AI-assisted Software Development Effort Estimation

Master of Science in Technology Thesis
University of Turku
Department of Computing
Software Engineering
2021
Karri Koskinen

UNIVERSITY OF TURKU
Department of Computing

Karri Koskinen: AI-assisted Software Development Effort Estimation

Master of Science in Technology Thesis, 126 p., 10 app. p.
Software Engineering
June 2021

Effort estimation is a critical aspect of software project management. Without accurate estimates of the developer effort a particular project will require, the project's timeline and resourcing cannot be efficiently planned, which greatly increases the likelihood of the project failing to meet at least some of its goals.

The goal of this thesis is to apply machine learning methods to analyze the work hour data logged by individual employees in order to provide project management with useful estimations of how much more effort it will take to finish a given project, and how long that will take. The work is conducted for ATR Soft Oy, using the data from their internal work hour logging tool.

At first a literature review is conducted to determine what kind of estimation methods and tools are currently used in the software industry, and what kind of objectives and requirements organizations commonly set for their estimation processes. The basics of machine learning are explained, and a brief look is taken at how machine learning is currently used to support software engineering and project management. The literature review revealed that while machine learning methods have been applied to software project estimation for decades at this point, such data-driven methods generally suffer from a lack of relevant historical project data, and thus aren't commonly used in the industry.

Initial insights were gathered from the work hour data and analysis goals were refined accordingly. The data was pre-processed to a form suitable for training machine learning models. Two different modeling scenarios were tested: Creating a single general model from all available data, and creating multiple project-specific models of a more limited scope.

The modeling performance data indicates that machine learning models based on work hour data are capable of achieving better results in some situations than traditional expert estimation. The models developed here are not reliable enough to be used as the sole estimation method, but can provide useful additional information to support decision making.

Keywords: software development, effort estimation, machine learning, project management

Contents

1 Introduction

For most software projects, developer effort is the most important resource, whose allocation has a significant role in determining how the project's cost and timeline will ultimately play out. Consequently, the ability to estimate the required developer effort in advance, and adjust those estimations during the whole project life cycle is paramount to achieving success in a single project instance. Inaccurate (typically overoptimistic) effort estimations are a common cause for project failure [1], with obvious implications for the wider organization's success.

As a result, effort estimation has been the focus of a considerable amount of research in the past decades, with the goal being to evaluate and improve upon existing estimation methods, and devise some entirely new ones [1] [2] [3]. Historically, the most commonly employed estimation method is some variant of expert estimation, ie. relying on human subject matter experts to study the relevant data and come up with an estimation. Data-driven methods have also been developed, but these are less commonly used in practice for a variety of reasons [3]. Among these data-driven methods are various machine learning techniques. Applying machine learning methods to this problem area is not a new development, research in this direction has been conducted fairly consistently in the past decades even before the current surge of AI-related interest and research [1].

The goal of this thesis is to apply machine learning techniques to individual employees' hours tracking data in an attempt to predict their workloads, and thus provide information that project and company management can use to support their decision making when determining how to allocate developer time to different projects. Additionally, we will perform some exploratory analysis of the data, at a minimum to determine how the hours logging process might be improved to support potential future data analysis projects.

It is worth noting that while some research has been done regarding prediction of individual workloads or developer availability [4], the majority of research appears to consider wider project management issues. This division is somewhat arbitrary however, and obviously the two cases have overlap. In this thesis the initial focus is on predicting individual workloads, but by necessity this work needs to be grounded in the background of more general project management and effort estimation theory. Thus these topics will be covered in the background chapters in some detail. Furthermore, it is possible that the focus of analysis will need to be shifted during the process as new insights are discovered from the data.

The research questions we seek to answer are the following:

- **How is machine learning currently utilized to support project management / software engineering work?** Before we attempt to analyze the actual data set on hand, we must know what has been done in the field thus far; are there ready-made tools for such a case, or do we need to adapt general machine learning techniques for the task.
- **Is it possible to predict an employee's workload from their reported hours data?**
- **How might the hours logging process be improved to better support data analysis projects in the future?**

The data set we have available consists of the logged work hours of all employees of ATR Soft Oy since spring 2009. The data contains the number of worked hours, the related project and task, as well as short free-form descriptions of assignments done during those hours. The objective of the analysis is to determine how full the employee's workdays currently are, and to hopefully predict when their current projects might wind down sufficiently for them to take on a new project. This and any additional unforeseen insights that may be found during the analysis process should be combined to form suggestions to management regarding how to staff upcoming projects.

The rest of the thesis consists of 5 chapters.

Chapter 2 provides background information regarding project effort estimation, the classical methods used for it, the reasons for its importance, and various problems that tend to occur due to inaccurate estimations.

Chapter 3 provides detailed background information about machine learning (ML) techniques and their application in general. A brief look is also taken at how they are currently applied to support software engineering and project management.

Chapter 4 explains the background and goals of the practical analysis work. The ATR Soft data set is introduced and the assorted initial understanding and pre-processing steps of the data analysis are performed.

Chapter 5 describes the practical application of various machine learning methods on the previously prepared data set. Model performance is evaluated and results are discussed. Potential directions for future work are also presented.

Chapter 6 provides the conclusions and final discussion of the results of the practical sections. Research questions are answered, and the results are discussed in a broader context in comparison to the starting assumptions and background information.

2 Resource/project management in software engineering

In general terms, project management consists of the various tasks needed to steer a project to successful completion. In practice this means communicating with various stakeholders, planning and allocating the use of any and all resources available to the project, monitoring progress and, as circumstances evolve, adjusting the aforementioned plans and allocations accordingly. This is generally a well-studied and commonly applied discipline, universally used to steer projects in countless domains, from bridge building to software engineering. The particulars will vary somewhat by domain, but the idea of balancing the goals of scope, cost and time are universal [5].

For software engineering in particular, project management has been a very active discussion and research topic for decades, due to the field's constant difficulties in consistently achieving success in its projects. The exact numbers vary between sources and definitions of success and failure, but generally less than a third of software projects are considered fully successful, with the remainder being partial or total failures [6] [7].

This so-called software crisis was first identified in the late 1960s, as increasingly powerful computers enabled the creation of increasingly complex software, and this increased complexity then overstressed the software engineering tools and processes of the time. Since then, considerable amount of improvement has happened in both categories, but the demands placed on software have also kept growing. As a result, the fundamental issue of software projects slipping outside their time, quality and cost constraints remains [2] [6] [8].

2.1 The role of estimation

One of the key challenges in software project management is accurately estimating the total effort needed to create a given piece of software. Such estimations have a significant effect on the budget and expected time frame of a project (though other factors also play into this), so their accuracy or lack thereof will have a considerable effect on the project's chances of success right from the start. Poor estimations result in over-optimistic project plans, which then contribute to projects missing deadlines and going over budget. Because of this phenomenon, improving software project estimation tools and processes has been an active field of research for decades [1] [8].

The concepts of estimation and measurement go hand-in-hand. Anything that can be measured can be estimated (with varying methods and degrees of accuracy), and anything you wish to estimate has to be measurable in some form. Measurements can generally be divided into two categories: Direct and indirect. Direct measures are things that can be measured objectively, like number of lines of source code, project cost or running time of a particular task. Indirect measures are more subjective in nature, things that can't be directly calculated but have to be based on the judgement of a person or group, like quality or usability [2].

In addition to effort, defined here as person-hours needed to achieve the project goal, measures like time and cost are commonly estimated during project planning and tracked during the project's life cycle. The exact collection of measures and estimation methods used will vary depending on the project's needs and the preferences of the people running it, but they tend to be strongly connected to each other; a change in one will have an impact on others. Furthermore, the actual low-level tasks that a software development process consists of are highly interconnected, and in many cases contain inherent uncertainties that make them difficult to measure accurately [2] [3].

The result of this interconnectedness and inherent uncertainty is that accurately estimating software project parameters during planning is extremely challenging. Indeed, even shorter-term prediction later in the project life cycle when there already is some amount of project-specific data to base such predictions on can still be a challenge. This in turn greatly increases the risk of the project failing in part or even in its entirety. An additional complication is that many estimation methods specifically tend to err towards the over-optimistic. Projects don't exist in a vacuum, cost overruns and missed delivery deadlines have very real implications for the surrounding business processes. While inaccurate effort/time/cost estimations aren't the sole cause of project failure, it is a significant enough of a problem to merit study.

2.2 Objectives, requirements and restrictions

At a high level, the purpose of software project estimation is to produce information to support decision making in and around the project. The exact objectives will however vary somewhat between organizations. Trendowicz et al. (2011) [3] found the following:

- **Project planning and tracking:** Self-explanatory. Estimating the time and resources needed in advance, and keeping track of them during execution is a staple of project management work across all fields.
- **Process improvement:** Improvements to the software development process

in terms of quality, cost, etc. is a constant need in the industry. Information obtained from effort estimation can support this goal.

- **Project management overhead:** Somewhat overlapping with the previous objective, keeping unproductive overhead to a minimum is a common goal. This applies to project management in general, and the effort estimation methods employed themselves. Organizations want effort estimation to be as lightweight a process as possible.

- **Negotiating project costs:** Effort estimation can be used to justify development costs and projected timelines during software procurement process.

- **Risk management:** Effort estimation can provide information for identifying project risks in general. Conversely, estimation should be clear on its own assumptions and expected accuracy, to help manage the risk caused by the estimation itself.

- **Productivity improvement:** Estimation can help identify parts of the development process with most potential for increased development productivity.

- **Project benchmarking:** Effort estimation can help compare different projects, and potentially the organizations that ran them, thus assisting with decision making in relation to scenarios like outsourcing and selection of suppliers.

- **Change management:** Estimation can support the management of changes during project execution, when requirements are modified due to some unforeseen circumstance.

Estimation methods need to be selected to meet the organization's estimation objectives, but the objectives themselves aren't the only parameter that affects method selection. Methods should also be evaluated based on some qualitative requirements of the method itself. Some of the most critical such requirements according to Trendowicz et al, (2011) [3] are:

- **Expert involvement:** How much expert effort is needed to apply the method? Generally organizations would rather have their experts doing something other than estimation, so methods that require less expert effort are favored.
- **Required data:** Organizations tend to have fairly sparse data available about past similar projects. Thus if a method requires a large amount of input data, it will likely be unfeasible to apply.
- **Robustness:** How well the method handles poor quality data. While you can't expect good results from poor quality data ("Garbage in, garbage out"), some methods handle various data quality problems (incompleteness, inconsistency, etc.) better than others. As such, focus should be on selecting a method that deals well with the quality issues in the data set at hand, which of course implies that those issues are actually known.

Several other requirements such as flexibility, complexity, support level, etc. [3] might also be evaluated, as well as possible budget constraints and personal preferences of the team itself. The end result of all of this is that the selection and application of estimation methods tends to be a very team/organization-specific process, and as such the resulting method selections generally aren't portable between teams as-is. This is in addition to any team- and project-specificity inherent in any generated models themselves.

It should be noted here that while problems with effort estimation are a significant contributor to software project failure, they aren't the only cause. Because of this, improvements in effort estimation (whether better method selection or improvements in the methods themselves) will not be a comprehensive solution to project failure, merely a piece of the puzzle. This in turn means that for an individual organization, the role of estimation itself, the related tools and any available improvements to them will likely be a part of a wider process improvement endeavour, where the local circumstances formed by that endeavour will affect the estimation part in some way.

For example, one of the most commonly applied software engineering process improvements in the past two decades has been the adoption of agile development methodologies. In contrast to the old so-called waterfall method of software development, agile methods seek to make it easier to adjust things like requirements and specifications during the course of the project, as any deficiencies in the initial plans make themselves known during the development process. In practice this means (among other things) that there's less emphasis on planning and analysis, because the goal isn't to have the plans set in stone before starting the implementation. A process like this won't make effort estimation unnecessary, but it will change its focus and set some practical constraints.

Currently, the estimation methods most commonly used in agile software development are various expert-based subjective assessment techniques [9]. This is likely because such methods are relatively lightweight in terms of required data and manual work, and thus fit well into a process where time is rather spent on something like practical prototyping. However, research suggests that estimation accuracy is fairly poor in this context too [9], which can cause problems for the project even if an agile development process is less sensitive to such issues at the level of individual development tasks.

2.3 Classical estimation methods

For the purposes of this thesis, the definition of a classical method of effort estimation is anything that isn't based on machine learning. This is a somewhat arbitrary definition, since the field of machine learning does contain methods that aren't necessarily all that distinct from some of the more data-driven effort estimation techniques that we'll cover here as classical methods. Furthermore, older machine learning methods like linear regression and decision trees are often referred to as classical methods within the field of machine learning, which might cause confusion in the context of this thesis if not addressed specifically here.

The estimation methods covered here can be broadly divided into three categories, based on whether they focus more on expert opinion, project data, or include both (hybrid). Most estimation methods of course do contain features of both camps, but calling them all hybrid methods based on that alone would be an oversimplification, since most methods do have a clear focus one way or another.

2.3.1 Expert-based methods

These methods mostly rely on expert judgement to estimate projects and their components. The exact methodology may be highly structured or more ad-hoc in nature, and while individual experts are likely to factor any data they are aware of into their estimations, it's not the focus of the method itself.

Expert-based methods are currently the most commonly used method class in the industry, which is somewhat incongruous given how loath software organizations appear to be to allocate expert effort into estimation tasks when said experts have other project tasks to perform. The discrepancy is likely explained by the fact that expert-based methods generally require little if any past project data, and many of these methods can be used to estimate projects even when said projects are very poorly or sparsely defined. Humans can always produce some kind of a guess after all, even with very limited prior experience or knowledge of the problem at hand (accuracy will of course be impacted). Nevertheless, the unwillingness to allocate much expert effort to estimation may still result in a situation where the estimation isn't performed by the actual experts best suited to the task, but someone else in the project organization who was available [3] [10].

PERT

This method was developed in 1958 by the US Navy as a tool for estimating the schedule of any large-scale research and development project. The project that drove the development of this method, and the first one it was applied to was the Fleet Ballistic Missile (FBM) program, which eventually produced the Polaris missile. Since then the method has been applied to various other types of projects, software among them.

In simple terms, the method seeks to identify the key events (tasks) of a given project, and the order those events take place. This sequence of events produces the desired end result. The sequence and the interdependencies of the events are depicted as a directed graph (called flow plan), where the nodes are events and the arrows between them depict the estimated time needed to accomplish an event once its pre-requisite events are complete. The time estimate is produced by combining expert estimations of best, most likely and worst case scenarios into a single time distribution. Thus, while statistical techniques are employed, at its heart the method still relies on expert opinion [11].

PERT is commonly combined with the Critical Path Method (CPM, developed at around the same time as PERT), and is mostly concerned with estimating the project schedule, rather than effort or cost. The method is fairly cumbersome to apply, and as such it is most commonly used in large-scale projects.

Wideband Delphi

Wideband Delphi [12] is an evolution of the Delphi method originally developed by the Rand Corporation in the 1940s. The idea is that each member of a group of experts (estimators) provides their estimation of the given problem, and these are then summarized by a coordinator, who presents the initial estimates to the group members, who can then revise their original estimates based on this new information. This process of feedback and revision continues until either the estimations have converged sufficiently (the criteria for this can be defined on a case-by-case basis), the estimators are unwilling to revise their estimates any further, or simply a pre-determined number of rounds have been completed. The estimations are kept anonymous, ie. the estimators don't know who made which individual estimate.

In the original Delphi method, the feedback was delivered in writing, but this was later determined to be a bottleneck. The written format didn't allow sufficient transfer of information between participants for them to meaningfully adjust their estimates based on the feedback. This resulted in the development of the wideband version in early 1980s. The basic mechanism remains the same, but in Wideband Delphi each estimator prepares in advance, and then the actual feedback process is done as a meeting. The estimations are still written down and presented anonymously by the coordinator, but feedback is handled as a face-to-face group discussion.

The main advantage of Wideband Delphi is that it requires no historical project data whatsoever. Additionally, structured group consensus as an estimation paradigm has been studied and validated fairly extensively in both academic and industrial contexts. The greatest weakness of this method is that it requires quite extensive expert involvement, which makes it both expensive and cumbersome to apply in practice [10].

Planning poker

Planning poker was introduced in the early 2000s as an estimation technique suitable for use in an agile software development process [13]. As agile development places less focus on extensive planning, estimation techniques used in such a context need to be correspondingly lightweight.

The estimation process is something of an evolution of the Wideband Delphi method, but there are important differences in the details. In planning poker, what the experts actually estimate is the size of a given user story. Once consensus is achieved on that, the estimated size of the story is compared to previously implemented stories, which gives the team an idea of how much effort the development will take.

In practice, each estimator has a deck of cards with different story point values printed on them. After an initial discussion about a user story, each estimator assigns a point value to the story size, by playing one of their cards face down. This avoids the anchoring effect as other members can't initially see each other's estimation. Then the cards are flipped, and the estimations are discussed and justified (particularly outliers), after which estimations are revised and once again presented with the cards face down.

Planning poker is less cumbersome to use than Wideband Delphi, while providing a fairly similar feedback cycle process. However, it does still require a significant amount of expert involvement, even though there is less overhead. The method is typically used in agile development processes for iteration and release planning [10].

2.3.2 Data-driven methods

These methods basically seek to fit some model using data from previous projects, and use that to estimate the current project. In a broader sense, this is very close if not outright identical in concept to classical machine learning methods, and indeed some of these methods feature some kind of regression/curve fitting components that are a staple of classical ML. In addition to differences in the models themselves, these methods differ from each other in the scope and type of the data that is used. Some methods seek to be universal in the field, such that the model itself is formed from some given set of project data from across the industry (resulting in a fixed model), whereas others rely more on data from the project organization at hand.

