects of a complex design and architecture before starting to write code. Software has to grow organically instead. Secondly, customers typically don’t really know what they want from the product, until they start interacting with it. Legally it might be a good idea to have all the aspects checked off before getting to work, but it will not keep the customer satisfied. Finally, customers and stakeholders will loose interest and the faith a project will be completed if it takes a long time before a working product is delivered.

2.3 The Twelve Principles

The Manifesto was accompanied by a set of twelve principles that flow from these values. They are more applicable than the four values and are thereby the principle guidelines when making choices. They are (numbering added by me):

1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4) Business people and developers must work together daily throughout the project.

5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6) The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7) Working software is the primary measure of progress.

8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9) Continuous attention to technical excellence and good design enhances agility.

10) Simplicity–the art of maximizing the amount of work not done–is essential.

11) The best architectures, requirements, and designs emerge from self-organizing teams.

12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

In Chapter 4 we will interpret them in the data science context.

Chapter 3

Agile Methods

We learned that Agile is not so much a workflow or a method, but a philosophy that guides us when choices have to be made. Agile promotes being critical to strictly following a workflow, we should continuously monitor if the way we work is optimal for following the Agile values and principles. If this is not the case, the workflow should be adjusted. This does not mean there are no workflows related to Agile. In fact, a number of workflows were developed by the men (yes, no women were there) who drew up the Agile Manifesto. Following these workflows should make it more likely the team works in an Agile way. Here we look into the two best known workflows; Scrum and Kanban.

3.1 Scrum

The most well-known and most-applied workflow is Scrum. Developed in the late eighties and early nineties it is a tried and tested methodology. Scrum works with sprints, set time units after which a shippable product should be ready. Most teams use two-week sprints, but they can also be shorter or longer. Teams are completely self-organising, they decide what they will do in the next sprint and how they will do it. Tasks to do are gathered on the product log, they are all formulated such that it is clear how they will add value to the product. These **user stories** take the form "As *role user* I would like to *action/functionality* such that I *benefit*". Say that you run a website that sends out a newsletter to all its customers, but there is no option for opting out yet. The user story for creating such a functionality could then be "As a subscribed user I would like to be able to opt out for the newsletter such that I only receive information when I want to".

3.1.1 Scrum Roles

The responsibility of the **scrum master** is making sure the team will make the sprint goals. To do so, she must have a watchful eye. If the sales manager wants something done and tries to persuade one of the developers at the coffee machine, he is kindly redirected to the product owner to get it on the back log. If some of the team members lack some necessary skills, she will make a plan with them how to acquire them. Also, she will be the organiser and facilitator of the different Scrum ceremonies.

The **product owner** is responsible for what the product looks like. He monitors the needs and desires of the customer, so one of its key responsibilities is stakeholder management. He translates feature requests and improvement plans into user stories. Thereby he maintains the product log, ordering the stories in how important they are.

Where the product owner decides what should be done, the **team** of developers decides how to do it. It is completely self-organising. At the beginning of each sprint it makes an assessment on which stories can be done in the upcoming sprint. Team members usually have different expertise, but the completion of the stories is the responsibility of the entire team.

3.1.2 Ceremonies

Four ceremonies (meetings) are part of the Scrum cycle. At the beginning of the sprint there is the **sprint planning**, in which the stories are scoped and the definition of done is determined. During the sprint there is at least one **stand-up** per day, in which team members quickly share what they are working on and what they need from each other. When the sprint is done the team organises a **sprint review** in which it presents to people who are not on the team what work was completed. Finally, there is the **retrospective**, in which the team discusses what went well during the last sprint and what can be improved.

3.2 Kanban

Kanban is much lighter and less process heavy than Scrum. It does not work with fixed time units, such as the Scrum sprints, but it also aims to achieve a continuous flow. Just as with Scrum the tasks to do are formulated in user stories, but the commitment is just to one story at a time. Stories are gathered on a back log and are continuously ordered in importance. Each time a story is completed the most relevant or pressing story is started next.

Central is the Kanban board, which can be physical or virtual, that at least has the columns *to do*, *doing*, and *done*, but can be tailored to the wishes of the team. Unlike Scrum there is no official role of Product Owner, still it can be useful to

have someone handling the incoming requests. This can be a designated person or a team member who is also doing development work. The team should not focus on too many tasks at once, everything that is pulled from *to do* should be finished as quickly as possible. This assures that the focus is always at the most important task ahead and there is minimal multitasking.
A team can set a cap on the number of stories that can be in each column.

The central metric is the amount of time it typically takes to complete a task, the *cycle time*. Effective Kanban teams have short cycle times, they are able to complete the tasks quickly. They can give estimates when work will be done with confidence. Just as with Scrum the entire team is responsible for completing a story, not just the “designated” person for the job. In order to have the tasks completed as quickly as possible team members might fulfill tasks now and then that are a little bit out of their comfort zone.

3.3 Scrum vs Kanban

The highly structured Scrum and the lightweight Kanban are two workflows that could make a team work more according to the Agile principles. They both aim for continuous shipping of working software instead of working towards one major release. They also both give focus on the part you are working on right now, Scrum by fixing the stories that are done in the sprint, while Kanban limits the number of stories that the team is working on. But there are also some remarkable differences. In Scrum the team commits to the stories it selected to do this sprint and the building has to be on fire before it will take on work that is not in the sprint. Kanban only prescribes to limit the number of stories that are in work in progress, finish what you start first then start something new. However, for the story to be done next everything can change at any moment. Scrum is quite heavy on the ceremonies, while Kanban does not prescribe recurrent meetings.

Both methodologies are applied with great success and it’s important to keep in mind that they are a means to an end, not religions. The Agile values and principles should be the primary guideline and when selecting one of the workflows you do so because it is the best way to work in an Agile way in the team’s situation. The team should decide for itself what is the best way of working and should monitor if the choice is still the best as the situation changes. In general, however, it makes more sense to use Scrum when a team is working on the completion of a project, whereas the flexibility of Kanban is best suited for a service team that is dealing with a lot of incoming requests.

Sources:

https://www.youtube.com/watch?v=2Vt7Ik8Ublw https://agilescrumgroup.nl/product-owner-rol-taken/ https://www.atlassian.com/agile/kanban/ https://agileknowhow.com/2018/03/01/do-we-need-a-product-owner-with-kanban/

Chapter 4

Agile Data Science

4.1 Data Science *Waterfall*

As discussed in the previous chapter, Agile is a response to Waterfall methodology that was widely adopted in the eighties and nineties. Many projects using Waterfall did not deliver, because of their long duration. In data science, as far as I am aware of, there are no such formal methodologies that are followed by many practitioners. However, there are many data science projects that do not deliver. Ineffective workflow is by no means the only cause of this, but I am convinced it is one of the main culprits. Just like software development in Waterfall, data science projects can take many months or even years before the results are productionised. Apparently, more often then not the plug is pulled before this stage is reached. Data science is especially susceptible of what I call *local optimisation*. The data scientist might optimise endlessly to give the best predictions possible or have all the envisioned dashboard functionalities in place, before sharing the results. As a result, stakeholders might loose interest and confidence in the successful completion of the project. Moreover, if will cause defects in the project setup to remain hidden for a long time. The code might be poorly organised, making the product unfit to be applied in the "real world". Or there might be unclarities on what to predict or which data sources are in scope, due to lack of communication between stakeholders, business people and data scientists. Whatever the reason, adhering to the principles of Agile can get you more productive and efficient. Here we take the time to interpret the twelve principles in the data science context.