A core limitation of these methods is naturally the data they are based on. In case of a fixed model, it is important to ensure that the data set underpinning the model is actually sufficiently reflective of the project to be estimated. This is not a trivial thing to check, especially given that the data set might be proprietary (the actual algorithm is rarely a secret, but the data set can be hidden as part of a software tool or such) and thus access to it can be very limited. On the other hand, a fixed model does largely bypass the problem of needing historical data from the project organization. This can be a useful feature, because in many organizations there is little if any available data on relevant past projects, which makes it outright impossible to use any method that requires such [10].

In general, data-driven methods are less commonly used than expert-based methods. There are also considerable differences between the popularity of different methods belonging to this category. Regression-based methods are fairly common, whereas fixed models like COCOMO (see below) are much rarer [3].

COCOMO II

The Constructive Cost Model (COCOMO) is a fixed-model estimation method that was originally published in 1981 [12], and has since been updated and adapted as the software engineering field has evolved. Several versions of the method have been developed, with various levels of detail, and with some even being more hybrid than data-driven in nature (Expert-COCOMO, COCOMO-U). COCOMO II was introduced in 2000 as a response to the field having moved away from processes and techniques that were common when the original model was developed. Among the new and updated features are support for different size metrics, support for re-use and re-engineering, and various updates to assorted effort drivers included in the model.

At its heart, COCOMO is based on a statistical regression model, described by the following equation [10]:

$$Effort = A * Size^{E} * + \prod_{i=1}^{n} EM_i \tag{2.1}$$

where

- **Effort** is the total project effort in person-months
- **A** is the productivity coefficient (initially 2.94)
- **EM** are the effort drivers
- **E** is the effect of scale, which is calculated from a constant and a number of scaling factors

Scale factors and effort drivers are essentially numeric coefficients that represent the benefit or detriment caused by some characteristic of the project, organization or circumstance [10].

While the model can be used as-is with a default configuration, in most cases it is necessary to tailor the model to the organization using it, by adapting the assorted parameters like effort drivers and scaling factors to more accurately represent the organization's situation. Furthermore, COCOMO II accounts for the varying amounts of information available at different phases of a project by including two sub-models. The Early-Design model is intended to provide a rough estimate early on in the development process, when only limited data is available about the project. The Post-Architecture model is intended to provide more detailed predictions later on, when the architecture already exists.

As a fixed model, COCOMO II doesn't technically require any historical data from the organization that uses it. However, in order to tailor it for improved accuracy for a given organization, some data is required, which can still make the model difficult to adopt when data is scarce. Similarly, while expert effort from software engineers isn't required, a considerable amount of method-specific expertise is still necessary to adapt the model to the organization's circumstances [10]. As such, while COCOMO is one of the more well known estimation methods, it is much less commonly applied than expert-based methods or define-your-own-model statistical regression [3].

There are several conceptually similar parametric estimation methods, like SLIM and SEER-SEM [10].

Case-based reasoning

Case-based reasoning is a memory-based estimation method, that seeks to predict new projects based on how similar past projects went. The estimation is entirely based on quantitative project data.

This is essentially identical to the classical machine learning method called k-nearest neighbors (kNN). The only things needed for case-based reasoning are quantitative data from past projects, and some definition of a similarity metric between said projects. Some collection of important project characteristics (effort drivers) are selected, and then the values of those from the past project data (so-called analogs) are compared to the corresponding values in a new project (called target). Prediction for the target is derived from the known effort values of a number of closest-matching analogs.

As with kNN, there are various ways to measure distance between projects. The most common way is to calculate the Euclidean distance from the respective effort driver values of the analogs and the target. The number of closest analogs to be considered will likely have a significant effect on the produced predictions, so some consideration will need to be given to the matter of selecting that number. Effort drivers will also likely vary in importance, so scaling them to favor the ones deemed most important may be necessary. The prediction itself may also be conducted in different ways, for example you could take the mean or median value of the closest analogs.

Case-based reasoning doesn't require a great deal of expert involvement, and doesn't require any highly-specific pieces of data. Anything available can be used, though poor data will of course still lead to poor predictions. As it is essentially an application of kNN, it has the same weaknesses: sensitivity to outliers, inability to deal with very high-dimensional data sets, and inability to learn to weight the input data in any particular fashion (ie. if weighting of features is needed, it will need to be done as pre-processing) [10] [14].

2.3.3 Hybrid methods

Hybrid methods seek to to improve estimation accuracy by combining the strengths of expert-based and data-driven methods. The exact means of achieving this vary by method, but the general idea is to use expert evaluation where necessary, and then complement that with relatively lightweight data-driven methods to mitigate the biases and inaccuracies potentially introduced in the experts' opinions. This way the expert effort will reduce the amount of data needed, making the general method more accessible, but at least some of the benefits of a quantitative approach are still retained. The results of a hybrid estimation process also tend to be more human-comprehensible than those of a purely data-driven method, due to human experts being closely involved in the process.

On the other hand, hybrid methods do tend to be fairly complex as a whole. The techniques involved extend outside the normal software engineering skill set, which might necessitate additional training for the personnel involved. Furthermore, most currently existing hybrid methods aren't very good at dealing with redundant, inconsistent or incomplete information. The combination of data and expert judgement can identify such situations fairly well, but means of coping with it beyond that are limited [10].

CoBRA

Cost estimation, benchmarking and risk assessment (CoBRA) is an effort estimation method developed in 1998 for the express purpose of estimating software projects [15]. It is parametric and model-based like other, more data-driven methods, but has considerably lower requirements of measurement data and is also able to include expert judgement.

At its heart, CoBRA divides software development effort into two categories: Nominal effort and effort overhead. Nominal effort means the effort required to complete a project in ideal circumstances, and overhead means the effort consumed by the necessity of overcoming various difficulties encountered in real-world software projects. This principle is implemented via two models; effort overhead model and productivity model.

The effort overhead model is obtained from experts like experienced project managers. At first effort drivers (circumstances that affect the development positively or negatively) are collected as qualitative data, which is then converted to quantitative effort multipliers via expert judgement. Effort multipliers are essentially percentage increases to the nominal effort.

The productivity model is derived from the organization's historical data. This is achieved by fitting a simple regression line to past project data where the amount of overhead is known, and thus the nominal effort can be estimated to a reasonable degree. It only takes approximately ten past projects to fit a regression line, and CoBRA doesn't require any measurement data other than that. Thus the typical issue of sparse measurement data is less of a problem for applying CoBRA.

The final effort estimate is produced from the new project's size, effort overhead and the (nominal) productivity model. As the exact project characteristics aren't typically known with certainty early on, the characteristics are modeled as distributions. Overhead and final effort distributions are then derived from those via a Monte Carlo simulation.

As the theoretical basis for CoBRA is relatively simple and expert opinion is involved in the process, the method tends to produce fairly human-comprehensible predictions. The data requirements are fairly light, but the requirement of expert involvement may still be an issue, even though it isn't the sole focus of the method [10].

2.4 Estimating individual workloads

Classical effort estimation methods are generally targeted at estimating the effort required by a whole project, with individual development tasks being about the smallest estimation target seen in literature. The estimation (and prediction) of an individual software developer's workload appears to be an area that classical effort estimation has generally not been applied to, beyond dividing the estimated aggregate effort between the available development team members in some fashion. As the topic appears absent in the literature, we cannot say for certain why this might be, but we can speculate on the reasons, and infer some requirements for a potential individual effort estimation tool from that.

Effort requirement: An obvious explanation is that even the most lightweight estimation systems are too cumbersome to apply to a significant amount of individual developers. A common trend seen in estimation method selection and application is that it should take as little time and expert effort to apply a method as possible, and more complicated methods will only ever be applied if the project itself is very large in scale. This already rules out the majority of classical methods; it simply isn't considered worthwhile for 5 people to spend time trying to predict what one person might be doing a week or month from now, especially if the same process would then have to be repeated for a significant number of team members.

Accuracy/usefulness: Another potential issue is the accuracy and usefulness of any predictions obtained. Effort estimation is a notoriously difficult process at all levels. When dealing with the lowest commonly encountered level of estimation, ie. that of individual development tasks, the estimations are often off by a significant margin even when the project is already somewhat mature and the developers have some experience-based knowledge of how the system works. Given such inherent volatility, it might simply be that no known method produces estimations of sufficient accuracy to be useful at this level.

Psychology, privacy: People generally don't like to be micromanaged or have their activities scrutinized in great detail. Focusing a significant amount of such effort is likely going to upset at least some people, which will then have various problematic effects on the general work environment. Furthermore, at some point privacy concerns will also enter the picture, especially now that GDPR is in force and and some data collection and processing techniques will likely run afoul of the new lawfulness, fairness and transparency regulations [16].

With these considerations in mind, it is not surprising that organizations would generally prefer to handle individual workload estimation as more of an ad-hoc management task than any strictly defined estimation process. Estimating the workload of individual people is simply left up to them and their closest superiors, to be dealt with as the working relationship between them dictates. Assorted reporting tools (hours tracking, various business intelligence systems etc.) can be used as deemed necessary, but ultimately people are expected to make the best of their circumstances among project deadlines and other organizational goals.

Given such general situation, any individual effort/workload estimation system needs to be very lightweight; its application cannot require a significant amount of time and expert knowledge. This essentially dictates an automatic tool of some kind, operating on data that is already being generated as part of the normal work process (ie. no added reporting responsibilities to developers or their closest superiors). While some tuning of the prediction process parameters needs to be possible to adapt to changing circumstances, the basic use cases should be available to the manager with one or two clicks.

Prediction accuracy also needs to be better than what any existing ad-hoc process can produce. The cut-off point for this will likely vary from one organization to another, and may indeed vary considerably within even a single small (<10 people) team. A developer and manager who enjoy a good rapport built over a sufficient number of past projects can be able to come up with surprisingly good estimates, whereas a more troubled relationship (personal conflicts, varying levels of competence) may start at such a low level of estimation accuracy that almost anything is an improvement.

It is entirely possible that a good reporting/visualization tool for existing reporting data alone improves workload estimation and tracking considerably, even without explicitly predicting anything. Organizations often struggle to make use of data they already possess, turning that into useful knowledge might simply be a matter of presenting it better. Any actual prediction system should thus be able to offer something better than what most existing business intelligence (BI) tools can already provide.

3 Machine learning

Machine learning (ML) is a broad category of methods for solving problems that are difficult to approach with traditional, manually written software. The idea is to create algorithms that can be molded to different tasks by training them with different sets of input data, essentially learning a pattern from the training data that can be generalized to perform some task for as-yet unseen data [17] [18].

The machine learning field is essentially a subset of general artificial intelligence (AI). AI is a catch-all term for systems that mimic human intelligence, and machine learning deals specifically with the learning part of it. Learning is understood here as the ability to remember, adapt and generalize based on past experiences. Other aspects of intelligence, such as reasoning or logical deduction are less relevant here, though the demarcation between these can be a bit arbitrary at times [17].

It is important to note that the terminology can often get confused in public discourse, with terms like artificial intelligence, machine learning, deep learning, data mining, pattern recognition, etc. often being used inaccurately or interchangeably. This is particularly evident due to the field of AI experiencing an ongoing boom in interest, which brings in people and organizations not well-versed in the underlying science.

The field of AI came into existence in the 1940s, alongside the creation of the first digital computers, when the first mathematical neural network model was developed. The notion of intelligent machines has been a compelling one for scientists, fiction writers and the general public alike, with the field going through ups and downs as interest in the topic has waxed and waned. For scientists and engineers, the cycle has mostly been driven by development of new methods and the subsequent discovery of their limitations [19]. For example, the first algorithm representing an artificial neuron, the perceptron, was introduced in the late 1950s [20], and prompted a great deal of interest in artificial intelligence until the late 1960s when Minsky and Papert proved the limitations of the perceptron [21]. However, while general AI has proved to be incredibly difficult to achieve, there has been steady progress in more limited applications, including in various machine learning methods. Things like the nearest neighbor method, neural networks with backpropagation, decision trees, ensemble models etc. have proven to be quite adept at tasks like image recognition, sound recognition, and complex association analysis for large data sets, where conventional statistical and software engineering techniques have difficulty.

The latest AI/ML boom has largely been precipitated by the explosion of available data resulting from the wide adoption of computers and assorted smart devices in the last two or three decades. With people spending more and more time online and most if not all of this activity generating data, the past problems with scarcity of data have turned into a new problem of overabundance of it. Data itself isn't valuable, the knowledge extracted from it is, but extracting said knowledge is a far from trivial process. Machine learning methods have proved to be an effective way to find interesting patterns (knowledge) from this so-called Big Data [14].

3.1 Types of machine learning

There are several different types of machine learning, essentially defined by how the learning algorithm is supposed to improve (ie. learn) from data. The method of improvement has far-reaching implications on what kind of data is needed to train a model, and what kind of problems can the model be expected to solve [17].

3.1.1 Supervised learning

In this variant, our training data set includes the correct responses that are expected from the ML model. That is, we know what output the model is supposed to provide for a given training input. This data is fed to the algorithm, which afterwards will hopefully be able to generalize from these known input-output pairs the correct responses to new, previously unseen inputs. This is the most common type of machine learning process, and is naturally suited to classification and regression problems [17] [18].

Classification

In classification, the goal is to predict which of a limited number of classes/labels is the correct one for a given input. Here we generally assume that for each input there is exactly one correct class, and we know all possible classes an input can belong to. We train the classifier as described above, by feeding it example inputs and their matching correct outputs (classifications). Afterwards the model can hopefully accurately classify new, unseen inputs. Essentially we define a decision boundary (see figure 3.1), some function that separates the different classes from one another.

This kind of an approach can be used for example to find out which images in a set have some particular item in them, or what kind of a customer is interested in a given type of product [17] [22] [18].

The typical assumptions about us knowing all possible classes, and each input having exactly one correct class don't necessarily hold for all real-world problems. For example it is possible that something was overlooked in problem definition and there are in fact more classes than we know of. In this case it might be desirable for the classifier to be able to detect that some input is outside the normal operating envelope. This is called novelty detection, and if such a behavior is desired, it will need to be considered when training the model.

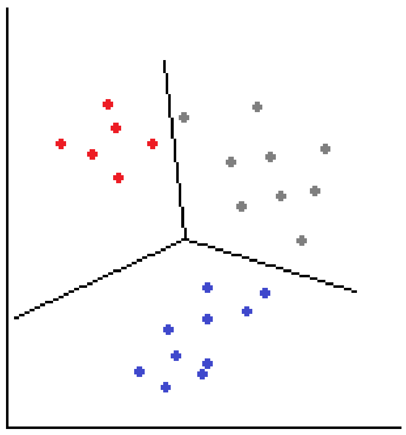

Figure 3.1: Decision boundary that separates three different classes. Note that here the separation is very clean, but in reality there tends to be some overlap. Here the boundaries are also straight, but there are techniques capable of drawing much more convoluted borders.

There is also the potential situation of a single input belonging to several classes at once. For this problem there are so-called fuzzy classifiers specifically built for such considerations. Incidentally, this kind of classification has in fact been suggested for use in software project estimation, as a way to account for the inherent uncertainties in the development process [17].

Regression

Otherwise a fairly similar problem as classification, but in regression the target value we're trying to predict is a number, rather than a set of discrete classes [18]. Common problems like these would be trying to predict the future value of something in the stock market, an interest rate, or some such.

For example, we might have a set of X and Y coordinate pairs, and we need to find out the value of Y for some given X value that isn't among the known pairs. We can try to find a function that seems capable of reproducing the known points (up to a point, typically we won't match the points exactly), and then see which

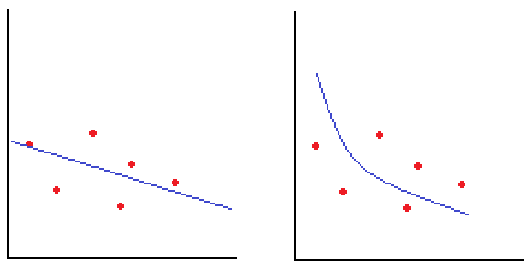

Figure 3.2: Different ways to fit a curve to a given data set. Predicted values will differ based on what kind of a function you select to describe the data set.

value the function would produce for the given X value. In figure 3.2 you can see how a different choice of function can affect the prediction a model will produce.

This kind of curve fitting is a fairly common technique in machine learning, and in fact some of the classical software estimation methods covered in the previous chapter use techniques like this [10] [18].

3.1.2 Unsupervised learning

Here our training data set does not contain the correct responses. The algorithm simply tries to find any similarities between individual items in the data, and thus hopefully describe at least some part of the underlying process that generated the data. This kind of machine learning is more exploratory in nature. Rather than trying to solve a very specific problem, we look for essentially any interesting patterns in the data. The advent of big data has considerably increased interest in unsupervised learning, as the newly available large repositories mostly contain unlabeled data [22] [17].

This type of machine learning is commonly used to solve problems like clustering and outlier detection. In statistics, this kind of analysis is known as density estimation [18].

Clustering

Clustering is the problem of dividing some data set into groups of items that meet some criteria for similarity. For example, we might look at a store's transaction data in an attempt to divide the clientele into groups based on items that they tend to buy together.

The exact criteria of similarity, and thus the kinds of clusters created vary greatly between different methods. Focus can be on things like how compact a cluster is, or how clearly it is separated from others. The result of any such analysis will invariably need some expert evaluation of its usefulness, as the methods themselves cannot guarantee that any associations or correlations they discover are actually of interest.

Clustering can also reveal outliers in the data, though again such cases will need to be carefully evaluated. Assuming the achieved clustering is considered to be satisfactory, items falling outside the found clusters can be considered outliers in some respect [14] [18].

3.2 Data, knowledge and preparatory tasks

As alluded to before, data and knowledge are separate things. Data can be understood as the record that something happened, or what the state of something was at a given time. Knowledge on the other hand covers things like why something happened, or how the state might change over time. The value of data comes from the knowledge that can potentially be extracted from it, which can then guide our actions later on. However, the data itself simply exists, and won't do anything by itself.

An important early step in a data analysis process is obtaining an intuitive understanding of what kind of data there is at hand. This might be less trivial than it sounds [14].

3.2.1 Data format and feature domains

Generally, data is understood as a matrix rows and columns, where rows represent instances and columns represent features. An instance for example might be something like a particular car, with the features being the car's model, color, number of doors, engine power, size of the gas tank, etc. Each feature can further be categorized into various domains [14].

Feature domains

Domain is essentially the definition of the type of values the feature can have. The domain of a feature has a considerable effect on how it can be used in the analysis process. There are three main domains: nominal (categorical), ordinal and numeric. The numeric domain has some further subtypes [14].

Nominal (or categorical) features' values come from a finite set. An example of such a feature could be the species of an animal (dog, cat, etc). Features like this can often serve well as labels for the data, so they are sometimes referred to as classes or categories. For nominal features, there is no further structure to the values than whether they are identical or not. There is no hierarchy between two different values, you can't for example say that one is greater than the other.

The issue of granularity might however come up with these features. That is, is it sufficient to operate at a rough level where values might be "electronics, cookware" or are more detailed ones like "phone, computer, etc." necessary.

Additionally, while the values come from a finite set, the set might not be static. In some cases, values can be added and removed as time goes on. This can have a significant effect on analysis results, as for an example the learning data set might be dominated by items that are no longer sold [14].

Ordinal feature values also come from a finite set, but there is a linear ordering to them. That is, you can say one value is greater than another. For example, school grades are like this. At the same time, it is important to understand that while an ordinal feature may be (and often is) represented by numbers, the only thing that can be said when comparing values is which one is greater. The actual numeric difference between values however typically doesn't mean anything. This is important to keep in mind to avoid inferring something from the data that isn't actually there. Operations commonly done with numeric features, like calculating the mean will produce meaningless values if performed on an ordinal feature [14].

Numerical features have number values. Numerical features can be discrete or continuous, and may have an interval, ratio or absolute scale.

As mentioned above, discrete numerical features can easily be confused with ordinal features, due to the superficial similarity in values. However, numerical operations like calculating the mean value are actually valid for discrete numerical features, even though the resulting value might not fit into the discrete value space.

The difference between interval and ratio scales is the definition of zero. For intervals, the definition is arbitrary, whereas for ratio scales some canonical definition exists. An absolute scale is defined by a unique measurement unit, and the meaning of the field is dependent on that (for comparison, ratio scales retain their meaning even if the unit changes, since it's all in relation to the canonical zero).

Complex data types: In addition to the main feature domains discussed above, there are numerous other data types that don't fit into such concise categories. The generic term for these is complex data types. An exhaustive list can't feasibly be provided here, but some examples are text (ie. natural language, rather than categorical labels), graphs and images. Highly structured data like this generally isn't suitable for most analysis methods as-is, so it is usually necessary to either extract some kind of simpler features from this data, or adapt the methods being used to deal with the specific kind of complexity exhibited by the data at hand [14]. Feature extraction/engineering is covered in section 3.2.3.

Dimensionality

Dimensionality refers to the number of features we have in our data. While generally having more training data is considered a good thing, more features (ie. higher dimensionality) can be problematic. For starters, early-phase data understanding efforts often revolve around visualizing the data, ie. plotting it and looking for any interesting correlations. One graph can't effectively show more than two or three dimensions however, and while you can work around this by drawing multiple pair plots for example, or reducing the dimensionality with techniques like principal component analysis (PCA), interesting information can still be missed as a result [14] [17].

Another consideration is that not all features are actually useful. Some features are simply redundant or irrelevant for some particular analysis. In this case, there's no benefit to having them around, and some ML methods (like k-nearest neighbors) can actually perform significantly worse with such features present in terms of accuracy. Higher dimensionality will also of course increase the computational complexity of method application, though how much of a problem this actually is will depend on how the method in use scales in this regard, and what kind of processing resources are available [18] [14] [17].

Dimensionality of data also affects how much data we actually need to train our model. The higher it is, the more data is required in order for the model to learn to generalize well. This phenomenon is known as the curse of dimensionality [17].

3.2.2 Data quality

The most important question when starting a data analysis process is whether the knowledge we seek can actually be found in the data we have. In addition to understanding exactly what data we have, we must also form an understanding of the quality of the data, that is, how accurate, complete and relevant the data set is. If accuracy is poor and/or the data set doesn't properly represent the situation we're trying to model, then our analysis isn't likely to produce useful results even if theoretically the kind of records and features we have should contain the knowledge we seek [14].

Accuracy

Accuracy means how close a value in the data is to the true value. There are numerous causes for why a given feature value might deviate from the true value. For numerical features, the instrument used in measuring said value might be limited in precision, or there might be some source of noise. For categorical attributes, values might be misspelled. How difficult such issues might be to detect and correct or compensate for depends on what kind of inaccuracy we're dealing with. There are two kinds: syntactic and semantic.

Syntactic accuracy means that values are within the acceptable bounds for a given feature. For example, if a feature is supposed to contain integers between 1 and 5, values like 7 or 3.6 would be obviously wrong, as the first is out of bounds and the second isn't an integer. These kinds of errors are quite easy to detect, though what is to be done with such values will still need some consideration.

Semantic accuracy means the actual correctness of the value. A syntactically correct value can still be semantically wrong, for example in the aforementioned case of integers between 1 and 5 being acceptable, a value of 4 would be syntactically correct but if the true value is 3, it would still be semantically wrong. Semantic accuracy is difficult to check, and can in fact be completely impossible in some cases. Domain knowledge, data visualization and other techniques can still give us some idea of what kind of semantic inaccuracies we might be dealing with, even if we can't easily find them all [14] [18].

Completeness

Completeness deals with the question of what is missing from our data. Records, features, feature values, or some combination of the three? Each has its own implications on the accuracy of our analysis results, and recognizing that something is missing (and what that might be) is a non-trivial task.

The simplest case is when feature values are explicitly marked as missing. That is, the field contains some defined marker for a missing value (null, n/a, etc.). However, some times the missing value is simply denoted by some default value, for example a zero in a numerical field, which can still be a syntactically and semantically correct value for that field. In this case we're left wondering whether the zero is an intended, valid value (measurement or some such), or just something that was filled in lieu of an actual measurement.

Whole records can also be missing from the data. It might be that certain records were lost over time in data migrations, or maybe the data wasn't collected in the first place for some reason. This can be difficult to detect, because we're quite literally looking for something that isn't there. Sometimes the absence of records can be inferred from the data, typically by reflecting against domain knowledge [14].

It might also be that our data set doesn't contain some important/useful feature at all. It might not have existed in the data set in the first place (ie. it wasn't recorded to begin with), or we might have even discarded it ourselves during feature selection (see below) because it didn't appear important. In such a case we might still be able to usefully predict something via another feature or combination of features, and the effect of the neglected (hidden, latent) feature looks like noise in the analysis [18].

Fundamentally, completeness is a part of understanding how well the data set at hand actually represents the real-world situation. The missing data itself (be it values, records or features) might form a pattern, and if it does, understanding it is crucial because it likely means the data set is biased in some way. If we understand the circumstances that cause a value or record to be missing (temperature sensor only works above a certain temperature, for example), we can either compensate for it or limit our analysis conclusions to situations we actually have data for. On the other hand, if the data is simply missing at random, there's less need to correct for it, since it shouldn't affect the actual value distribution much as long as the data set is sufficiently large [14].

Either way, it is necessary to consider how to deal with the missing data. Should we fill in some kind of default values or best guesses to empty fields? Should records with missing values be dropped entirely, or otherwise ignored in analysis? The answers will always have to be decided on a case-by-case basis, keeping in mind what kind of an effect they will have on the distribution of the data.

Other data quality issues

Unbalanced data sets can be a problem. If for example we know the records in our data set fall into two different classes, but the vast majority of the data belongs to the first class and the second is barely represented, we'll obviously have trouble predicting the second class, since training examples for it are much rarer. This kind of a situation isn't uncommon in industrial production for example; the majority of the time the machines are running as expected, so any sensor logs are mostly going to contain data about normal operations, rather than fault situations. This is a somewhat related concept to completeness, as a pattern of missing data can result in an unbalanced data set.

Another potential issue is timeliness, ie. is the data sufficiently recent to still be useful? While some data might be useful almost indefinitely, in practice it is common that old data is less worthwhile to analyze. As business processes evolve, old data eventually ends up describing situations that no longer exist. For example, past a certain point a store's transaction records may contain products that are no longer sold, which might not be useful for analyzing current market trends [14].

Outliers

Outliers are feature values, value combinations or even whole records that fall significantly outside the constraints that the majority of a given data set follow. There is no universally applicable formal definition for them beyond that, and neither is there any universally reliable detection method for them. They may be simply erroneous data, but they might also be correct data produced by some rare circumstances.

For single attributes, outlier detection can be done with visualization tools like boxplots, or statistical tools like Grubb's test. For multidimensional data, visualization tools like scatter plots can be used, possibly combined with some kind of a dimensionality reduction technique like PCA. It's worth noting that while these techniques may suggest values/records that might be outliers, any findings will always still have to be judged on a case-by-case basis because of the inherent looseness of the concept of an outlier.

The handling required by outliers once identified is as varied as the potential detection techniques. If the values can be shown to be simply erroneous, they can be rectified or removed from the data, but even non-erroneous outliers may require exclusion. This is because some machine learning methods are especially sensitive to outliers. That is, even just a few wildly abnormal values/records in the input can have a considerable adverse effect on result accuracy if the used method isn't sufficiently robust with regards to outliers.

Alternatively of course, the outliers may in fact be the precise information we're interested in, and we might specifically tailor our method selection to cater to this need [14].

3.2.3 Data preparation

Data preparation consists of various tasks that we perform on the data in order to improve its quality and thus give the actual modeling phase the prerequisites for success. Care must be taken here to ensure we don't inadvertently bias the data set in some fashion; in extreme cases even data that actually has no signal at all can appear to yield some insight in the analysis phase if some part of data preparation is done carelessly [14].

Data/feature selection

The first task is selecting the records we wish to use for modeling. While generally we wish to have as much data as possible, in practice all the data at hand may well not be suitable or relevant to analysis. A good example is data that falls afoul of the timeliness issue mentioned previously. If the data contains entries that are deemed too old to accurately represent the current situation, it is best to exclude those from further analysis.

Similarly, not all features in the data are necessarily useful. We might wish to exclude an entire feature column if the data in it is deemed too noisy or sparse. Alternatively a feature may simply be irrelevant or redundant, even if the values themselves are deemed sufficiently accurate. Determining whether a feature is potentially useful or not can be done in many ways, but the fundamental goal is to pick features that we think contain the most useful information, and to discard the rest. This will simultaneously reduce the dimensionality of the data [14] [18].

One way to do feature selection is via subset selection. The idea is to find the best performing subset of a given set of features. Performance is measured via some suitable error function. In practice, unless our feature set is very small, we can't feasibly test every single subset, but with suitable heuristics we can narrow the selection down to a feasible number of good enough subsets. The actual selection can be done either forwards or backwards. In forward selection, we start with no features, and we add features one by one, until we reach the point where new features no longer reduce the error. In backward selection, we start with all features in the set, and remove them one by one until we reach a point where removing any more would increase the error measure [18].

Data cleaning

Cleaning means correcting assorted simple errors that have been identified in the data earlier in the process. Common actions at this point are things like unifying the format of numeric values and dates, fixing spelling mistakes in categorical values as well as leveling case sensitivity, and splitting fields with mixed information (ie. a "weight" field with 100g as value turns into "weight" and "unit of weight", with 100 and g values respectively). The goal is to ensure the data is consistent in format.

At this point missing feature values can also be dealt with. As mentioned before, how exactly these are handled will need to be determined on a case-by-case basis, but common techniques are removing the whole record from further analysis, imputing some suitable default value (mean value of the column for example), or marking the field with some explicit value that denotes the value is missing.

Generally it is a good idea to document any changes made to the data. Changes that seemed reasonable at the time can turn out to be problematic later on, or may simply have an impact on the final analysis results that needs to be known to put the result in a proper context [14].

Feature engineering

Feature engineering covers techniques for transforming the existing data into some desirable format, and generating entirely new features from the raw data. There are three typical goals for this. To ensure the data is in a format that our chosen analysis method can actually use, to curtail the disproportionate influence that larger magnitude features can have, and to improve analysis efficiency in general.

Most machine learning methods have some limitations on what kind of data they can process, so at a basic level, we'll need to ensure all our data fulfills such criteria. All input values might need to be numbers for example, in which case categorical feature values will need to be represented by numbers somehow. With such transformations, it is important to consider what kind of operations are going to be applied to the feature, and whether they will be meaningful in the end. For example, transforming a categorical variable into a sequence of numbers is a fairly trivial task, and such design is commonly seen in database tables, but for data analysis this can have unwanted consequences if mathematical operations like calculating the mean are performed on the feature column. The operation can be done, but the result will likely be meaningless or outright harmful, since it implies certain assumptions about the values having an order and distance.

Raw numeric feature columns will often contain wildly different value magnitudes. This can cause the features with larger magnitude values to dominate the analysis, even though in reality the particular feature's importance doesn't warrant such. This issue is typically rectified by normalizing the feature values, with techniques such as min-max normalization, z-score standardization or decimal scaling. The idea is to retain the features' individual distributions, but scale the actual values so that the overall scale across different features is roughly the same.

Efficiency improvements are perhaps the hardest to formally pin down. Beyond basic operability, it is worth considering how well a model can learn from some particular representation of data. For example, in the practical section of this thesis we will deal with software project logged hours, which sometimes contain the planned deadline of the project as a date. While a neural network might eventually be able to learn the connection between that and the date of the logged hours, the amount of data needed for such training might be unfeasible. On the other hand, a simpler model like k-nearest neighbors is unlikely to make such a connection in the first place. In this case, it could be useful to transform the planned end date field into a "planned days left" field, where we calculate how many days are left from the logged date to the planned end date. The original information is still essentially there, but in a more easily digestible format [14].

Among other things we might do to improve efficiency would be to eliminate observed non-linearities via some suitable mathematical operation, or apply techniques like PCA to hopefully retain the most important information (for PCA this means preserving variance) in a high-dimensional data set that is intuitively difficult to understand [18].

3.3 Model selection

Model selection covers two related tasks: selecting the model class (aka method) to use, and selecting a particular model produced by that class.

Model class selection is based on our understanding of the analysis goal and available data. The goal itself will narrow the choice down somewhat, since most classes are only suitable for certain kinds of analyses, but generally this will still leave a number of methods for consideration. Some common criteria for further narrowing the choice down are [14]:

- **Data characteristics:** Does the data set contain a significant amount of outliers or non-linearities that we can't or don't want to remove in the data preparation phase? Is the data noisy or sparse? Different model classes have varying degrees of sensitivity to such issues, so naturally we should try to select a model class that is robust against the issues we know to be present.

Data set size is also something to consider. Complex models such as neural networks generally require more data to perform well, so if we deem our data set to be too small to support such a model class, we will have to restrict ourselves to simpler classes like k-nearest neighbors or linear models.

- **Interpretability:** Different classes of models vary wildly in how well they explain the predictions they make. A decision tree for example can be followed from root to leaf, to see exactly what kind of decision rules produced the final result. Neural networks on the other hand are essentially opaque to this kind of exploration. In many cases so-called black-box models like this are unacceptable, because the explanation is often deemed as important (if not more so) than the actual prediction/decision. Indeed, making unexplainable decisions can be outright illegal in some circumstances, like in the field of medical care.

- **Computational complexity:** Model classes differ in what kind of computational resources (processing power, memory, etc.) are needed to train and use them. While cloud computing has made computational resources more easily available, the time and cost of some complex models can still be prohibitive.

In practice the tasks of selecting the class and individual model tend to overlap somewhat. That is, we'll likely do at least some preliminary experiments and evaluation on models from multiple classes, before focusing on finding the best model in a single class. This is because even careful consideration of the aforementioned model selection criteria often won't narrow the class choices to one, and estimating which class can produce a better model isn't very intuitive without running some kind of evaluations, to have some preliminary performance data to guide the class selection.

Model selection means finding the best configuration of the selected model class. What this entails in practice is very dependent on the class, but typically involves finding the best values for whatever class-specific hyperparameters there are. This can mean something as simple as finding the best-performing k-value and distance measure for a k-nearest neighbors model, or something as complicated as coming up with a good setup for an artificial neural network (number of neurons, layers, training system, etc.) [14].

The goal is to find a model that generalizes well, not one that simply memorizes the training data. This means that model selection is something of a balancing act; you want to find a model sufficiently complex that it is capable of capturing the underlying pattern in the data reasonably well, but no more complex than that. Too rudimentary a model is unable to find the pattern, this is called underfitting. A too complex model on the other hand will capture the individual training data points (including any inaccuracies and noise) but not the pattern, this is called overfitting [17] [18]. Figure 3.3 illustrates the issue.

How model selection happens in practice will depend on what kind of learning are we doing and what kind of result domain we're operating with.