4.2 The Twelve Principle in the Data Science Context

1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Just as Waterfall prescribes a complete and fault-free product delivered at once, data scientists might be inclined to only release a machine learning model to production once they are confident its predictions are spot on. This principle was a revolutionary break from Waterfall, you should not wait with releasing software until it's perfect. Instead get it out in the open when it is just good enough. It is called the MVP (*Minimal Viable Product*). After the MVP is released it is closely monitored for how users interact with it and where the biggest room for improvement is. The team tackles the biggest problem in the MVP to create a slightly better version of the product, which is again released right away. This cycle of release, monitor, improve, release is repeated many times, such that the product gets better and better. There is no clear definition of done, instead there is debate on if the software can be further improved and if the required investments are worth the effort.

The machine learning equivalent to this would be a *Minimal Viable Model*, a model that is just good enough to put into action. Other data science projects, such as reporting, ETL and apps, might have more software elements in them and less data uncertainty. For these an MVP can be defined. Releasing something that is barely good enough might be scary and counter intuitive to the high standards you hold for yourself, but it is preferable over long optimisation before releasing for at least the following reasons:

- *It will keep stakeholders excited.* Managers and users of the model who commissioned the data science project are anxious to see results. As the projects drags on without any output they are likely to lose interest and confidence the project will end well. Eventually they might pull the plug or put it on hold before anything meaningful has come out. If they can interact with the results soon, even if they are from perfect, their enthusiasm and confidence will remain high.
- *You will fail fast.* There is a wide array of reasons a data science project might fail; the business problem appears not be translatable into data science problem, data quality it low, there is not enough historical data, or the necessary relationships simply don't exist. If there are such problems, you will not be able to create a MVM. This is a clear sign the project is probably not going to be successful, and you can consider stopping it early.
- *You will get feedback sooner.* Lets say you build a churn model which the sales department uses for customer retention. You agree with stakeholders what the model should look like and start to build an MVM. As soon as they start using the MVM they find out that the prediction interval is too short, some customers already have a contract with a competing party at

the moment of interference. Instead of further optimising this model, you first change the model structure so it predicts a longer time ahead. Or the user of Shiny app realities that the ratio you agreed on is not what she wanted as soon as she sees the app with the first variable implemented. Before including the twelve other variables you first fix the ratio such that it is as she wants it.

What the MVM looks like is project-dependent of course, but it might make sense to define it in terms of regular model quality statistics, such as recall or mean absolute error. In some situations there might be a natural baseline that the MVM should outperform, such as a business rule that the model needs to replace. Data science products that are not model based might be captured with an MVP, instead of an MVM. For both an MVM and an MVP a possibility is only releasing the product for a subset of your target audience, such as a geographical area or users of a certain age.

2) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

This principle comes naturally to data science, since the success of a project typically dependent on the data relationships discovered during the project. The Waterfall approach in which every step of the project is planned would be hideous to even the biggest lover of process in the data science context. Keep in mind that flexibility should not only be exercised towards assumptions of your data or the models and algorithms you use. Requirements can also be in the framing of the business problem or the way the results are exposed. The very reason we are releasing early is that customers can put it to the test and through interaction they discover what they really want. Developing the product is a discovery for them as much it is for you.

3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Whereas the first principle is about the philosophy of early deployment and iteration, this one is about the frequency of deploying updates. The Scrum framework is really strict in the amount of time that can be spent until the next release. The team commits itself to making certain changes to the product in typically a two-week period. At the end of this period the improvements, small as they might be, are deployed. The Scrum mindset is not totally applicable for data science, especially when there is a lot of data uncertainty, as we will explore in the next chapter. However, it is good to keep in mind that every improvement to the model should be deployed as soon as it is ready. This creates momentum and excitement by customers, stakeholders, your teammates and yourself.

4) Business people and developers must work together daily throughout the project.

Too often data scientists are operating in isolation. Certainly when projects are of a more explorative nature. Having no connection to the business makes it unlikely your work will ever affect anything outside your computer. Stakeholder management can be done by a representative of the technical team, such as a product owner. If there is no such role on the team, the stakeholder management is the responsibility of the data scientist. A second way the business should be involved is for getting information about underlying processes. Unless the data scientist has extensive knowledge of the business, she needs somebody to validate the found results and answer questions about surprising finds. Finally, you might need to involve a person with technical knowledge of the data, typically a DBA, who can explain the structure and peculiarities of the data you work with.

5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

This principle is the antithesis of Waterfall in optima forma. Instead of meticulously describing how the job should be done, just set the goals of the projects and leave it up to the team how these goal are attained. Data scientists typically already enjoy this type of freedom for the sheer reason that stakeholders don't really understand how the products are built. It can happen that business people get overly involved in the process, they can have a strong opinion on which predictors should be used or how the target should be defined or what the best way is to visualise something in a dashboard. Take their advice at heart but also trust your instincts. If you feel a different approach will yield better results than rely on your expertise. You know about overfitting, multicollinearity, non-converging algorithms, theory behind data visualisation and many other topics the business has no intuition for. If you think your approach is better, take the time to explain why you think a different approach is better (in lay men terms of course).

6) The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

A data science project is rarely done end-to-end by one single person. Data might be made available by a DBA, a back-ender might expose the product on the website, the front-ender builds the interface for interacting with predictions, etc. If possible working with these people directly will speed up decision making and improve alignment. Communication by email or chat programs are often slow. Make an effort to be in the same room with your direct colleagues, for at least a part of the project time.

7) Working software is the primary measure of progress.

If your product did not leave your system you have not attained any results yet. As long as your Shiny app only runs on your laptop, it does not exist. As long as the machine learning model predictions are not reaching someone, it does not exist. You have not added any value as long as your product is not "out there". Only when the update to the predictions is fully implemented and the predictions are ready to be consumed by the business, has there been true improvement. Only when you have shipped your MVM app to the company server and Sales is basing decisions on it, you have delivered something. Sometimes reported model improvement discovered in research scripts does not hold when it is implemented in the full model pipeline. It has been done on just a subset of the data that was conveniently available, or the new feature was tested in isolation and there is not yet a sense of multicollinearity. There is only one true measure of how well we are currently doing, and it is what is currently exposed to others as the product.

8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

The deadline way of doing data science; the stakeholders and the data scientist meet, they agree to have a first version of the project ready at some future moment. The sponsors forget about the project until right before the deadline, busy with attending meetings and writing memos. The data scientist goes to work, having ample time before the deadline there are many things that can be explored. The result is an array of research scripts and intermediate results. Suddenly, as the deadline comes near, all this separate research has somehow come together. Pulling an all-nighter he is able to deliver a result, which is presented to the sponsors. The project is then continued, a new deadline is set, and the cycle starts over.

Don't do deadlines!

They are a recipe for hastily created, non-reproducible results. They promote a workflow of taking it easy at first, stressing out when the deadline comes near and exhaustion after it. Instead set small goals that are quickly attainable, update the product if the results are favorable and set a new small goal. This will result in better quality code, reproducibility and a happier team. Moreover, it will result in a product that is constantly improved, which excites sponsors and users.

9) Continuous attention to technical excellence and good design enhances agility.

Most data scientists have much to learn from software engineers as it comes to standards and rigor. What makes data science unique is that a part of the code is written for personal use. It is not meant to ship in the final product, it will never leave the data scientist's system. Still it is very much part of the project. Data cleaning, splitting into train and validation sets, running algorithms that produce the models, doing research on relationships in the data and many more steps are often for the data scientist's eyes only. It is tempting to cut corners when you are the sole user of your own code. Why go to the trouble of writing unit tests and documentation for your functions?

Continuously improving your product in short-cycled deliveries is only feasible if the entire product is of high quality. If you want to expand a poorly designed product, it is much more likely the whole thing comes tumbling down as soon as you start to make adjustments to it. There will be a lot of attention to this topic in the second part of this text, because you simply cannot be Agile if your code is of poor quality.

10) Simplicity–the art of maximizing the amount of work not done–is essential.