In supervised learning, model selection happens by fitting (training) different models and checking how well their results (predictions) match with the known target values. How this matching is done will depend on the domain of the values we're predicting; in case of numeric values we might calculate the mean square error from the predicted and true values, and then find the model that minimizes that. In case of classification, a basic metric could be classification accuracy, that is, how many times the model predicted the correct class. Various ways of measuring model performance, and assorted pitfalls involved in the process are covered in more detail

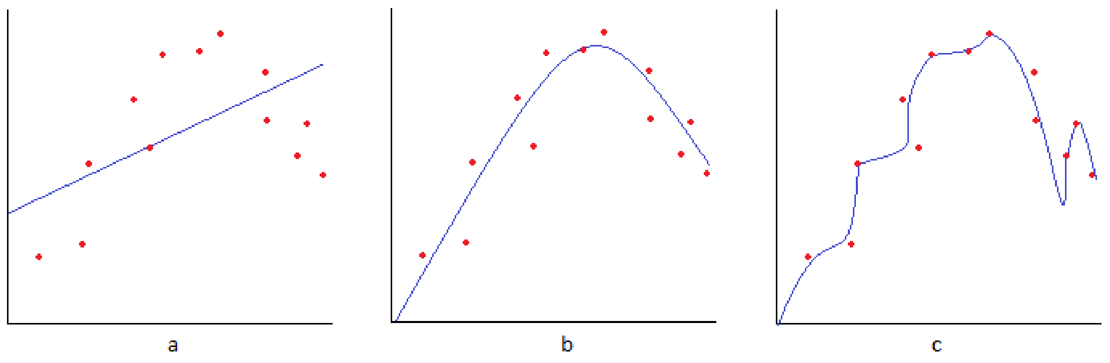

Figure 3.3: In plot a, our model is too rudimentary and underfits, ie. fails to capture the pattern from the training data. In plot b, the model appears to capture the general pattern reasonably well. In plot c, the model is too complex and overfits, ie. appears to be be learning noise. Notice that the model in plot c would still have the lowest measured error between the model predictions and and the training data points, but it'd likely generalize poorly to previously unseen data.

in section 3.3.2.

With unsupervised learning, the process is somewhat less well defined, simply because we don't know the correct values for our training data. As such, the process revolves more around trying to determine if the patterns found appear interesting or coherent. If the goal is clustering, we might for example concentrate on how clearly the found clusters are separated from one another, or how closely-packed they are. Tools like calculating the silhouette score might be used here [14].

Another thing that needs consideration is how exactly is the training done. For some simpler methods this might be essentially a non-issue, but with neural networks you might get very different results depending on whether training data is input sequentially or as a batch. The former means that the network's neurons adjust their weights (learn) after every single input, whereas in the latter the learning only happens after a larger batch of training data has been processed. The order or distribution of the training data itself might also be altered during training, to focus more on cases that the nascent model is doing poorly on [17].

3.3.1 Model class examples

k-nearest neighbors

The method of k-nearest neighbors is based on the idea that the value of a given attribute can be derived by looking at known attribute values in other records that are similar to the one being predicted [23] [24]. Similar records (ie. closest neighbors) are found by calculating some distance measure between records in the data set, and choosing the k nearest based on that. Once the k nearest neighbors are found, the prediction is formed by having those vote on the value. This nearest neighbor method can be used for both classification and regression, the only difference is in how the final vote is conducted. For classification, typically the majority vote is selected, whereas for regression the value is typically the mean of the neighbor values.

In terms of model selection, the most important choice with kNN is selecting the value of k, ie. how many neighbors are included in the final vote. The behavior of the model changes considerably depending on the selected value, with low values being prone to overfitting, and too high values essentially producing averages of the whole data set. The best value of k is usually found simply via trying different values and seeing which produces best results according to some suitable performance metric. This is commonly done with cross-validation. Some other model parameters can also be tuned, like the type of distance measure to use (typically the Euclidean distance between feature values is used), and whether all neighbors should have equal weight in the final vote or not.

This method is simple and easy to apply, and can often perform surprisingly well compared to more complex methods. Some drawbacks are that it cannot perform any feature selection, so redundant and irrelevant features can compromise prediction accuracy, and the model runs into performance issues with a large number (in the thousands) of features. KNN also cannot handle missing feature values, so those either need to be filled in somehow, or the records with missing values need to be dropped from the data set. The model is sensitive to the scale of feature values, so input data should be for example z-score standardized to ensure all features contribute to the distance measure equally [18] [22].

Decision trees and random forests

A decision tree is essentially a sequence of questions about the features in our data. We start at the root, and each answer to a question directs us down one of several paths, until we reach the leaves of the tree, where we obtain some prediction/decision (see figure 3.4). They are most commonly used for classification and regression, though clustering applications are also possible.

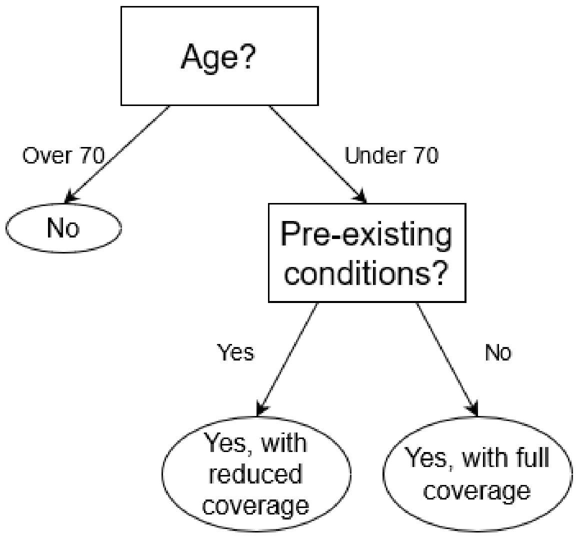

Figure 3.4: Example of a simple decision tree for determining whether to sell a health insurance policy to a potential customer, and under what terms. Each answer directs us further down, until we reach a decision.

There are several ways to construct decision trees. Most are variations of the general theme of starting at the root, and selecting the best feature to evaluate at each node. Comparing features in this respect can happen in several ways as well, with information gain and the chi-squared test being some common ones. In case of continuous attributes, suitable splitting points also need to be determined. Furthermore, as tree depth increases, the risk of overfitting grows, so it might be necessary to prune the initial tree into a more simple configuration, by replacing problematic branches according to some criteria.

For more complex cases, several trees can be combined into an ensemble model, known as a random forest [25]. Here the idea is to train several trees on different samples of the data with randomly chosen attribute subsets, and combining their individual predictions into a final result.

Trees deal well with sparse data, and forests can also handle high-dimensional data quite well. Trees are also quite easy for humans to interpret (ie. it is easy to tell how some prediction was reached by following the tree to the relevant leaf), which makes them suitable for applications like the medical field, where unexplained decisions are not allowed. Forests are less so, though still a considerable improvement on models like neural networks. Decision trees have also been applied to software project estimation, with the so called CART model (Classification And Regression Trees) [17] [10].

Multi-layer perceptron

The perceptron [20] was the first model of an artificial neuron, and the multi-layer perceptron (MLP) is an evolution of that concept. It is a network of multiple perceptrons arranged into a number of layers. Unlike the perceptron, a suitably configured MLP is capable of solving linearly inseparable problems, and forming nonlinear mappings in general.

Essentially, an MLP consists of an input layer, output layer, and one or more hidden layers of neurons between them. Each neuron has a number of numerical weights that are applied to its inputs, and some defined activation function (often each neuron in a given layer has the same one), that maps the neuron's weighted inputs into some output value. Training the model consists of feeding it data, and then adjusting the weights in each neuron based on some error calculation. This is typically done via the so-called back propagation method, where errors are calculated at output, applied to the output layer's weights, and then proceeding from that to update the next last layer's weights, and so on until the input layer is reached (ie. error propagates backwards through the network).

MLP is a universal approximator, that is, given sufficient data it can learn any problem. It is quite prone to overfitting, so care needs to be taken to stop the training before that happens. MLP training is also computationally intensive and requires a very large amount of data (old rule of thumb being that it needs 10 times as many training examples as there are weights in the model) [17].

3.3.2 Performance evaluation

Finding out which class/model performs the best requires some way of comparing models, that is, to evaluate their performance against each other. Additionally, we of course want to have an accurate assessment of our model's ability to generalize with unseen data. It is important to understand that while same techniques can be used for both of these tasks, they are still separate tasks. Depending on how model selection is done, it is possible the performance metrics obtained during the selection process don't accurately reflect the generalization performance, even if they do indicate which model is likely to be better.

There are numerous ways to measure model performance, and even more special circumstances that may need to be considered/accommodated to get a reasonably accurate performance estimate. Realistic performance estimation requires that the validation/test setup accurately reflects the actual planned use case for the model.

Furthermore, the old adage "you get what you measure" very much holds in this regard, since model selection as a task is essentially optimizing the result of some performance measure. If the performance measure doesn't represent reality well, then the model selected based on that measure is unlikely to be of much use either [14] [17].

Since we're interested specifically in the model's ability to generalize to data it hasn't seen before (during training), at a minimum we're going to need a set of testing data separate from the training data. We'll train the model with the training data, then see how well it predicts the relevant values for the test data. There is no hard and fast rule for how to split the data into training and testing sets, but usually the training set is larger of the two, comprising of 2/3 of the whole set for example. For some scenarios however, training and test sets aren't enough.

Consider model selection: We train a number of different models with the training set, and then measure their performance against the test set. We select the model with the best score. While we may now have the best model, what we don't have is an accurate estimate of its performance, because the model was specifically selected based on its performance on the test set. That is, while the measured performance is likely better than what the other tested models had, it's likely over-optimistic with regards to unseen data. To rectify this, we need yet another set of unseen test data. This would mean splitting our data set into three parts; training, validation and test sets. We train the model with the training set, we perform model selection against the validation set, and we obtain the final performance estimate by running the selected model against the test set [14] [17].

The problem here is that there might not be enough data for all of this. When working on some Big Data project, this likely isn't an issue, but in many cases the data set at hand is small enough that splitting it into three parts will reduce the amount of training data to an unacceptable degree. A common technique for overcoming this problem is cross-validation (covered later in this section) [17].

When splitting the data into training, validation and test sets, the requirement for good representation of the real-world use case needs to be kept in mind. For an example, if we're solving a classification problem, then the distribution of the classes should generally be about the same in all data sets. If each class is approximately as common, then just splitting the data randomly may be acceptable. However, many data sets are skewed in some way. If the data set contains two classes, A and B, but A makes up 90% of the data, then a split that doesn't take this into consideration may result in a case where the B class is almost or completely missing from training or test data. This will of course result in problematic performance evaluations, because either the model hasn't seen the minority class at all in training, or its ability to distinguish the minority class hasn't actually been tested at all. This can be rectified via stratification [18], ensuring that the proportion of classes is equal in each set.

A common assumption in ML performance evaluation is that the data is independently and identically distributed (IID). In intuitive terms, this means that a single record in the data doesn't tell you anything about the other records. For a practical test case, this would mean that a model fitted with some training data set and tested on a separate test set essentially hasn't "seen" the test data, and thus its ability to predict the test data is a realistic measure of the model's generalization performance. There are plenty of cases where the IID assumption doesn't hold however [26].

For example, say we have a number of samples of various fluids, and our data consists of measurements taken from those samples, but we know our measurement process produces somewhat noisy values. To mitigate the effect of noise in the analysis we might repeat each measurement several times per sample. This means that several records in our data set are in fact related, as they were taken from the same fluid sample. If this isn't considered in the data split, we'll end up with a poor performance estimation. If our real-life scenario is a case where we don't expect to have actually measured a particular fluid before, then our performance estimation will be overly optimistic. In the data split, this situation can be simulated by ensuring that all measurements of a given sample are either in the test or training set, but not split between both.

Cross-validation

Cross-validation (CV) means splitting your data into several sets called folds, and then training a model with all but one of the folds, and using the last fold for testing. The process is then repeated so that each fold is left out of the training set in turn, with the end result being that the model has been tested against all records in the data (see figure 3.5). The number of folds can be selected on a case-by-case basis, generally driven by processing time constraints. The more folds, the longer the processing takes, with the extreme case being where each record alone is a fold (so-called leave-one-out cross-validation). The more common case is called k-fold cross-validation, where k is the selected number of folds [17] [18].

Cross-validation allows you to get the most out of a limited-size data set, but the usual considerations about data splitting still apply. That is, for unbalanced data stratification should be used, and any possible underlying connections in the data will need to be considered in terms of whether they can bias the performance estimation. Cross-validation assumes that the data is independently and identically distributed (IID), so any violation of this principle will need to be considered in the data split [26]. For example, time-series data is not IID, as records closer to each other in the timeline tend to be more similar than records further apart.

Furthermore, in order to get useful results from CV, the learning algorithms used should be relatively stable, ie. a small change in input results in a small change in

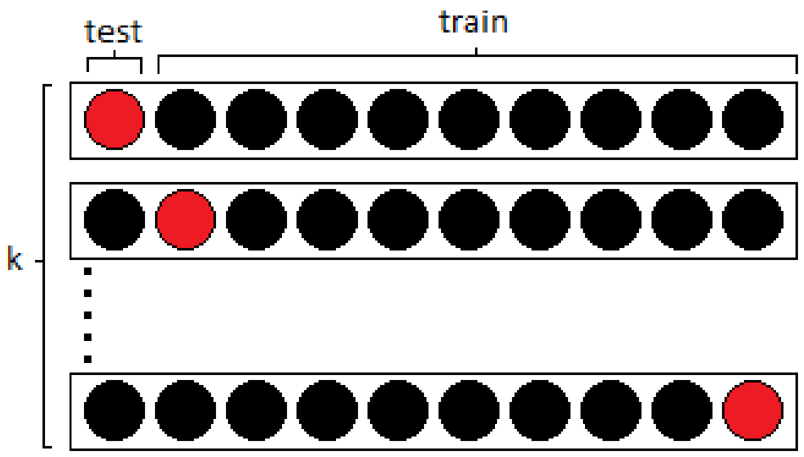

Figure 3.5: k-fold cross-validation. Each rectangle represents the whole data set, with each sphere being a fold of data. For each round, we set aside one fold (red sphere), train a model with the remaining data (the black spheres), and then test the model against the fold that was left out. Process is repeated for each fold, so k times in total.

output.

Confusion matrix

For classification problems, a simple performance measure is how often the model predicts the correct class for a test set. However, beyond this basic accuracy measure, we might be interested in what kind of misclassifications the model most commonly does. This can be visualized with a confusion matrix.

The idea is simple. We create a matrix where all classes are listed on both horizontal and vertical axes. The horizontal coordinate represents the predicted class, and vertical represents the true class. Thus correct predictions are along the diagonal starting from top left square, and other coordinates show how often a particular misclassification happened. Figure 3.6 shows an example confusion matrix.

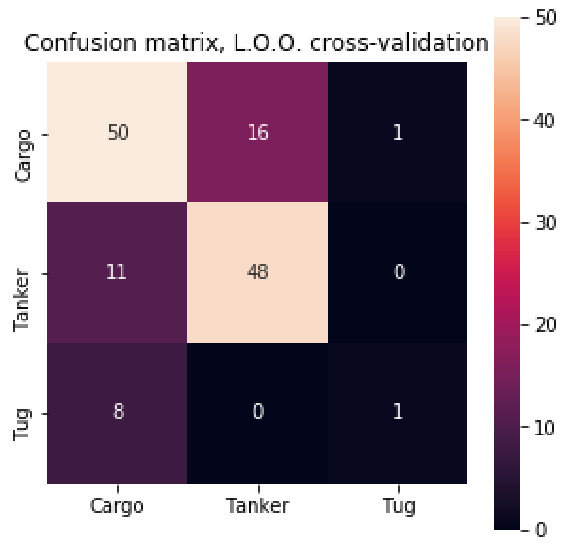

Figure 3.6: A confusion matrix obtained in an exercise where ship types were to be predicted from various measurements (size, speed, etc.). Leave-one-out cross-validation was used to test model performance, and the result was plotted in a confusion matrix. We can see cargo ships and tankers were identified with fairly good accuracy, but tugs were misclassified most of the time. This was a fairly expected result, since the data set contained a very limited number of tugs, so the model had very little training data to go by.

This information can be useful for understanding in more detail the particular weaknesses a given model has. From this information we might be able to identify ways to improve our model fitting process, or at least know which particular classes a generally well-performing model can't be expected to predict with good accuracy. A further consideration could be differing costs assigned to different misclassifications, this kind of visualization is known as a cost matrix [17].

Classifier accuracy metrics, ROC and AUC

With a binary classifier, the resulting confusion matrix is 2x2 in size, and will essentially show how many true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) the classifier produced. In addition to simple classification accuracy, we can calculate other measures like sensitivity, specificity, precision and recall (equations 3.1 - 3.5, "#" here is shorthand for "number of").

$$Accuracy = \frac{\#TP + \#TN}{\#TP + \#FP + \#TN + \#FN} \tag{3.1}$$

$$Sensitivity = \frac{\#TP}{\#TP + \#FN} \tag{3.2}$$

$$Specificity = \frac{\#TN}{\#TN + \#FP} \tag{3.3}$$

$$Precision == \frac{\#TP}{\#TP + \#FP} \tag{3.4}$$

$$Recall = \frac{\#TP}{\#TP + \#FN} \tag{3.5}$$

These can be used to track model performance in areas that classification accuracy alone won't reveal, and help optimize model performance in areas considered to be the most important for a given analysis task. Of course, care must be taken to understand just what are we optimizing for. For example, if the model predicts everything as positive, then there won't be any false negatives and thus recall will be 100%, but precision will of course be very poor [17].

These measurements can be used to compare classifiers in a number of ways. One is to plot the Receiver Operating Characteristic (ROC) curve, which is basically the classifier's sensitivity (AKA true positive rate, TPR) against its false positive rate (FPR) at various decision thresholds, and calculate the area under the curve (AUC). Different ROC curves can be seen in figure 3.7.

This is a common way to compare diagnostic tests in medicine, but has recently seen wider adoption in the performance evaluation of machine learning models as well [17].

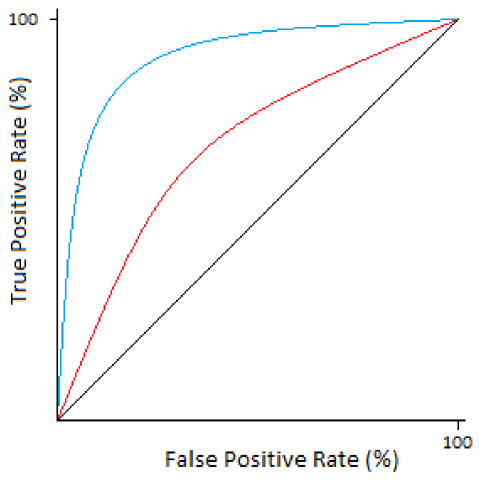

Figure 3.7: The diagonal line represents a classifier that is essentially making random guesses. The red and blue lines represent increasingly accurate models, which are able to achieve higher true positive rates before the false positive rate climbs to unacceptable levels.

Mean square error, mean absolute error

When working on regression problems, it is necessary to know how far the prediction was from the target value, rather than just whether the prediction was correct or not. Indeed, the latter would result in a useless performance measure, because in practice regression almost never produces predictions that hit the target square on.

The most commonly used regression error measurement is the mean squared error (equation 3.6). If we're trying to fit a line $y = ax + b$ to some data set, then MSE is defined as

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(ax_i + b - y_i)^2 \tag{3.6}$$

As the difference between the target and the predicted value is squared, this measure is somewhat sensitive to outliers in the predictions. A single really poor prediction can result in a very large mean square error even if the model otherwise predicted fairly well.

Another common measure is the mean absolute error (MAE, equation 3.7).

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|ax_i + b - y_i| \tag{3.7}$$

Things like mean Euclidean distance between a data point and regression line, or simply the maximum error encountered with a model might also be considered [14].

Concordance index

Unlike measures like mean square/absolute error which depict the amount a prediction deviates from the true value, concordance index (or c-index for short) measures how well the predictions are ordered. That is, the actual predicted values for samples A, B and C aren't important, only their order with regards to one another. If the predicted values A', B' and C' retain the original order of the true values, then a perfect score of 1.0 is produced by the c-index metric (see figure 3.8). A score of zero indicates the original order was reversed by the model, and a score of 0.5 indicates that the model is essentially producing random guesses [27].

This way, c-index measures how well the overall trend is captured, but the individual predictions' accuracy isn't a concern. For software project effort estimation, we'd ideally want a model that produces fairly accurate estimates of project durations, but even a model whose predictions are inaccurate but well ordered may be of some value. While it can't be used to plan the actual duration of an individual project, it can still describe fairly well the order in which different projects are likely to be completed. Furthermore, knowing that a model is inaccurate in absolute terms (measures poorly with MSE/MAE) but good at ordering samples may allow us to direct our efforts at improving the model in some way, rather than discarding it outright.

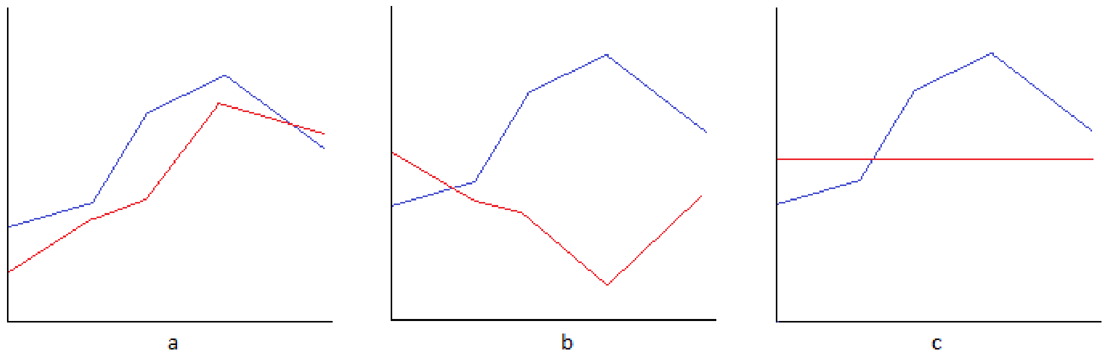

Figure 3.8: Different scenarios for c-index calculation. The blue line represents the true values, the red line represents the values predicted by some model. In plot a, the model has captured the trend perfectly, even though the individual predictions are inaccurate to some degree. The c-index score here would be 1. In plot b, the model has captured the trend in reverse and corresponding c-index value is 0. In plot c, the model has learned nothing at all, and c-index value would be 0.5.

Baseline comparisons

When working with various performance evaluation methods, it is a good idea to occasionally do comparisons between the machine learning models being worked on, and absolutely basic prediction methods like random guesses, majority voters, and the like. Surprisingly often an ML model fails to outperform such methods, in which case it is necessary to re-evaluate the whole process thus far.

For example, if the model achieves a 90% classification accuracy on some data set, but a majority voter (model that always predicts the most common classification in the data set) also achieves 90% accuracy, our model isn't actually useful. In fact, a closer inspection of the model's predictions in such a situation may reveal that it has learned to behave like a majority voter. With such an unbalanced data set (one would hope such a glaring balance issue in the data would've been noticed earlier, but better late than never of course), something in our modeling process needs to be changed in order for the model to actually start catching the minority classifications, otherwise the whole exercise is pointless.

3.4 CRISP-DM

In broad terms, a typical data analysis process consists of the following phases: Data collection, feature selection/extraction, algorithm choice, model selection and evaluation [17]. For simple, small-scale projects, some kind of an ad-hoc process made up of these parts might be sufficient, but in a more complex case, something more structured is necessary. There are a number of such process frameworks, but the most commonly used one is the CRoss-Industry Standard Process for Data Mining, or CRISP-DM [14]. An overview diagram of CRISP-DM is presented in figure 3.9.

An important thing to keep in mind is that the CRISP-DM process isn't expected to be run in a waterfall-like fashion, rather it is expected that each phase will likely need to be performed several times. This is because initial assumptions about the project are often proven incorrect in some fashion as the analysis proceeds, which will necessitate returning to previous phases to revise goals and redo various tasks based on the newly discovered information. The most typical phase rollbacks are included in figure 3.9, but in practice it is possible to end up returning from any phase to any preceding phase [14].

Project understanding is somewhat similar to the typical kick-off and planning phases in software development. The goal is to understand what exactly is the target or expected utility of the analysis project and what kind of a solution are we looking to produce. General understanding of the problem domain is also sought, as typically data analysts aren't familiar with the particular business environment a new project resides in. Conversely, the domain/business experts typically aren't well-versed in data analysis and machine learning, so effective communication between the parties can be an issue (one typically encountered in software engineer-

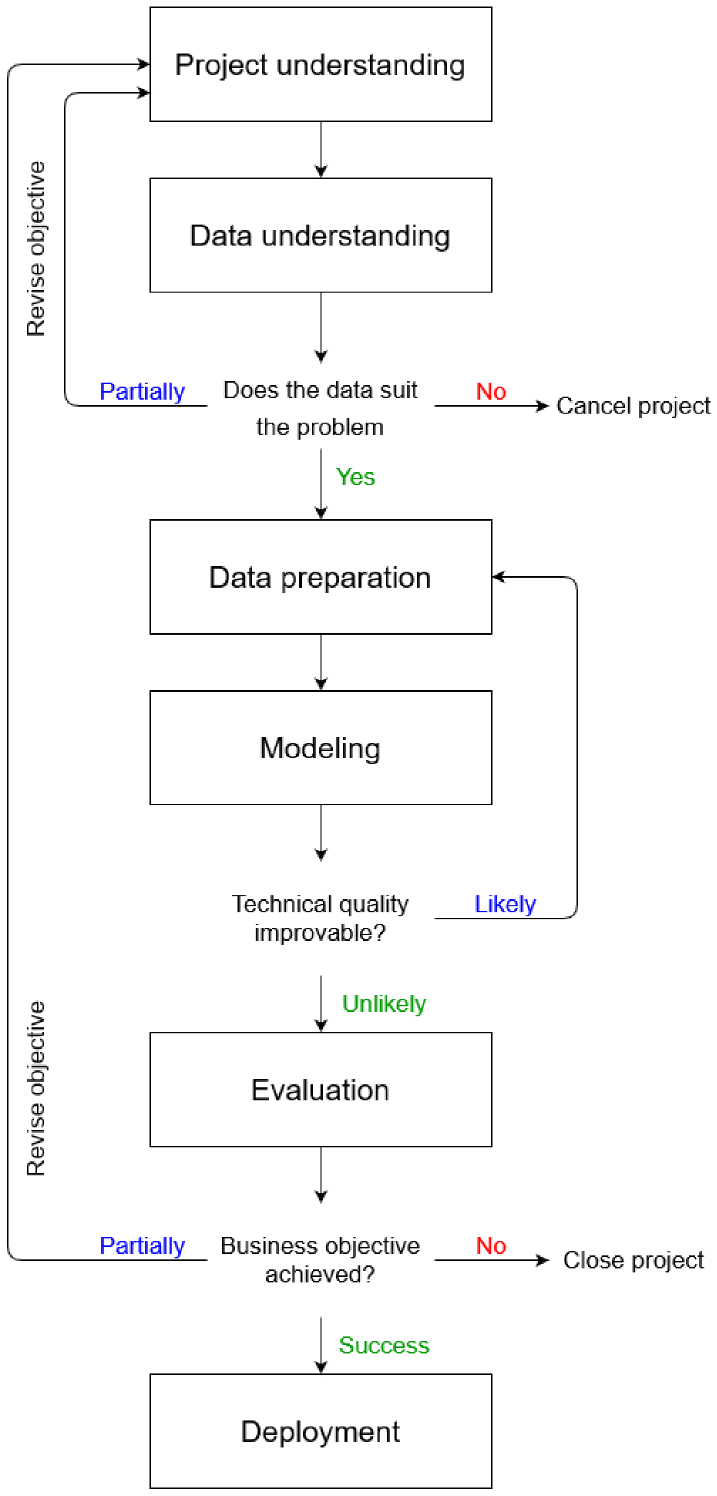

Figure 3.9: The CRISP-DM process, adapted from [14]

ing at large). This can potentially be somewhat eased by the different parties and stakeholders establishing some kind of a common understanding of the project.

Data understanding seeks to find some initial insights about the data. For starters, what data is actually available, how well does it appear to reflect the real-world situation, and does it seem to contain the answers we are seeking? Of course the last part won't truly be known until the data analysis project has been completed, but some favorable indicators at least should be found at this point to justify proceeding with the analysis. In practice, data understanding is going to involve mapping out the general structure of the data (tables, fields, relations, etc.) and trying to find any obvious correlations, redundancies, potential data quality issues (missing values, outliers, unbalanced value distributions, etc.) with simple visualizations, statistical tests and simply looking over the data manually.

The initial insights gained from data understanding may well necessitate a review of the project objectives, or may indeed result in project cancellation if it appears the prerequisites for success simply aren't there. On the other hand, if data understanding suggests that the amount, relevance and quality of data are acceptable for the goals set in project understanding, the process can move on to data preparation.

Data preparation consists of selecting the most interesting parts of the data, and doing some pre-processing in an effort to improve data quality and generally make sure the data is in a suitable format for the modeling phase. Typical quality improvements are things like dealing with missing values and outliers, unifying the spelling of categorical attributes and applying some kind of a normalization scheme to numerical attributes. Features might be converted to some format more easily comprehensible to the planned modeling techniques, and entirely new features might be generated in an effort to prod the modeling to some particular direction. Care must be taken in this phase to not inadvertently bias the data in some way, as all of the aforementioned operations can have unintended consequences.

Modeling is the process of selecting suitable model types and finding the best configurations for them. While this is in a sense the central part of the analysis process, in practice the earlier steps in the process often take much more time. This is because model selection usually needs much less manual work than the preparatory tasks. Of course it is entirely possible that the modeling itself reveals some unforeseen issue that sends us right back to data preparation, if for example an otherwise promising model appears to have a problem with the results of some particular data pre-processing technique that was previously applied.

Evaluation can somewhat overlap with modeling, as discussed before. This is because different models are typically compared based on some performance measure, but it is important to understand where this overlap is acceptable, and where it results with inaccurate performance estimates. Ultimately the goal of evaluation is to get an accurate measure of how well the produced model is likely to perform once deployed. This may again lead to project closure or a return to some earlier phase (even right back to project understanding) if it turns out the performance simply doesn't meet the goals of the project.

Deployment is the final stage of the CRISP-DM process. The model is deemed to be of acceptable quality, so it's time to apply it in practice. Notice that this doesn't necessarily mean a practical deployment of the created model as-is, more development work may still be required to turn the model into a product or system that can be used as part of business operations. Further maintenance work may also be necessary if the model needs updates during operation (a common enough case), and it may indeed be that no practical deployment is even done; the model may simply form the basis of some report of findings made during the modeling process. For an example, it might be that the desired end result is simply the knowledge of what kind of data will need to be collected to enable some future data analysis project years down the line [14].

3.5 Machine learning and the software industry

The application of machine learning to various software development processes is a fairly natural fit. ML and software engineering skill sets overlap to a significant degree, and the software development field constantly deals with various uncertainties at all levels of the process, so there is demand for tools that can deal with such. As a result, ML/AI techniques have been explored for both the practical engineering work (design, programming, testing, etc.) and the broader process management (estimation, planning, etc.) [28] [1].

Currently, it appears most of the practical deployments of ML technology in software development are centered on the practical engineering work, in the form of AI-driven code completion tools and various review and testing aids [29]. More elaborate tools for assisting in higher level design and architecture work are the subject of ongoing research [30] [31] though with these there still seems to be some way to go before they can be adopted in practical software engineering work. For project management and effort estimation, it appears most organizations still rely on some form of expert evaluation, though in academic circles applying ML to effort estimation has been a fairly active topic for decades at this point [2] [3] [1].

The most likely reason for this discrepancy between research and practical deployment is that ultimately ML techniques require data to learn from, and relevant historical project data seems to be fairly scarce in most organizations. As discussed in chapter 2, this issue has restricted the use of classical data-driven effort estimation methods to large, mature organizations and projects capable of producing such data. It seems logical that the same problem makes the deployment of ML-based effort estimation methods unfeasible for most industry organizations. Additionally most software development is currently done in some kind of an agile project framework, where inherently cumbersome planning processes are avoided on purpose [9].

The software industry does appear to have considerable interest in various AI-powered tools, and consequently such tools are actively developed in various commercial and open-source ventures [29]. As such, issues like usability and suitability for agile processes in estimation tools will likely be overcome eventually. The availability of suitable data to learn from will likely be a bigger hurdle. Even when working around the same general topic (like software development), different organizations can produce data in very different format, quantity and quality. This means that any individual ML-based effort estimation tool, even if built/trained as a universal/fixed model, is unlikely to fit the organization's data and goals as-is. Thus the deployment of any ML-based tool for effort estimation will probably require a great deal of expert involvement for the foreseeable future. Said expert involvement will likely consist of various data pre-processing and model configuration tasks, or even building an entirely custom machine learning model for the organization, in case available tools aren't deemed suitable even as a starting point for customization.

Finally, while the software industry is in a unique position with regards to the adoption of machine learning and other AI technologies in their work processes, a lot of the progress in the field is driven by the needs of other industries and organizations. In some cases ML solutions originally developed for some particular industry or organization may have considerable cross-industry appeal. For example, a study has been conducted on predicting software developer turnover from monthly reported work hours data [32]. Something like this could be applicable and of great interest for companies in other industries as well.

4 Case ATR Soft / ATR Works data

ATR Soft is a mid-size software company that does both consulting work and own product development. Both kinds of work are generally organized as projects of varying types, ranging from small internal product development needs that may last only a day or two, to larger customer projects that may last several years. Due to the great variance in project scopes, durations and assorted customer needs and preferences, project planning, estimation and tracking processes also vary somewhat. There is however a general desire to improve project effort estimation capability both in the planning stages and later on in the project life cycle.

Improved effort estimations at all points of the project life cycle are expected to produce at least the following benefits:

- **Work allocation efficiency.** While resource allocation isn't done solely based on project estimation (employee preferences and opinions are given considerable weight), allocation can still be made with greater ease and efficiency when good quality estimations are available. This in turn can improve the utilization and productivity of employee time.

- **Improved customer satisfaction.** Better initial estimations mean that projects are less likely to exceed their schedules and budgets. Adjustments to schedule or budget due to scope changes and such are also easier to accommodate for both the customer and the provider.

- **Improved sales.** It is possible that an available project isn't taken because management is not certain the resources needed are actually available at the desired time. This can result in essentially money being left on the table. With improved effort estimation, such situations should be less common.

- **Improved employee satisfaction.** With a better idea of when a project is actually going to wind down, management can minimize the likelihood and degree to which different projects overlap. Such overlaps often cause various high-stress situations in the daily working environment, so keeping them to a minimum will improve employee well-being.

While the tools and processes for general project management can vary between projects, all work hours are logged into ATR Works, an internally developed hours tracking application. This hours tracking data is the subject of the practical work of this thesis.