A machine learning project's goal is often straightforward, predict y as best you can such that some business goals can be achieved. Unlike software development there is not much choice in which features should and should not be included in the final product (features as in characteristics, not as in predictors). The options how to arrive at predicting y, however, are abundant. The biggest challenge is often "what should I explore next?". Should we explore another database in which we might find new predictors or should we try a different model on the current predictors which involves some additional preprocessing of the data?

We can roughly estimate what the amount of work would be to explore both options. It is, however, very hard to predict what the amount of value is the new part will add. A good rule of thumb is that when in doubt choose the option with the least unknown components. Choose an algorithm you know well over one you have never used in practice. Only tap into an unknown data source if you are convinced that the options on the well-known database are exhausted. Data science is a field with rapid developments, it is often tempting to seize the opportunity to dive into a new technique or algorithm. Be critical before doing so, is there really no way to obtain similar results with something already familiar to you?

11) The best architectures, requirements, and designs emerge from self-organizing teams.

This is another principle that is a clear antidote to Waterfall. Instead of meticulously planning every aspect of the project upfront, let the developers come up with the most important project designs as they go. It is impossible to foresee all the aspects of the software project before implementing it, so trying to come up them with before writing code is a guarantee for going back and forth between the planning and implementation stages. Due to the iterative nature of building data science products and the uncertainty we have upfront, this principle seems quite natural in our context.

12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Every development process has its inefficiencies, whether they are unclear goals, poor communication or not having the right priorities. Reflecting on the process forces you to look critically at all aspects of the project. Inefficiencies can quickly become project features when they exist for a while, the sooner they are tackled the better.

Even when you are not in a team following an official methodology such as Scrum or Kanban, you do best in planning regular reflection meetings. Even when you are the only data scientist or even the only developer on the team, you should also address your technical issues here. Maybe you are wanting to refactor a certain part of the project for a while but are unsure if it is worth the time. Business colleagues might not understand the technical aspect of the problem, but they can still challenge you on your choices.

4.3 In Summary

After this lengthy point-by-point exploration let me wrap-up to what I think is the essence of Agile Data Science:

Deploy products as early as possible, don't loose yourself in endless explorations. Update the product continuously, instead of saving everything until some deadline is near. Hold high standards for all the code used by the product, also if it doesn't get shipped with the product. Have stakeholders, colleagues and customers closely involved the whole way, don't retreat.

Chapter 5

A Methodology for Agile Data Science

Now we have interpreted the Twelve Agile Principles in the data science setting, we explore what an Agile data science workflow might look like. Let us remind ourselves that an Agile workflow is always a means to an end. The Agile values and principles are the guidelines and the workflow should serve following the values and principles the best you can. If at any moment in a project the team feels the workflow is no longer the optimal way to make decisions in an Agile way, it should change it. This chapter should be considered as an exploration, a bunch of thoughts. If some of it does not work for you for whatever reason, by all means find a better way.

5.1 Linear and Circular Tasks

The tasks in Agile software development are what I call linear tasks. They come from product feature requests by stakeholders or ideas within the team itself, collected by the product owner. The envisioned outcome is captured in a user story. The team translates it into the technical tasks and starts working on it. Neither Scrum nor Kanban prescribes the steps a task should go through, but it typically looks like the following.

These type of tasks lend themselves well for scoping and committing oneself to what the product will look like in a few weeks time, as is done in Scrum. (Note that it is the tasks that are typically linear, the entire process of building a product is iterative by nature). In contrast are data science tasks are circular, at the start of a task it is uncertain what the outcome will be. The starting position is the latest version of the product. If one has not yet a good idea how to further improve it we would typically do an explorative analysis in which a

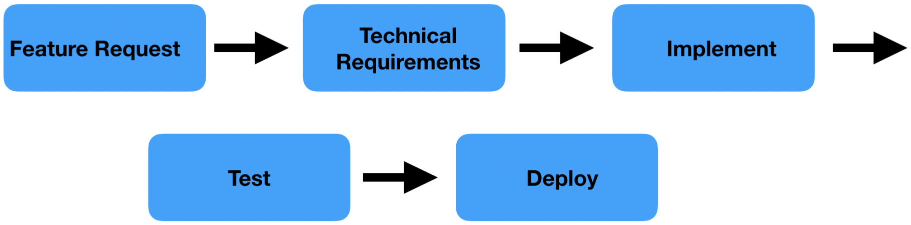
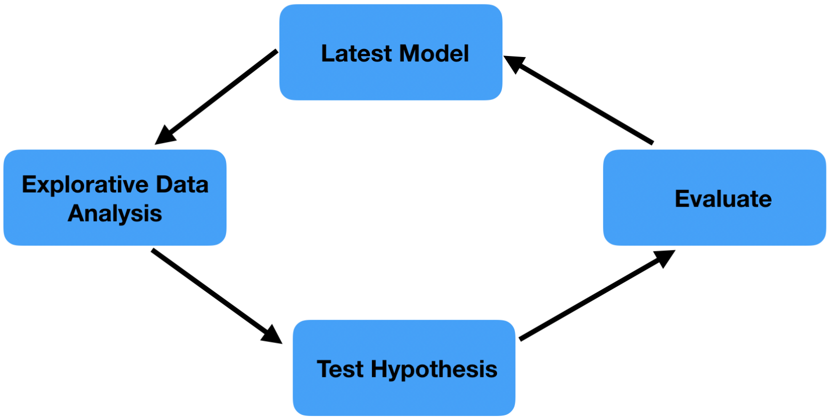


Figure 5.1: Linear flow in software development

hypothesis can be formed. This hypothesis is then tested. We evaluate if we can improve the product based on the hypothesis tested. If not we go on with the next hypothesis right away, if so we improve it and then go on with the next hypothesis.



Data science projects also encompass linear tasks, such as setting up a pipeline to import the data and do basic data wrangling, creating apps or exposing the model results in an API. However, the circular nature of leveraging relationships between variable makes the highly structured Scrum method unfit for most data science projects. We simply cannot guarantee that a statistical model or a machine learning algorithm will be improved in two weeks time, because we don't know if the hypotheses we are going to test will lead to anything. If we had committed to a set of tasks for a fixed time period from which we cannot deviate, we are slowed down because we cannot directly act on newly gained insights.

Data science encompasses a broad array of project types. Maybe some don't have circular tasks at all and are basically software development projects. Examples might be building an app on a known data source or automating reports to be sent out every month. For these type of projects it might make sense to follow a Scrum, cutting up work into user stories that are implemented in fixed time intervals. Most data science projects, however, will have a mixture of linear and circular tasks. For any project with circular tasks the flexibility of Kanban is preferred over the highly organised Scrum. In the remainder of this text we assume projects to be at least partially developed using relationships that are unknown at the start of the project, and thus contain circular tasks. If this does not apply to your projects, you might find literature on regular Agile software development more appropriate.

5.2 A Two-Way Workflow for Development

I propose a two-way workflow for data science product development, in which circular and linear tasks are combined. This workflow makes a hard cut between the *data product* and *exploratory research*. The data product is software, and thus built by linear tasks. When starting a task it is clear what should implemented. The data product is complete, which means that it contains all the necessary steps to go from source data to the end result. It can be a pipeline that contains loading, wrangling, preprocessing, modelling, and exposing steps. Or it can be the querying of different databases, joining the different data, calculating statistics and building plots, wrapping everything in a Markdown report. No matter what the product looks like, it should be high quality software so we can rely on it. This will make results completely reproducible and automatable, the two requirements for continuous delivery of improved products. Exploratory research on the other hand can be quick and dirty. In order to test hypotheses quickly, exploration scripts can be interactive analyses without software requirements or even being reproducible in later stages. They should quickly give an indication if the tested hypothesis could improve the product.

5.3 Formulating Tasks

In both Scrum and Kanban the tasks ahead are formulated in user stories, clearly stating what the benefit for the customer will be once the user story is completed. If you try to define a user story for a machine learning model it will always go "As the I want to have the best possible model, such that is achieved best". For a data science project user stories are not of much value, I think. Still it is valuable to clearly separate your work into dedicated tasks. Tasks are either updating the product or exploring new ways for improving it. From its definition it is clear what part of the code can be touched by it and which not.

5.4 Using Kanban for Data Science

We have concluded that Scrum is too rigid for a data science project with circular tasks, because the explorative nature of data analysis is not suitable for the tight planning of deliverables. Kanban on the other hand gives us the flexibility to change the next task we are going to complete. Within a two-way model for doing data science there is the data product that has to be good quality software and there is the explorative research in which you can do whatever to come to quick conclusions. You could formulate tasks for the Kanban workflow for both task types. Weaving these two types of work together results in a Kanban workflow with at least the columns *to do*, *doing*, and *done.* Both hypotheses to be tested and planned work on the data product are gathered in the *to do* column. This is the backlog and it is always ordered from most to least promising, so it is clear what to do next.

Kanban gives focus, finishing one task at a time. Too often when doing data science we have interesting finds on which we jump right away without finishing what we were doing. To prevent that, just add the new find as an hypothesis to the board. This will make sure that the tasks that are currently in *doing* gets completed first, and that after each completion there is a moment where it can be decided what is the most urgent change to the pipeline or the most promising hypothesis to explore next. As a rule of thumb, never work on more tasks simultaneously than that there are data scientists on the team. As mentioned, tasks are either software or research tasks. If the research task results in a proposed model update, the update can be captured in a newly formed software task (which can be placed on top of the *to do* list right away).

5.5 Scoping Tasks

Scrum uses story points to scope its stories. The team determines the number of points awarded to each story. The team knows how many story points it typically completes in a sprint, so after scoping the sprint can be planned by selecting stories such that the total of their points does not exceed the team's capacity. Kanban does not scope stories. In fact, the average time of a tasks completion is Kanban's key metric of effectiveness. When doing data science with Kanban it might still be valuable to scope the tasks ahead, especially for exploring hypotheses. One of the major pitfalls of trying to improve a data science product is endless exploration of an hypothesis. We like to have just one more look from this other angle, or maybe this new fancy algorithm that you are anxious to test for a while will give a major boost. Data scientists are typically assiduous people, this is what allows them to master a wide range of difficult topics from statistics to programming in the first place. However, this could lead to stubbornness, unwilling to give up what was thought the way to get a major improvement. Scoping for data science is then not just estimating how long a task will take to complete, it is also time boxing. If used in this way, the scoping should be done in time units, not in a subjective measure such as story points. The data scientist should not take longer for the task than the team agreed upfront, wrapping up even when he does not feel completely finished. If he found an alleyway that is still worthwhile exploring a new task should be put in the backlog, instead of persevering in the current task. Scoping also helps with prioritising. If there are several candidate tasks to do next, the one with the least time to complete might be best done first.

5.6 The Product Owner Role

When doing software development with Kanban there is typically a product owner involved. She aligns with customers and stakeholders, and adds the feature requests to the *to do* column of the board. For data science engineering it can also be desirable to have someone other than the data scientist doing stakeholder management and communication of the model results. This will free up time and energy for model development. Gathering the tasks to do cannot be primarily laid at the product owner. The data scientist will probably post most of the tasks on the Kanban board, because both hypotheses to test and maintenance work on the data product typically require an in depth knowledge. Product owners can add feature requests to the data products, especially when the product has a large software component, such as a Shiny application. You should discuss the tasks you put on the Kanban board with the product owner, even when they are technical. This will demystify the model building and makes sure she can do a better job explaining the work to stakeholders. Especially when you are the sole data scientist on the project she also needs to get involved in prioritizing and scoping. Discussing how much time it will cost to complete the task and what it would bring can lead you to more accurate estimates of the time and value of the task. Also, the product owner might raise concerns from the business side that you did not think about, leading to a different prioritisation.

Part II

. . . with R

Chapter 6

Code Organsation

Continuously delivering an updated product can only be achieved when the code is of high quality. Many data scientists, especially those who are drawn to R, have a background other than software development. Writing code is something we do to solve problems in our professional field, such as statistics, ecology, or actuary. The code is typically written interactively with the data. Write a few lines of code, run against the data set at hand, check if the results match expectations, write a few more lines of code. This approach might be fine for a quick explorative data analysis in which you write code to answer a few ad hoc questions. It does not work for large scale projects in which the data product should be reliable and stable, because this way of working results in low reproducibility and inflexibility towards new situations.

Reproducibility is low because code is not developed as part of an end-to-end product. We could call it *random walk programming*, the next step is determined only by the current state of the code and data. Code developed this way only keeps working when upstream code and data stay exactly the same. The larger the project becomes the less likely this condition is met. You can say that we are *code overfitting* on the data. The code only works in a particular situation, as soon as there is a small change the code breaks. Oftentimes, this way of developing is combined with saving intermediate results, to prevent having to run all the code time and time again. Together they can be a reproducibility recipe for disaster. If the code never runs end-to-end, it can quickly become unclear which intermediate results are out of sync. We no longer know which reported results are obtained on which data set and if this was before or after some modifications to the code. Working in this way will create uncertainty, stress and an unreliable product. We cannot build an Agile workflow on such a basis.

6.1 Using the R Package Structure

To remedy this high stress, low reproducibility workflow we should turn to the best practices of software development. Rather than an endless exploration, the data product should be an integrated software package that works on all possible input data, is reliable and does not contain redundant code. Software development is not foreign to R, fortunately. It is provided in the package structure that enables users to store their own function and share them with others. (Although R has ample opportunities for doing object oriented programming, the *ADSwR* approach to code organisation is to build a data product that solely consists of functions.) Lower level functions are combined to form more complex operations. These operations are then chained together to create the end product. If your product has some kind of flow in it, such as the scoring of a machine learning model of turning last month's data into a research report, you can apply the functions in a pipeline. In the subsequent chapters we will explore several aspects of building a high quality pipeline, that gives reproducible and reliable results.

In the previous chapters it was concluded that the circular nature of explorative data analysis makes data science projects fundamentally different from 'regular' software design. Whereas the data product should be of high quality, exploration should be done quickly to indicate if we can advance the project by incorporating the results of the analysis. R packages have Vignettes, documents in which the package authors show how to use the package, such as this one. One of the ways of creating such a Vignette is by using Rmarkdown documents. These happen to be ideal for data exploration, combining code in the code blocks with comments and observations in the text around it. Because Vignettes are part of a package, all functions developed for the product are readily available for doing research as well.

6.2 Further Reading

Doing Agile Data Science with R relies on a good understanding of the R package structure. Hadley Wickham wrote a comprehensive book on developing software in R packages. It is freely available online, here. You should read it!

(It was brought to my attention that an updated version, co-authored with Jenny Bryan, will be released soon. You find it here https://r-pkgs.org/)

Chapter 7

Automation

R users tend to run code by hand. Press a button, see results, press some more buttons, see some more results. R is designed with interactivity in mind, it enables the user to have a direct Q&A with the data. This is an unparalleled feature of the R language, making it uniquely suitable for drawing quick insights and conclusions from data. A data product, however, should not be run by hand, it should be running automatically. From start to end no human should be needed to obtain and expose results. There are at least three advantages of automation.

7.0.1 Efficiency

The obvious reason data science products should be automated is that it will save a lot of time. It takes extra effort to set up an automated pipeline, but once it is in place only monitoring and maintenance are needed. If instead each report or each model update should be created by manually running the code, the data scientist should always schedule time to do so. This is not only a waste of time, but also a waste of focus. Once you have reached a stage that the product is of satisfactory quality and can run without interference for a time, you should be able to move on to a new project without thinking about the completed one too much. However, if the product is run manually you have to keep spending a part of your attention to the completed product. Continuously switching between projects is cognitively strenuous and not satisfying.