The practical data analysis project will follow the CRISP-DM process (see section 3.4). All the practical work (plotting, pre-processing, modeling, evaluation) will be done with Python, using the Anaconda distribution. The Python version used is 3.8.3 and the Anaconda Individual Edition version is 2020.07 [33].

Notes about handling potentially sensitive data

Per discussion and agreement with ATR Soft management, data displayed in this chapter will be scrubbed of potentially sensitive information, like employee and customer names, system names and such. Things like project and task names will be replaced with generic identifiers (`"project 1"`, `"task 1"`, etc.) if the original names are considered sensitive. Employee names are always replaced with generic identifiers, and work hours data not related to software development projects is excluded from processing entirely.

Note that generic identifiers aren't consistent between plots, ie. if two plots refer to `"project 1"`, the original underlying project isn't necessarily the same in both. In cases where such consistency exists and is relevant to analysis, it will be specifically mentioned. For more generic task names ("implementation", "testing", etc.) the values will be shown as-is.

These measures are taken to ensure that employees and customers cannot be identified from the data presented here.

The data processing/analysis we intend to do was deemed to be in line with the stated purpose the data was originally collected for (project planning, estimation and tracking).

4.1 Project understanding

While the company has the aforementioned broad goals for effort estimation improvement, the initial primary goal of this analysis project is limited to improving project resourcing by predicting when a given employee's workload is going to fall sufficiently low that they can start work on a new project. In practice this means predicting how a given employee's work hours logged to his active projects are likely to develop.

There is some reporting functionality that uses the data already, but an in-depth machine learning analysis of this nature hasn't been previously performed. As such, a lot of the work we do in this project will be exploratory in nature, as current understanding of the patterns in the data is fairly limited. At a minimum, we will seek to form suggestions on how the hours logging process and gathered data might be improved to support future data analysis projects. Any additional insights encountered during the analysis process will be noted as well.

4.2 Data understanding

ATR Works contains the logged work hours of all company employees from mid-2009 onwards. In addition to the work hours themselves, there is some related metadata like basic information about projects, employees and customers that is needed to give structure to the work hour logs. The system has seen some development and expansion during use, so older data may be incomplete in some respects, because fields that have been added later have only been filled for older projects in case there was a specific need for it. This however is mostly an issue with the metadata, the actual work hour data model appears to have stayed remarkably consistent.

For the purposes of this analysis, we are primarily interested in the hours data and the metadata immediately related to it. This means the projects and tasks that the hours data is structured around. A short description of each entity and their interrelations follows. This is generally restricted to fields and features that we consider interesting from the data analysis point of view:

- **Booking:** A single instance of logged work hours. Consists of employee id, date of the booking, hours spent, references to the project and task this booking belongs to, a short free text description of the time spent, as well as various system fields (created and modified timestamps, etc). Project reference is always present, but task reference isn't mandatory, as some projects don't have any tasks defined for them.

  Note that employees can and frequently will make several bookings per day for time spent on different projects or tasks, and the booking isn't necessarily created on the same day as the work was done. In general, employees have fairly significant latitude to log their hours as and when they prefer, although customers may have some requirements in this regard.

- **Project:** Basic information about any given project, like project name, description and responsible manager. From the analysis point of view, we are mainly interested in the fields that categorize projects (status, type, category) as well as fields that contain information about project planning (planned start and end dates, planned work hours).

The planning fields have been added later on, and thus old projects generally have null values in these. The current process requires them to be always filled, though case-by-case exceptions may still exist. The practical usage cut-off for these fields appears to be around the start of 2015, data older than that will seldom have the fields filled. Project category also seems to be a more recent addition that sees more frequent use from early 2016 onwards.

- **Task:** A project may have tasks defined to further categorize related work hours. Notice that it is not mandatory to create tasks for a project, and the broader semantics of creating tasks are left entirely up to a given project's responsible manager. The system does contain some default tasks that can be used if the manager wishes to, these essentially represent the traditional understanding of project life cycle phases and common activities, like meetings, design, development, support and such, but these kinds of task semantics aren't forced. Another common way to define tasks is to make each task represent a software feature, like some particular system integration or UI view for example.

  In addition to the basic name, description and project reference fields, tasks have a wide variety of fields that can be used for planning, like defined deadlines, planned and remaining hours and such. However, these appear to be very sparsely used, likely because their use isn't mandated by the system or process. Due to this sparsity, the planning fields don't appear initially very useful for analysis.

The data is stored in a relational database, and each of the aforementioned entities occupies a single table. In addition to the relations described explicitly above, things like different project statuses and categories are described in their own tables, but from analysis point of view, those essentially form categorical attributes so the smaller tables aren't separately described here.

4.2.1 Employee work hour plots

The most obvious first step in data understanding is to make some basic visual plots of the relevant interesting data, and see what turns up. As we're interested in predicting individual employee workloads, we plotted the February 2021 work hours of all employees. As mentioned earlier, only work hours data related to actual software development projects is included. Each color-coded plot represents the hours the employee logged for a given project. Some examples from this set are shown in figure 4.1.

It turns out there are essentially two archetypes to how employee hours are distributed. Either they have a very limited number of projects (1-2) and very steadily spend most or all of their time on those, or they have significantly more projects (around 10 or more), and hours are distributed between those with high variance and little apparent predictability.

This is problematic for our planned analysis. In the first case, the prediction ML systems are likely to come up with is simply that the workload will continue as is. For the latter, it is unlikely any worthwhile prediction is achievable, as the variance between days is too great. The only potential avenue for worthwhile individual predictions would be via analyzing the free-form booking descriptions, but that kind of text comprehension is complicated and likely out of scope for this project. In summary, the individual work hours data doesn't appear likely to yield useful trend predictions.

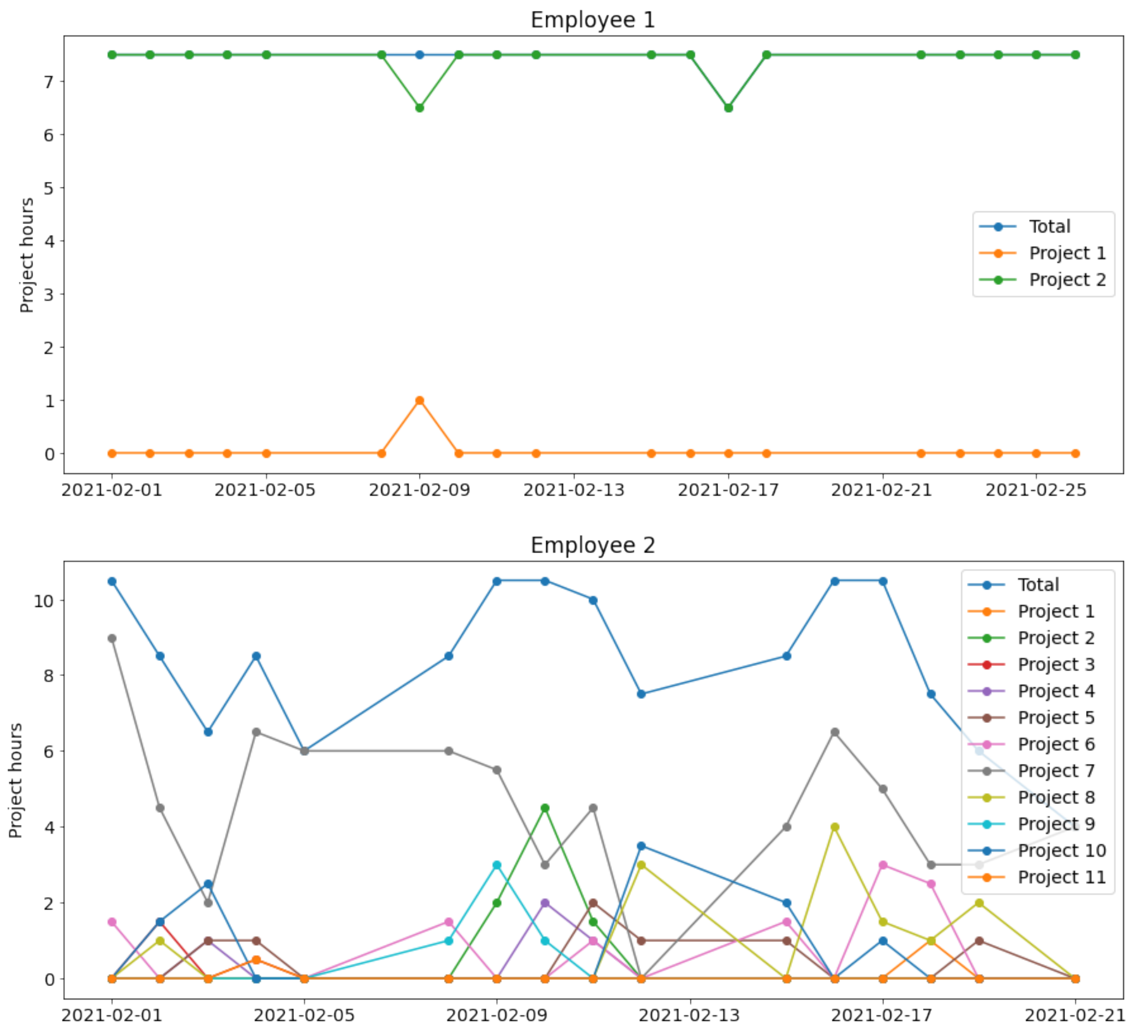

Figure 4.1: Total and project-specific work hours logged by two employees in February 2021. Employee 1 has a very steady workload clearly focused on working full-time on a single project. Employee 2 on the other hand has several ongoing projects that take greatly varying amounts of time each day.

It is worth noting that the total hours plot may be of interest to management and HR as a general employee well-being metric. Great variance in that may indicate wildly fluctuating project demands/workloads, which itself can be a problem depending on how well an individual copes with such circumstances.

4.2.2 Project work hour plots

Another way to approach individual workload estimation is to see how each project as a whole is proceeding, and then inferring employee workload from any trends seen on project-level in the projects that they are working on. With this in mind, some projects' collective hours per day were plotted. See figure 4.2.

There is still a significant amount of variance, but at least some upward/downward trends can be seen. Aggregating the data of multiple employees appears to have somewhat evened out the problematic variance seen in individual employees' bookings. There is still the question of how good of a general model can be derived from the data, given that projects like the ones shown in figure 4.2 aren't the most common type in terms of size and duration. However, this shows at least some promise.

Based on this, we will move forward with project-level prediction. This necessitates some discussion with ATR management, as the initial findings need to be communicated and corresponding analysis goal adjustments agreed upon.

4.3 Project understanding revisited

The first round of data understanding doesn't appear to support the feasibility of predicting an individual employee's work hours to a useful degree. However, a project-level view appears more promising. These findings were discussed with ATR management, and the goal of this analysis was accordingly adjusted to estimating

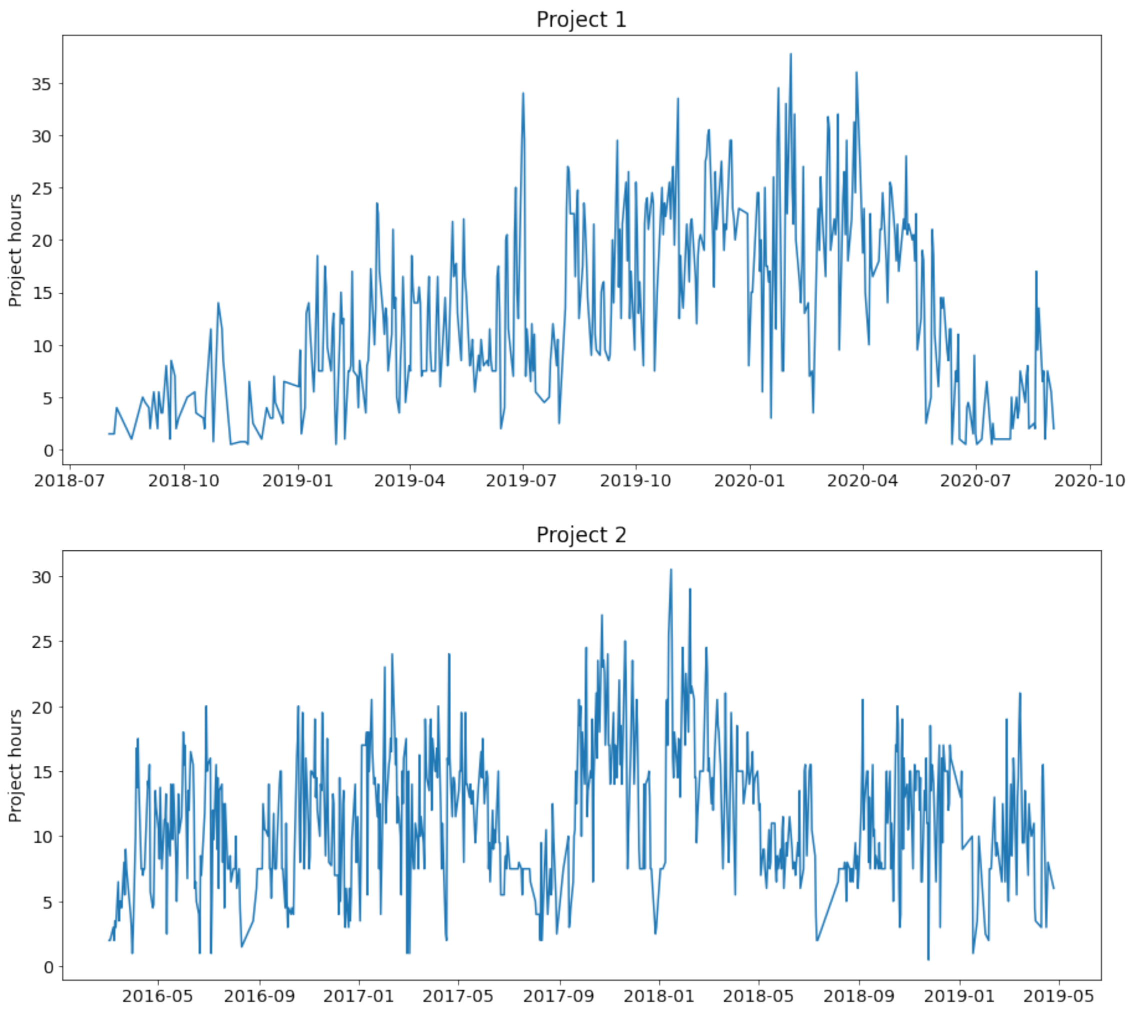

Figure 4.2: All logged hours from the whole duration of two projects. While there is significant day to day variance, some general trends can be seen.

remaining project durations. While this changes the focus of the data analysis project somewhat, the initial goal can still be covered in an indirect way, and project-level estimations are considered useful in and of themselves.

Thus, the new primary goal of this analysis is to predict how many work hours are needed to complete a given project. This prediction should be obtainable at any time during the project life cycle. So we seek to answer the following question: "On date X, how many work hours are needed to complete project Y?" Any additional insights encountered along the way (during data understanding and preparation) are still welcome, and once analysis related to the primary goal has been completed, additional unsupervised learning / pattern finding may be conducted if time permits.

4.4 Data understanding, second iteration

With the focus of the analysis moved to the project level, a closer look at project tasks is merited. Tasks are used to give project hours some more structure and context, and while there is some variety in how the tasks are defined and used, our domain knowledge (previous experience in personally using ATR Works in this case) suggests the task data may be helpful in determining which part of its life cycle a project is in.

To see if such assumptions are warranted, we will plot daily task breakdowns of some larger scale projects in the system. Larger scale here means at least 3-4 people involved and the project lasting at least a few months. While discussions with ATR management have made it known that projects of such scale are somewhat uncommon (most are smaller), they are the most likely to show the suspected task/life cycle correlation. The potential representation issue is ignored for now, though it will need to be addressed later on if analysis proceeds along these lines. Figures 4.3, 4.4 and 4.5 contain the daily task breakdowns of three different projects. Each vertical bar represents one day, with said day's task-specific hours represented by

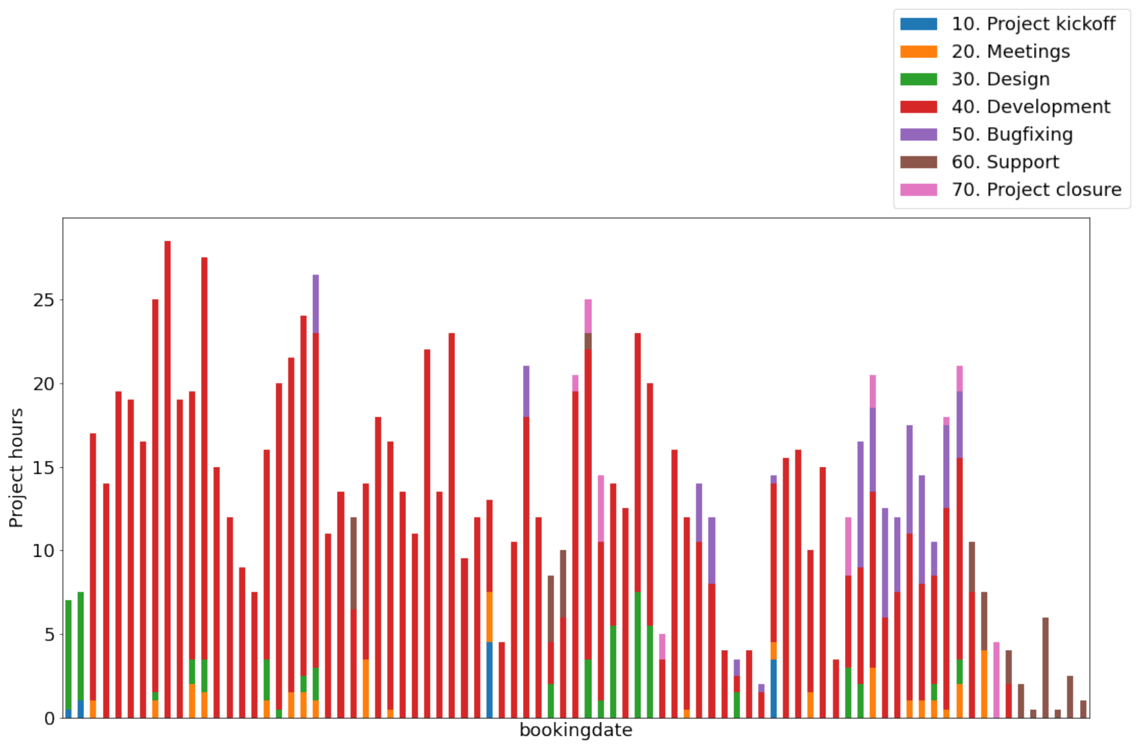

Figure 4.3: In this project the approaching end of its life cycle is clearly represented in the relative prevalence of tasks, with bug fixing and support appearing late in the process. The exact meaning of the project closure task is somewhat unclear, because the first bookings under that task appear midway through the project already.

different colored sections of the bar.