Not automating also creates a risk for the continuity of the product. It is not healthy to be completely dependent on the knowledge and actions of a single data scientist, because it is not guaranteed they will continue to work at the organisation for the duration of the product. A well automated product can be maintained by anybody with the right skills. If the project has to be continued by a different person, it will be a lot easier to maintain the product when automation is in place.

7.0.2 Automation is Documentation

A data science product is software, not random walk code. This means it is built bottom-up, small, unit-testable functions are used together to create more complex actions. The higher-level functions can then be used together in even higher order functions to create specific parts of the pipeline, such as data cleaning, preparing features or training a model. It does not, and should not, matter where each of the lower and higher level functions are created. As soon as you load the R package into memory all the functions are available. The functions are created to be applied in a certain order, of course. For instance first you load the data, then you clean it, prepare it and apply an algorithm on it. By completely automating the product, it is immediately clear how everything should play together.

7.0.3 Reproducibility

You can't be Agile without being completely reproducible. Results on the same input data should be the same each time the product is run, independent of when and by whom it is done. Small bugs in the code or unexpected values in the data can be compensated for on the fly when the code is ran manually. This is undesirable because this implies that the same problems will keep popping up in future runs. Even if you are the sole data scientist on the project, you might forget how you resolved it the last time. Rather, bugs and unexpected values should always be systematically resolved. This means that as soon as you hit something, you have to invest time in finding the source of this error, rather than quickly fixing it and moving on. As with the good practices discussed in the next chapter, not cutting corners will cost more time and effort in the short run but will pay off in both time and confidence in the product quality. Automation is a great way to enforce yourself to be completely reproducible, results will only be produced when the product is fault free.

7.1 How to Automate

We have discussed why we should automate, but not how to do it. You can build one meta function that comprises all the intermediate steps of the product. Calling the function will set everything in motion, capturing intermediate results and sending these to the next pipeline step. But as usual in R, someone else has already done the hard work for you. I suggest using Will Landau's fantastic `drake` package, instead of figuring out how to set-up the pipeline yourself. It caches data after the completion of each step, so when you make changes to the product you only have to rerun the parts that are affected by the change. Having all the intermediate data automatically stored has the additional benefit that you have them readily available when doing research for model improvements.

Chapter 8

Code that Fails

Your code will fail. Not once, not twice, but hundreds, thousands of times over the course of a large data science project. Everybody's code fails, many times. Your code, my code, Hadley Wickham's code. Not because we are bad at what we do, but because we cannot foresee all the possible ways the code will be used or all the possible data inputs that the code will be confronted with at the moment we write it. Getting better at programming does not mean you will write fault-free code (although you will make fewer errors as you grow more experienced), but writing code that fails fast and informatively. Writing such code means that you will spend more time developing than with the *random walk* method described in the previous chapter. This is an investment that will pay off big time as your project grows in complexity.

But before we look into how to write code that fails fast and informatively, why is this important for Agile Data Science? High quality code is part of the twelve principles, but this is a means to an end. The essence of Agile is continuous delivery. Updating and shipping your product every time you make progress can only be done when your code is adjustable. As the project advances, the end-to-end product will grow in complexity. Sooner rather than later it will reach a point in which you cannot have a full overview of all the aspects of the product anymore. When introducing a new feature you want to make sure all the other elements in the code base keep doing what they were designed for. If not, you want to be clearly informed of what is going wrong so you can either adjust the newly introduced feature or modify the existing elements so it works with the new feature. If every time you want to make a change the whole thing comes tumbling down without informing you what goes wrong, you cannot achieve the objective of fast, continuous delivery.

8.1 Assumptions to your Data

Automation means not calling functions yourself anymore. You just flip the switch and the whole thing is set in motion. This means that you no longer get the immediate feedback you are used to have when calling the functions yourself. When something goes wrong the whole thing just breaks with an error message. When you are lucky the error message is informative and you are quick to find the bug. When you are not, you may be confronted with `object of type 'closure' is not subsettable`, and you may have a grumpy afternoon ahead. Best to not depend on luck, but make sure you will always get a proper indication of what went wrong.

8.1.1 Type Checking

Most languages using functional programming are strongly typed. This means that for every argument that a function takes, its datatype is specified at function creation, as well as the datatype that the output takes. When the function is then called it is first checked if the data on which it is called is of the correct type. If not, it will break immediately and will tell the user why. R is weakly typed, the data type of the function arguments are not specified. When a function is called it just has a go on the objects that are fed to it. The function only breaks when somewhere in the body another function gets called with an invalid data type. Take the following for instance:

```r
add_two_numbers <- function(x, y) {
  print("Reaching this")
  print("and this")
  print("Doing some random other stuff")
  x + y
}
```

Now, what will happen if we accidentally call it on a string? It will only break when we hit the `+` operator, all the code before it runs. The error message it gives us is not so informative that we immediately figure out what went wrong.

```r
add_two_numbers(42, "MacGyver")
```

```
## [1] "Reaching this"
## [1] "and this"
## [1] "Doing some random other stuff"

## Error in x + y: non-numeric argument to binary operator
```

Pfff, what is a binary operator again? When this function is part of a framework in which higher order functions call lower order functions you may be up for an hour or two of sifting through your code to locate the bug. Fortunately type checking is very easily added to a function by asserting all argument data types in `stopifnot`.

```
add_two_numbers <- function(x, y) {
  stopifnot(is.numeric(x), is.numeric(y))
  print("Reaching this")
  print("and this")
  print("Doing some random other stuff")
  x + y
}

add_two_numbers(42, "MacGyver")
```

```
## Error in add_two_numbers(42, "MacGyver"): is.numeric(y) is not TRUE
```

We are now specifically informed which argument does not meet the assumptions and what the name of the sub function is.

8.1.2 Column Validation

Central to most data products will be the modification of data frames. Most functions you'll create are very specific to your project and will be called only once. For these cases you probably want to save yourself the overhead of making each function completely generic by parameterising the column names that are used. Say you want to add the log of the target to the data frame and you create the following function:

```
add_log_target <- function(x) {
  stopifnot(is.data.frame(x))
  x$target_log <- log(x$target)
  x
}
```

Now what if `target` was mistakenly removed from `x` in the step preceding this one in the product? Will it throw an informative error?

```
add_log_target(mtcars)
```

```
## Error in log(x$target): non-numeric argument to mathematical function
```

Uhh, no, not exactly. It gives the impression we tried to call the function on a non-numeric object, while actually the object is completely missing. It is a good idea to check if all the columns that are required for the operations in the function are present before starting them. I use this little function:

```
data_frame_with_col <- function(x, cols) {
  parent_frame <- sys.parent()
  calling_function <- sys.call(parent_frame)[[1]]
  stopifnot(is.data.frame(x))
  if(!all(cols %in% colnames(x))) {
    not_present <- setdiff(cols, colnames(x))
    stop(paste(not_present, collapse = ", "), " missing from the data frame in functio
  }
}
```

which is added to each function that uses a data frame

```
add_log_target <- function(x) {
  df_has_cols(x, "target", match.call())
  x$target_log <- log(x$target)
  x
}
add_log_target(mtcars)
```

```
## Error in df_has_cols(x, "target", match.call()): could not find function "df_has_co
```

8.1.3 Data Validation

Where the first two checks are about the type and structure of the inputs, data validation tests the assumptions to the data itself. Variables that cannot contain missing values, variables that must be strictly positive, or characters that can only take a limited number of values. If you have any such assumptions in your data, you do best to check them before you unleash internal functions on it. You could use external packages such as `validate` or `recipes`, or you can write them yourself. The `stopifnot` function can be used to check any assumption you have, as long as you can create a single logical for it. In the example below, since we are taking the log of `target` and it is the target variable it must be strictly positive and non-missing.

```
add_log_target <- function(x) {
  df_has_cols(x, "target", match.call())
  stopifnot(all(!is.na(x$target)), all(x$target > 0))
  x$target_log <- log(x$target)
  x
}
```

8.2 Unit Testing

The above type checking, column validation, and data validation, test assumptions to the data going into the functions. To data scientists learning about these it usually makes a lot of sense, because most who worked on a larger product without these have been bitten quite a bit. Moreover, these measures are quick to implement by just inserting one or two lines of code at the beginning of a function. Unit testing your code does not come so naturally to data scientists, unless they have a background in software development. Writing unit tests might seem as a pointless and time consuming exercise at first. This is because its benefits are not immediately obvious to someone who never applied or used them. Still, you should make it part of your software development routine, even when you are the only one using the software you write.

If you are unsure what unit tests are or how to do them in R, please read the chapter in *R packages* carefully.

8.2.1 Externalising Assumptions

Reading someone else's code is demanding, even when written by an excellent programmer. At the moment of writing the programmer is fully submerged in the problem, accommodating for all the inputs she foresees the code can be confronted with. Reading her code is tough because you have to recover from it what the problems are it tried to solve. After you wrap up a part of the project and move on to work on something else, the wrapped up code quickly becomes as if written by someone else. Unit tests capture all the cases the programmer envisioned the unit of code should handle. It is a service to the future collaborator, who very well could be the same person, so she doesn't have to mentally fight her way (back) into the code. As soon as something is changed in the function such that it no longer does everything it was designed to do, the unit test informs us. A workflow without unit tests means that changes to code need to be interactively checked against, as was done when the code was written. The code fails, but it might be unclear why it fails, leaving the programmer with quite a puzzle. An even worse scenario would be if the code does not fail, for now. The exception that should be handled happens not to be in the data that is used to interactively check the modification. All seems fine and the code is adjusted. Then when the edge case does appear again in the data the product breaks mysteriously.

From the above it should be clear that to work in an Agile way automated code testing should be in place. Agile's core objective, continuous delivery, can only be obtained if modifications to the product can be made quickly. Omitting testing will result in *fingers crossed programming*, which is neither effective nor fun. At first it might seem that skipping unit tests will get you to delivery faster. However, as time progresses and the product grows in complexity, a unit tester will be able to keep the same steady pace in delivering new features, where the system of those who cut corners will start grinding to a halt because they have to keep digging in old code that no longer works because of the newly introduced features.

In the visual representation of building a data science product below each feature to be added is represented by a color. On top is the data scientist who does

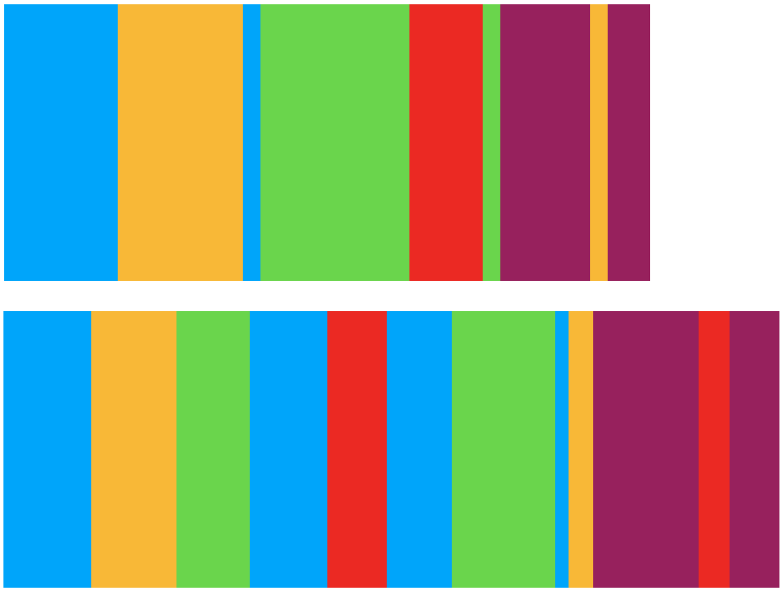


Figure 8.1: Time line for developing a product with (top) and without (below) unit tests

unit test, below the one who doesn't. The unit tester takes quite some time to implement the first two features, the non-unit tester delivers them quicker. After implementing the third feature, however, some code that was written for the first feature keeps failing and he doesn't know why. When the fourth is implemented, these problems arise again, in a slightly different form. When he finally manages to control it the code of the fourth feature gets unstable, asking for his attention. Meanwhile, the unit tester soldiers on. Sometimes she needs to make small adjustments to the old features, to make them click with new ones. In general however, she can uphold Agile's promise of continuous, equally paced, delivery.

Of course this example is contrived. After adding a new feature it gradually blends into the code base of the product. You will probably not say "I am going to revisit feature 2 now", but rather "I will look into this error that keeps showing up now and then when we run it on new data". Still, I think it is a good idea to keep this picture in mind, use unit tests and you won't be solving the same problems over and over again.

8.2.2 Writing Better Code

An additional benefit of unit testing is that it will improve the code quality directly. A point also made by Hadley Wickham in the first section of the chapter on unit testing. In order to be unit testable the code should comprised of many small parts, instead of big chunks. Like a broken machine that is easier to fix when components can be replaced separately, code comprised of many small components is easier to debug, optimise and maintain.

The main reason people are not eager to write unit tests, because at first sight it seems redundant and time consuming work. I hope by now you are convinced it is certainly not redundant. As you get the hang of it, you will notice that being slowed down does actually make you stop and think. What are the edge cases that this function can meet and how do you want it to behave when it does? By thinking through the scenarios before you put the function to work you can circumvent many problems upfront.

Chapter 9

Versioning

The approach in this text makes a hard cut between the data product and doing research to improve the product. The former should be treated as software, all best practices of software engineering apply to it. We have looked at automation and testing in the previous chapters, now we turn to versioning and version control. A version is a distinct snapshot of the working data product, to which a version number is assigned. Versioning makes it easier to trace the evolution of the product. In a machine learning model for instance, you can trace a performance measure as you go along. It is more than just assigning numbers, it is a clear indication that the product has been improved and that you close off a line of research.

9.1 Old Code

As you move from one version to the next, a part of the code written earlier might become obsolete. You might be tempted to keep this code around because it might be needed in future times. I would advise against that. As the project progresses the amount of unused legacy code grows, as well as the amount of used product code, because the product tends to become more sophisticated as you move on. It will become harder to figure out what part of the code is actually used, making it more difficult to reason about the product. Also, the package namespace starts to become clogged. It happened more than once to me that I had created two functions with the same name when there was some old version I was no longer aware of lingering. Happy debugging!

I think the product should only contain code what is used in the current version. This also implies that we should not do versioning within the code. Something I have done a lot in the passed and also seen with other data scientists. Whether versioned or not, you might be inclined to leave old versions of a product, such as a meta-function to call a model or to create a report, in place. Rather than versioning in your code, use the version control system. Keep your code base clean and as small as possible. If you really want to keep your code around, and don't want to rely on version control rollbacks to retrieve old results, keep an archive folder outside the `/R` folder so it is not part of the namespace.

9.2 Version Control

Data scientists more than software engineers tend to work in solitude. Fully using version control systems can then be considered as overkill. In practice, they might just be used as an external backup of the code. Write some code, push it to master, write some more code, push it to master. Whether working alone or together, it is a good idea to use version control for, well, version control. (If you are not yet comfortable with using `git`, Jenny Bryan has your back with this extensive resource.)