Figure 4.3 appears to support the hypothesis that certain tasks are more prevalent towards the end of project life cycle. In this case, bug fixing and support appear late in the process, and support especially forms the entirety of the final bookings.

Figure 4.4 is less obvious, since the task definitions themselves don't as clearly represent project life cycle phases, but the project management task does get more prevalent as the overall project hours trend down. One potential data quality issue can be seen in this figure: Project management is included in two tasks. In this case, it exists both as a separate task, and as a part of a task that contains other activities as well. In this case it seems like the mixed task was more focused on

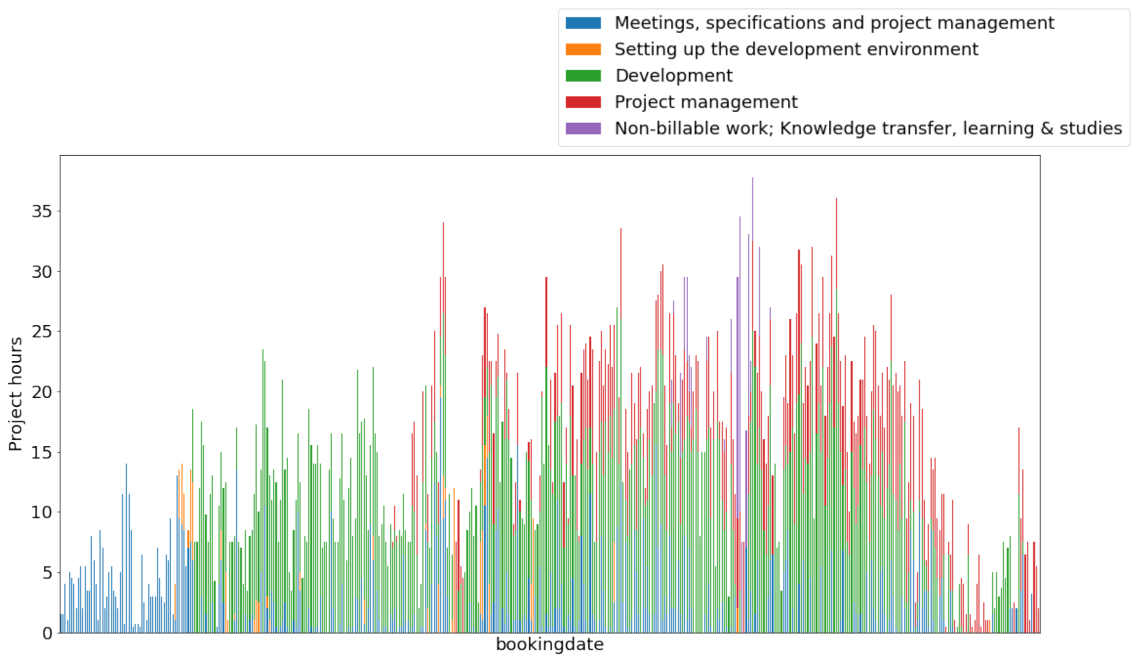

Figure 4.4: Here the change in task prevalence is less obvious than in figure 4.3, but project management does appear to increase in prevalence as the development winds down towards the end. Presumably typical project finalizing tasks were recorded under the project management task, or those tasks were handled by the customer, with the development team's participation in them being supportive in nature. Note that task definitions in this project contain some overlap, with project management present in two tasks.

the specifications and meetings part, since this task is mostly seen early on, with lower prevalence later on when the dedicated project management task starts to appear regularly. However, such overlap in tasks may be a problem if tasks are to be categorized along some life cycle phase pattern.

Figure 4.5 again has test-related tasks appearing towards the end. This project is also notable for having several feature-type tasks defined. While these don't provide any immediately obvious information about project life cycle phase progress, they theoretically could. Features typically have some kind of a prioritization or schedule, so if that data was available, it could assist in project duration prediction. Once

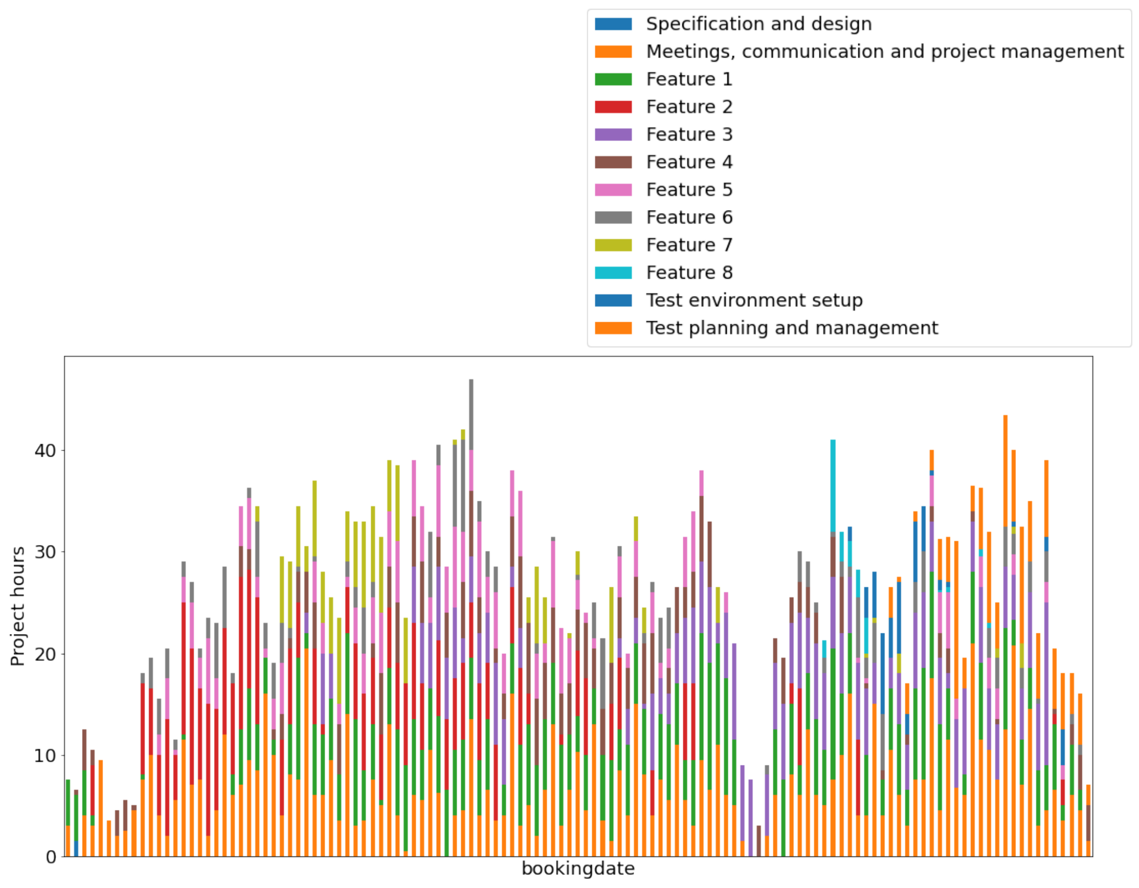

Figure 4.5: This project has several tasks that represent features rather than more generic project activities. However, test environment setup as well as test planning and management do appear late in the life cycle again.

low-priority or late-scheduled tasks start to appear in bookings, the project is likely getting close to the end. ATR Works doesn't currently appear to support storing such data, however.

In summary, the task breakdown appears to offer a way to track project progress. Together with the insight gained in the initial data understanding phase, this would suggest that it is worthwhile to proceed with the data analysis along these lines.

4.5 Data preparation

With the determination that our general analysis direction appears promising, we move on to selecting and pre-processing the data.

4.5.1 Project data

Projects in ATR Works are split into two main types: General and assigned. General projects are visible to all employees, and are intended for tallying up hours used on tasks like team/company meetings, generic office work (like filling out the time sheet), business travel and assorted absences (vacations, midweek holidays, etc.). Assigned projects are visible only to employees who have been assigned to work on them, this contains essentially all productive software development work. Projects are further divided into five general types, which are project, product, business development, continuous service and product project. Of these, discussion with ATR management suggested that business development should be left outside the analysis, because projects of that type don't contain any software development, rather they're about general process improvements and such, which likely won't be relevant to our analysis goals. There are some reservations about the relevance of products and continuous services as well, but those don't appear sufficient to leave that data out at this point.

Based on this and the previously found insight that project planning fields start seeing wider use around at around 2015, we fetched all "assigned" -type projects created since 1.1.2015, whose category is anything other than business development. These criteria yielded 615 projects in total. The basic project data was enriched by calculating the actual hours spent on each project, as well as the actual start and end dates (dates on which the first/last booking was made to that project). Notice that as projects weren't filtered by current status, this data set does contain currently active projects where end date and actual hours are still subject to change.

This will need to be kept in mind later on when actual modeling is done.

4.5.2 Project task data

All the defined tasks related to the aforementioned 615 projects were fetched next. There were a total of 1469 of them. As previously mentioned, the exact semantics of tasks are up to the people who created them, and while a default template for life cycle -type tasks exists, it's not mandatory to use. For this reason, the tasks aren't very useful as-is. Thus we decided to manually categorize them into a more formal representation of a typical project life cycle. For this, the following 7 categories were defined:

- **Unassigned:** Default value for when no category has been defined. Tasks aren't mandatory to use, so some bookings simply won't have any task definition, and thus no task category either.
- **Other:** Tasks which aren't obviously tied to any particular phase of the project life cycle. Meetings etc.
- **Feature:** As mentioned previously, in some projects the tasks represent features of the product or service being worked on. While these likely do have some scheduling or prioritization attached, that data doesn't exist in ATR Works. As such, their place in the life cycle can't be determined, but these tasks form a clear category of their own, so they aren't put in "other".
- **Planning:** Tasks involving design, project planning, kick-offs and other tasks that likely happen early on in a project.
- **Implementation:** Tasks that are likely to happen in the main development phase of a project.

- **Testing:** Testing-related tasks. While some testing is done as part of development in all phases of the project, a greater prevalence of testing indicates a project has likely reached some kind of a validation phase, and is thus approaching end.
- **Finalization:** Final polish, bug fixing, production installations or other delivery activities, product support. Tasks associated with project wrap-up.

All 1469 relevant tasks were assigned one of these categories. In the future, it might be useful to look into techniques that could automatically assign categories to tasks, but in this case the number of tasks was sufficiently limited that it could be done manually in a reasonable time. Additionally, to avoid such issues in future analyses altogether, a task category field could be added to the existing data model and made mandatory, so that life cycle classification would always be done when tasks are created. This might allow us to derive some project phase data from feature-based tasks as well, since those could be given category values according to schedule/priority.

4.5.3 Project bookings

All bookings related to the selected 615 projects were fetched next. This amounted to 65083 records. As we're planning to calculate task category breakdowns for various time spans from these bookings, we added the task category of each booking directly to the booking records at this point, so that information won't have to be cross-referenced from task data. Approximately 22000 bookings had a null task id (ie. no task defined). These were more common in older data. As mentioned in previous section, these received the task category value "unassigned".

Each booking contains a short free text description of what was done. More in-depth text analysis was deemed out of scope here, but some simple keyword matching was still performed, to obtain some insight from the description text. This was prompted by discussion with ATR management, where it was mentioned that one way to guess whether a project was about to wrap up was to check at a glance whether things like testing and bug fixing seemed to appear commonly in descriptions. The keywords were "test", "bug", "deploy" and "refactor/refaktor". The refactoring match was done with two spellings to include descriptions written in Finnish. Something similar was considered for "deploy", but a commonly used mostly unambiguous Finnish equivalent wasn't found. Even descriptions written in Finnish seemed to still commonly use some variant of the word "deploy", so that was deemed good enough.

Again, the logic is that such words are more likely to appear in descriptions towards the end of a project, when testing, bug fixing and code cleanup activities are more commonly performed. For each keyword, a new attribute was created, which contains 0 if no matches to that keyword were found, and 1 if at least one instance of the keyword was present in booking description. The matching was done as a simple substring match (ignoring case), so the keywords don't have to appear as-is, they might also be part of a compound word. This means that some of these matches likely are inaccurate, but manual inspection of some data suggests that such cases are fairly rare.

4.5.4 Aggregation and feature extraction

With the basic work hour data and the immediately relevant metadata categorized and enriched with some additional attributes, the remaining task is to aggregate this data into a coherent matrix where each row represents the state of a given project at some point in time. Various new attributes will also be generated from the enriched raw data.

Note that while we of course seek to generate attributes that we believe will prove useful for modeling, ultimately the usefulness (or lack thereof) of a given attribute can only really be known once it is used and evaluated in modeling. With this understanding, we aren't going to do any feature selection at this point; attributes generated here are assumed to have at least some value, based on domain knowledge obtained from conversations with ATR management and personal experience. Depending on what kind of performance we achieve in the modeling phase, we may attempt some feature selection there.

Each row in the aggregated data represents project bookings in the last 7 days, with some longer term historical data for comparison. There are 58 attributes in total, divided into the following three categories (number of attributes in each category in parentheses):

- **Tracking attributes (12)** are intended to assist with partitioning the data into subsets for training/testing, as well as to make it clear what project and time span a row represents. These are not initially intended to be used as training/prediction features, but may of course be used for that purpose as well if the need arises.
- **Training/prediction features (41)** represent the project's state over the time span represented by the row, as well as longer term historical data from the last 4 and 12 weeks.
- **Target attributes (5)** are the values that we seek to predict in modeling.

Notice that as mentioned earlier, some of the projects in this data set are in fact active and ongoing. These can't be used for model training or evaluation, because the final values of their target attributes aren't known at the moment. However, they are still included here because ATR Soft might be interested in what kind of durations are predicted for them.

Tables A.1 - A.4 in appendix A describe all attributes in the final aggregated data set.

4.5.5 Final notes about aggregated data

During aggregation, projects with 0 logged hours, as well as projects where either actual start and end dates or planned start and end dates were equal (ie. project's planned or actual duration was just one day) were filtered out, as it was deemed these aren't likely to contain any interesting data about general project trends. With this, the number of projects was reduced from the initial 615 to 537. The total number of aggregated booking data rows was 16370.

Of the 537 projects, approximately 450 were inactive and thus potentially suitable for use as training/evaluation data. This set was further pruned to approximately 400 projects, based on discussions with ATR management about the characteristics and circumstances that make the removed projects unlikely to represent the general trend well. This leaves us with approximately 8400 relevant rows of aggregated project data to be used for training/evaluation.

There are some known potential data quality issues in the aggregated data. The most important one is, that the project-level planned hours and end date fields, which are used to derive some of the features in the aggregated data may not be entirely reliable. According to discussions with ATR management, these fields are sometimes updated during a project, to reflect updated estimations or requirements. As such, especially for time spans early on in the project, features dependent on this information may not accurately represent what the project estimation was at the time. This may give machine learning models an overly optimistic view of the planning field accuracy. This might then cause those attributes to get more weight based on faulty information.

ATR Works does store some history information about project fields, so digging out the actual estimate value that existed during a given time span of a project is possible, but it was deemed overly complicated to do given current schedule constraints. This might be revisited if the modeling results and overall progress require/permit it.

It should also be noted that the feature values haven't been normalized in any way yet. This is because such normalization is relatively trivial, requiring only a single library function call, so doing it during modeling as and when necessary doesn't require much extra effort.

The most important thing to keep in mind is that we're dealing with time-series data here. Care must be taken to ensure that models aren't trained on data that wouldn't exist at the point of time we're simulating with testing data. This is the main reason for the existence of the aforementioned tracking attributes. Between the date and project status information contained in those, it should be relatively simple to separate the data into training and test sets without biasing the results with unrealistic training data.

5 Model training and evaluation

With the data pre-processed into a form suitable for modeling, we can proceed with fitting some actual machine learning models and seeing what we can predict. In addition to model selection and tuning, we may also need to do some more data/feature selection in a similar fashion to what was done in pre-processing. This depends on what kind of initial insights we gain from modeling, but as mentioned in section 3.4, this kind of backtracking is common in data analysis processes.