9.2.1 Master is Production

To move away from the *external backup* use of `git`, you should use branches instead of only using `master`. It is good practice to reserve `master` for finished versions of the product only. In 'normal' software engineering a new feature is developed on a fresh branch. Once the feature is finished, tested and approved in a code review, the feature branch is merged to `master`. When there is continuous integration in place the merge to `master` *is* deployment, the product is automatically updated when `master` is.

Remember that the workflow in this text makes a distinction between research tasks and software tasks. Research tasks are performed completely outside the `/R` folder, they do not alter the product. Whether the tested hypothesis in the task proved to improve the product or not, you probably want to merge the vignette that explores the hypothesis to master to keep an overview of the work you did. This is fine, as long as you indeed did not touch the `/R` folder. This is where the product is defined and where the versioning happens. As soon as the research task did indicate the product can be improved, start a software task to adjust the product. Merging the software task branch to `master` is to create a new version of the product, but merging a research task branch is not.

As an example, the git graph below is a simple example of the start of a new machine learning project. (The graph is read top to bottom.) After initiation there is the first setup of the project. A first query to the source database is created and we split into train and test. We then create a simple base rate model, this is `v1` of the product. Next, a research story to check if some algorithm can improve our current product is started. We do data prep and training and analyse the results. Alas, the algorithm did not improve the simple model too much, and we decide it is not worth the trouble. Still the branch of the research task is merged to `master` so we have an overview of what we did in the past in the vignettes folder. We try another algorithm. This one does a lot better and we decide to use it in the product instead of the simple model. First, the research task branch in which this was explored is merged to `master`, then we create a new branch for the software task. As soon as this work is finished and the branch is merged to `master` we release a new version.

9.2.2 Working Together

Because branching is used, several data scientists can work on different aspects of the model simultaneously. Creating separate branches to explore different hypotheses. It is advisable, however, to work on separate parts of the project in parallel. However, working on two or more software tasks at the same time is not advisable. This might give you headaches when you both try to merge to `master`, because the same files are modified. Also from a versioning perspective it is cleaner to make sequential changes. You can keep better track of the evolution of your product.

Software engineers rarely work alone. They check each others work by doing code reviews. Version control systems can even be configured such that a branch can only be merged to master when the changes are approved by at least one or two colleagues. Working together with multiple data scientists on the same project can greatly enhance the quality of the product. You can challenge each others ideas and you can look critically at each others work. Using version control properly will greatly improve your cooperation.

Chapter 10

Exploratory Tasks

We have explored the aspects of creating a data product in Agile data science workflow. The second part, the exploratory part, does not need so much attention, because by definition you can do what you like in her. One aspect deserves some consideration.

10.1 (Un)Reproducible Research

In the chapter on versioning it was advocated that you should not keep old code lingering as you move on. Rather use version control systems to keep track of the changes you are making to the product. Research tasks are explored in vignettes, these are kept around as the project goes along. It is likely that in research reports code from the software product and data from the pipeline is used. These are readily available, enabling quick exploration of hypotheses. However, as you move to next versions, the software product and the data will be updated. This means that the reproducibility of almost all research reports will be lost at some point.

It is up to you and the characteristics of the project if this is considered a problem. If you want to keep all reports to be reproducible at any time, there is a considerable amount of extra work to the research tasks. Data has to be stored separately and the source code used should be frozen somewhere outside the `\R` folder. I prefer to accept that the reports will get out of sync with the data and the source code. The conclusions are what matter most, and the code is always around to check if they were derived properly, even if it can no longer be run. If you really have to rerun the research report to verify results, you can always checkout the commit of the report and rerun the pipeline. The source code and state of the data at that "time point" are then available. I tend to number my research scripts to the version of the data product of that moment.

This will make it easy to understand the context in which the research was done at a later time point.

Part III

Miscellaneous

Chapter 11

TL;DR

A bullet point overview of this text:

Chapter 2

- Agile is a reaction to Waterfall, the standard for doing software development up until the nineties.
- Waterfall is often slow, projects can take years to finish.
- Agile wants to deliver software continuously, then improve it constantly, instead of one big delivery.
- The *Agile Manifesto* is a set of four values and twelve principles.

Chapter 3

- Workflows are serving the Agile philosophy and are not holy, change them if they don't serve you.
- Scrum works with *sprints*, in which the team commits to a set of tasks to complete in that time.
- Scrum tasks are formulated in *user stories*, that make clear how the completion of it adds value.
- In Kanban tasks are gathered on a backlog, the most important task is done next.
- Central metric is the *cycle time*, how long does it last on average to complete a task.
- Scrum is strict in its commitments and ceremonies, Kanban is flexible.

Chapter 4

- Ineffective workflows are one of the causes that many data science projects are currently failing.
- Early delivery of a minimal product or model will keep stakeholders excited, will make you fail fast and will get you feedback sooner.

- You should work closely with business people, if you want to work effectively.
- Every possible improvement to the product should be deployed right away.
- Improvements only count when they have left your system and others can benefit from it.
- Adopt the best practices from software engineering to your data project to assure high-quality code.
- Continuously reflect on the effectiveness of the workflow you are following.

Chapter 5

- Data science is different from regular software engineering, because it needs to exploit yet to be discovered relationships in the data.
- This makes it unfeasible to give a tight planning on what will change to the product, as is done in Scrum.
- A two-way workflow is proposed in which a hard split is made between the *product* and *exploratory research*.
- Tasks are added to a Kanban board and are limited on doing exploration or changing the product.
- For exploration it might be a good idea to scope tasks as the amount of time a data scientist can spend on it.
- If there is a need for exploring the subject more after it, create a new task for it.
- Working Agile is what matters, not adhering to a specific process. If something does not work for your product, change it immediately.

Chapter 6

- The R package structure is very suitable to organise a two-way workflow with exploration and product.
- Create low level and meta functions in the package, apply in a pipeline.
- Use Rmarkdown in the Vignettes folder to do data explorations.

Chapter 7

- The data product should be automated as much as possible.
- Automation saves time and energy in the long run, because the data scientist does not have to rerun the same thing over and over.
- Automation serves as documentation, because it makes clear how the product pieces play together.
- Automation assures reproducibility, if it is not reproducible it will not run.
- R has the fantastic `drake` package for automation.

Chapter 8

- Code should fail fast and informatively if you want to be able to continuously deliver.
- Because R is weakly typed it is a good idea to implement type checking at the start of functions.

- If your function depends on the presence of certain columns in a data frame, check for it at the start of the function.
- It is also a good idea to check all the implicit assumptions you have to the possible values of your data.
- The short term investments for unit testing pay off in the long run.
- Unit tests capture all the assumptions the programmer had at the moment of writing the code. They are an externalised memory of all the cases your code should handle.
- Because unit testing will slow you down and have you think through all possible situations, it will make you write better code right away.

Chapter 9

- Use versioning for the data product, increment the version number each time the product is meaningfully changed.
- Delete code that is no longer part of the product, use version control for versioning instead of your code base.
- Use branches to change the product. Get a "merge into master is pushing to production" mindset.
- Research branches should not contain changes to the product, so they can be merged to master without creating a new version.
- Using branches is essential if you work with multiple data scientists on a project at the same time. Chapter 10
- It will take a lot of effort to keep old research results in the vignettes reproducible as the project progresses. It is up to the data scientist and the project specifics if this is considered as a problem or not.
- You could number your research scripts to the version of the data product at that moment. You can rollback the code if you have to rerun the analysis.

Chapter 12

Case Study: Building the *Valuecheck*

My current employer funda is the market place for selling and buying homes in the Netherlands. A long standing company wish was to use the data of recent house sales for predicting the current value of all houses in the Netherlands. We knew that home owners used the asking prices of neighboring houses published on *funda* to keep track of local market trends. We wanted to facilitate them to translate the recent sale prices in an official estimate of their specific house. They no longer had to look at the houses that are offered, our statistical model had already done that. Moreover, they did no longer had to make the translation of offered houses to their own house informally, the model determined which characteristics of a house mattered and which did not. A final advantage was that we could use of the selling prices instead of the asking prices, these are not shown on the website. So the product reflected what their house could be sold for, instead of what the typical asking price of their home was.

To create what eventually would become the *Valuecheck*, a colleague data scientist and I joined an existing Scrum team. This team comprised of two back end developers, a front end developer, a UX designer, and a product owner.

12.1 Trying Scrum

The team had a lot of experience with Scrum and had all the workflows in place, so it made sense to try to fit our tasks into this framework. At first this worked out quite well, because the first set of tasks we had to complete were essentially software tasks. Setting up a server, build a first query so we had a modelling set, splitting into train and test, and doing some data cleaning. These tasks were well scopeable, we could estimate the time we needed to complete them quite accurately and they had a clear definition of done. Then the model building started and we got more and more trouble fitting the tasks into the tight Scrum methodology. We could not tell what the model would like in two weeks time, it depended on the relationships we would find in the data. We certainly could not give estimates for what the model quality would be then.

12.2 Informing the Business

Our product owner informed management about the progress of both the product and the model. In consultation with them he decided how we would roll out the product. We had to provide him with the information required to make such a decision. A *Shiny* dashboard appeared to be the way to go. In this dashboard we could show the basic model performance, reflected in an agreed-upon statistic. Moreover, the regional performance was shown on a map, making it clear where the model was performing well and where it was doing poorly. After a model update, the data frame with cross-validated scores underlying the dashboard was replaced to show the new situation.

12.3 Moving to Kanban

Having scopeable tasks is essential for building proper Scrum sprints. As a team you have to commit to what you are going to complete in the upcoming two weeks. No longer being able to do that, we could not really be part of the Scrum rhythm anymore. We found the alternative in moving the data science tasks to a separate Kanban board, stepping out of the Scrum cycles. The circular nature of data science, as discussed in Chapter 5, does not lend itself well for tight planning. We started with a Kanban board with six lanes *to do* - *test hypothesis* - *code review hypothesis* - *update model* - *code review update model* - *done*. This reflected a full cycle of researching a hypothesis and updating the software. It worked quite well, however, a portion of the tasks did not reach the finish line, because the hypothesis tested appeared not to improve the model. This problem does not exist if the research part and the software part are split in separate tasks, the software task is only created if the research indicated the model could be improved. We never did this during this project, but this insight improved our workflow in subsequent projects.

12.4 Building an MVM for the MVP

Building a predictive model that is part of a dedicated product is both challenging and rewarding. Too often data science projects are initiated as a proof of concept, without a clear vision on how to implement if the prediction can be successfully done. Knowing from the start that the model is going to be used is very motivating. On the other hand, this means that you need constant alignment with the team that develops the product around the predictions. The houses offered for sale on *funda* have many characteristics filed, giving us a rich feature set to work with. However, as an MVP we wanted to present the users with an estimation, without them having to fill in all kinds of characteristics of their house. Developing a product from static house predictions is far less complex and time consuming than from a dynamic model with adjustable inputs, both from modelling and a software perspective. This implied that we could only use features that are freely available for every house in the Netherlands. Luckily, this was true for the two most important features, location and time. Also the surface area of the houses were available in a public database. From this we started to build our initial prediction models. First using a simple regression model to create a baseline. We have a preference for statistical models over machine learning algorithms, because they not only give us predictions, but also insight. However, we needed some decent predictions fast, and it was clear we needed to exploit some nonlinear relationships. We therefore used ensemble methods that gave superior predictions over regression models.

Already it was decided, we would only release the MVP in geographical areas in which the MVM performed well enough. This is called a *soft launch*, release the product without giving it too much noise for a selected group. Even then we did not quite make the minimal performance goals we set ourselves. However, we could include a categorical feature and simply export the predictions for every level of the feature for every house, as long as there were not too many levels. The type of the house (apartment, one of several Dutch types of houses) appeared another crucial predictor. Finally, we wanted to show lower and upper bounds to a prediction, not only giving a point estimate. After some research we were able to this with random forests, that were trained on the desired quantiles. Predictions were exported in csv files, a front end and back end were built around these. Doing a prediction on the website was just a simple look-up.

12.5 Improving the Product

From the start users could provide us with feedback, using a simple *thumbs up, thumbs down* and if they wished subsequent comments. Of course you want your work to be liked, but as I quickly learned, in this stage the best feedback is negative feedback. You know the product is barely good enough at this point in time, both from a software and a data science perspective. Negative feedback indicates people care about the product and it bothers them it does not fully meet their expectations. Moreover the feedback can point to directions that gives the biggest satisfaction jump when improved. If it appeared that the users did not care about the product in the first place, the project could be killed and little resource was wasted on it. Fortunately, users did care, so we went ahead and started improving.

12.5.1 Interactivity

Using an interactive model instead of the static MVM would improve the product in several ways. It enabled us to use features in the model that were not publicly available, the user could enter them. Also, we knew that the data on house surface area was not of consistent quality. Comparing them to the “real” data in our database for houses that were placed on our website indicated that they could be off in both directions. In fact, for the MVM we used a correction to predict the “real” surface area based on the public data. When the product changed to interactive, the user could correct the prefilled information if necessary. Finally, it improved transparency and user experience. Having an interactive product meant we had to bring the model to the product, not just the predictions.

12.5.2 Changing the Model

From the start we saw the problem we were trying to solve would have a natural fit with Bayesian regression models. Price trends are hugely important in the housing market. With the *prior - posterior* structure of the Bayesian framework, we could easily update each month. Making time an implicit variable, rather than a feature we use for modelling. Secondly, we always wanted to give a lower and upper bound to the predictions, these are part of the posterior of Bayesian predictions. The only problem was, we did not have hands-on experience with Bayesian modelling, we just knew the theory behind it. After extensive discussion if we did not see another way of significantly improving the model, we decided to go for it. This was a gamble, because we needed a few weeks to refresh our knowledge and to get our feet wet with *Stan*. The modelling was basically placed on hold then. Fortunately, it paid off. After some experimenting we found a good model structure for our problem, improving the model especially at locations we did not do so well at before. Moreover, we could now start to add more features to the model. This improved both the accuracy of the predictions, the confidence bounds around our estimates shrunk, and improved user experience significantly.

12.6 Productionising the Interactive Model

Up to this moment in my career, productionising data science products meant building a shiny dashboard to interact with results or exporting plain text files. My background is in statistics, not software engineering, I could not tell what was required to expose a model to the millions of visitors to our website. Luckily our data engineer could help. I (my data science colleague was working on a new project) exported the posteriors of the parameters in flat files. He built a python API that took the feature values as inputs and returned the lower and upper bounds and the point estimate for the requested house. Instead of me telling him how the model scoring should be done, I built the same functionality in R. He then copied that functionality to python and added his caching, load balancing and garbage collecting magic.

Up until this point I am not sure if it is possible to create an R API that is up to the task. Sometimes it is argued by python evangelicals that you should only use python because you can do everything in one language. Doing it in R first and then in python causes double work, which is a waste. I beg to differ. First of all, the majority of the work did not have to be re-implemented, data prepping, model training and model updating are done on train data only. It is only the scoring module we implemented both in R and python, which is just a fraction of the entire R code base. Even this is not a waste, rather it served as double bookkeeping. A number of small bugs were caught because the python API did not return the exact same results as the R module.

12.7 Thank You!

Building the *Valuecheck* was an amazing experience in which I learned so much. About working with Agile, about productionising software and about cooperating closely with a software engineering team. It has substantially changed my views on what it means to do data science. A major thanks for goes to Daniar, Oriol, Stephan, Rick, Sander, Riccardo, and Marco. You are awesome!