General considerations regarding time-series data

As we're dealing with time-series data, we must make sure in each experiment that we're not training our models on data that wouldn't realistically be available at the point of time where we're trying to predict something. As the actual project end date is included on every row of the aggregated booking data, we can easily address this concern by designating a cut-off date between the training and test data sets, and ensuring only bookings related to projects that ended prior to this date are used for training. Conversely, aggregated booking rows whose time span starts after the cut-off date can be used for testing.

Note that testing rows need to be specifically filtered by time span start date, rather than project end date. Some of these projects were ongoing before the cut-off date, so some aggregated booking rows of these projects depict times before cut-off, which means some training data wouldn't realistically have been available at the time.

As mentioned in the previous chapter, the earliest data we have is from 1.1.2015. The closer the cut-off date is to this date, the less data we have for training. Conversely, the closer the cut-off date is to the present day, the less data we have for testing. In the former case, the problem is obvious; too small amount of training data means the model will likely underfit to a significant degree, and thus predict poorly. In the latter case, the problem is the accuracy of any performance evaluation we do, as the amount of testing data is such a small fraction of the whole data set that we can't be certain it properly represents the set as a whole.

As our data set is not particularly large, cross-validation would be very useful, but unfortunately it cannot be applied here due to the previously mentioned timeline constraints.

For the initial experiments, we'll use the cut-off date of 1.4.2020. This was determined by simply checking how the aggregated data splits on different dates, with the aforementioned date yielding roughly a 3-to-1 split between training and testing data sets in terms of both projects and aggregated booking rows.

The need to split the data along a cut-off date has some implications regarding the representativeness of the test set. Namely, when using the aforementioned cut-off date, our test set only represents projects that lasted a maximum of approximately one year, or a maximum of a year's worth of data from the tail end of projects that lasted more than a year. While moving the cut-off date further to the left on the timeline (ie. further into the past) would increase the maximum project run times that can be represented in the test set, this would also commensurately shrink the training data set. As we're dealing with a fairly limited amount of data to begin with, this is undesirable, and furthermore, a large number of ATR Soft's projects are shorter than a year, so from a practical analysis perspective the representativeness of the test set is good enough.

Tools and related considerations

We're using the scikit-learn (sklearn for short) [34] library for the practical modeling and pre-processing tools. These require that there are no NaN (Not a Number, ie. null) values in the data, even when using methods like decision trees that could theoretically handle such a situation. NaN values are present in the four planned duration features (planned days left absolute and percentage, as well as planned hours left absolute and percentage). These are likely to be quite valuable prediction features, so dropping the whole columns isn't an option, and there isn't an obvious way to impute values either. As such, rows with NaN values were dropped. This reduced the available data to 6009 aggregated booking rows representing 290 projects.

General performance baselines

For performance evaluation, it is useful to have some baselines to compare the achieved performance against. Occasionally performance scores that look promising turn out to barely exceed some trivial baseline, meaning that the model isn't ultimately useful.

For classification, we will use a simple majority voter as baseline. In practice this means always predicting the value "12w" for the "ends_in" column, which achieves a classification accuracy of 35% with a test set created from the aforementioned 1.4.2020 cut-off date.

For regression, we have two suitable baselines: Always predicting the median value of the label column, and predicting whatever the current expert estimate is at the time the prediction is made. Table 5.1 contains basic performance measures for the median baseline, table 5.2 contains the same measures for expert estimate.

Measurement	Attribute	Result
C-index	Hours left (abs)	0.5
C-index	Days left (abs)	0.5
C-index	Hours left (pct)	0.5
C-index	Days left (pct)	0.5
MAE	Hours left (abs)	60.4
MAE	Days left (abs)	50.4
MAE	Hours left (pct)	0.251
MAE	Days left (pct)	0.271

Table 5.1: C-index and mean absolute error scores for median baseline.

Measurement	Attribute	Result
C-index	Hours left (abs)	0.720
C-index	Days left (abs)	0.719
C-index	Hours left (pct)	0.771
C-index	Days left (pct)	0.769
MAE	Hours left (abs)	61.4
MAE	Days left (abs)	61.6
MAE	Hours left (pct)	0.458
MAE	Days left (pct)	0.388

Table 5.2: C-index and mean absolute error scores for expert evaluation baseline.

5.1 General model

The best case scenario for ATR Soft would be if we could create a general model capable of accurately predicting the duration of any project at an arbitrary point in time from all accumulated past project data. This is a very ambitious goal given the great variance in scope and circumstances that ATR's software projects can have, but it is fairly simple to set up an experiment to see how well such a model might perform.

5.1.1 Experiment setup

- **Data:** Data is split into training and testing sets along a cut-off date of 1.4.2020. Data from projects that ended before that date forms the training set, and data from after that date forms the testing set. The feature columns are z-score standardized to ensure they all have comparable magnitudes. No feature or data selection is done beyond this, so both data sets contain the full variety of project scopes and categories, and the full feature set is used.

- **Models:** Three different model classes will be tested. K-nearest neighbors, random forest and multi-layer perceptron. For each, some model selection will be done, the particulars of which depend on the model in question.

- **Goals:** We will attempt to predict the values for all of the five target attributes listed in A.4. Four of these are numerical, and require a regression model, the last is categorical and requires a classification model. The prediction will be made for every aggregated booking row in the test set, to simulate predicting project duration at an arbitrary point in time.

- **Evaluation:** The performance of each model candidate is evaluated by calculating the mean absolute error for each regression target, and by calculating classification accuracy and a confusion matrix for the classification target. Mean absolute error is used instead of mean square error to give a more easily human-readable approximation of how far from the true values the predictions typically are.

  Regression results will be compared to baselines established in tables 5.1 and 5.2. Classification results will be compared to the majority voter score.

Notes

Some of the features in the data set are conceptually somewhat problematic for this kind of a scenario. Features that depict project history over a longer period of time (4 and 12 weeks in this case) will obviously have misleading values when predicting either a project that never lasted that long, or one that hasn't yet run that long at the point of time where the prediction is made. A cursory check shows that out of the approximately 6000 rows in our data set, approximately 2400 depict situations where the project has run less than 12 weeks, and approximately 1000 depict run times of less than 4 weeks.

Prediction of absolute remaining calendar days and work hours is unlikely to attain good accuracy. This is because regression problems generally require the training data set to have example values of roughly similar magnitude as the correct target value in a distribution where the prediction mechanism is likely to arrive at some reasonably close prediction. In this case however, we know the training data contains true hours and days values ranging from the single digits to the thousands. Percentage values for hours and calendar days left may still arrive at a reasonable prediction, because the scale is static (0 ... 1) for all rows in training and test sets.

5.1.2 Model selection

k-nearest neighbors

For k-nearest neighbors, model selection was performed by training a model with various k-values and plotting the results. Distance measure was euclidean, and other parameters were left to the sklearn defaults. The k-values tested were 5, 11, 21, 51, 101 and 201. Odd values were used to reduce the amount of ties (less significant with regression, but potentially useful for classification) C-index and MAE result plots for each k-value can be seen in figure 5.1.

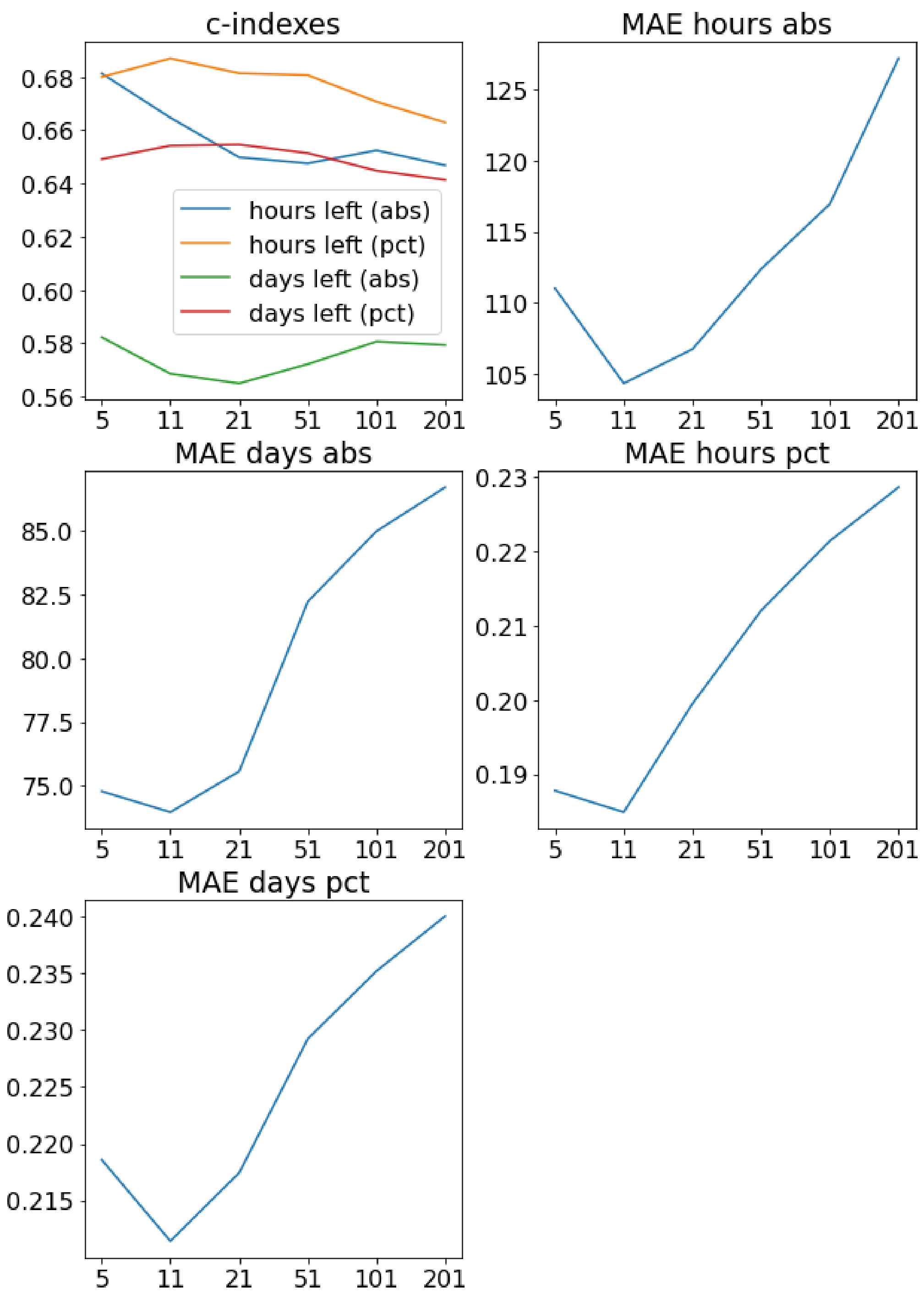

Figure 5.1: Regression accuracy metrics for kNN models trained with various values of k. K-value on X axis, calendar days and work hours as absolute values and percentages on Y axis.

For the classification target attribute (ends_in), a kNN classifier was trained with the same set of k-values and euclidean distance measure. Classification accuracies for each k-value can be seen in table 5.3. For the best k-value (11), the corresponding confusion matrix can be seen in figure 5.2.

k-value	Classification accuracy (%)
5	34.1
11	34.7
21	34.3
51	29.5
101	29.5
201	27.7

Table 5.3: Results of k-NN classifier for predicting the categorical "ends_in" target attribute.

Random forest

Random forest regressors and classifiers were selected mainly by changing the number of trees. Tree depth (values between 5 and 25) and alternate split scoring functions (mean absolute error for regression, entropy for classification) were also tried, but ultimately deemed to have little, mostly detrimental, effect on the results. As such the models depicted in the following results only vary by tree count, other parameters are left to their sklearn defaults. It should be noted that random forests are remarkably stable models, there is very little change in results between different numbers of trees either.

Random state was set to zero for all tests, to ensure comparable results between them (since random forest is inherently a non-deterministic model class).

C-index and MAE result plots for different tree counts can be seen in figure 5.3.

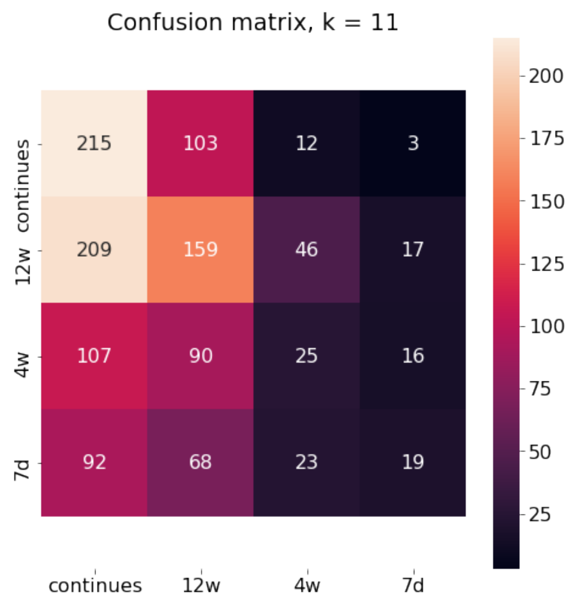

Figure 5.2: Confusion matrix for kNN classifier with k = 11.

For the classification attribute "ends_in", the classification accuracy achieved by each random forest model is listed in table 5.4.

Tree number	Classification accuracy (%)
20	44.8
100	47.0
200	48.1
300	48.2
400	48.1
500	48.2

Table 5.4: Results of random forest classifier for predicting the categorical "ends_in" target attribute.

The 300- and 500-tree models achieved identical classification accuracy, so confusion matrices of both models were inspected to see if there was some notable difference in misclassification types. There wasn't, the matrices were nearly identi-

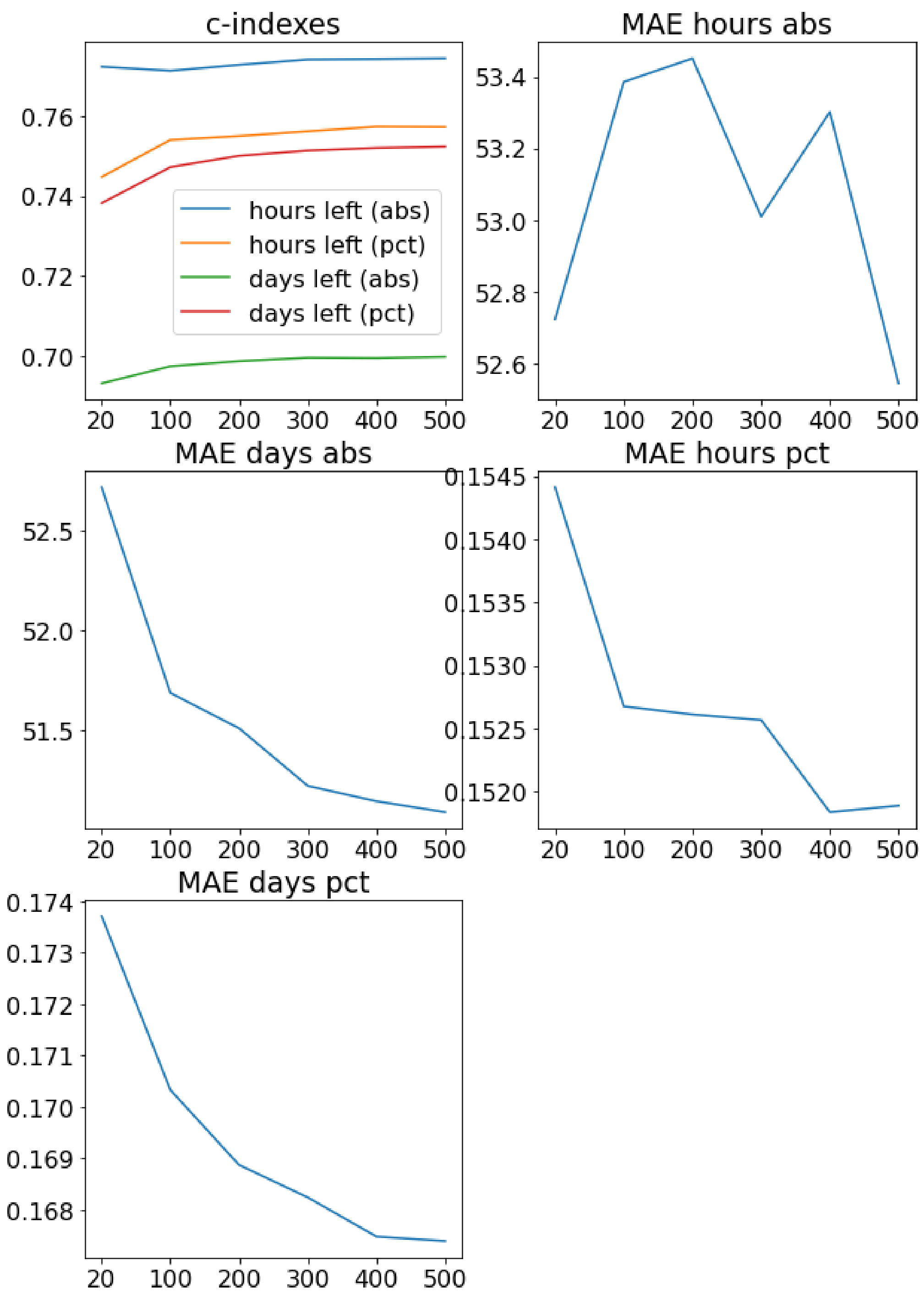

Figure 5.3: Regression accuracy metrics for random forest models trained with various numbers of trees. Tree number on X axis, calendar days and work hours as absolute values and percentages on Y axis.

cal, with cell values differing by 1-3 at most between models. The confusion matrix for the 300-tree model can be seen in figure 5.4.

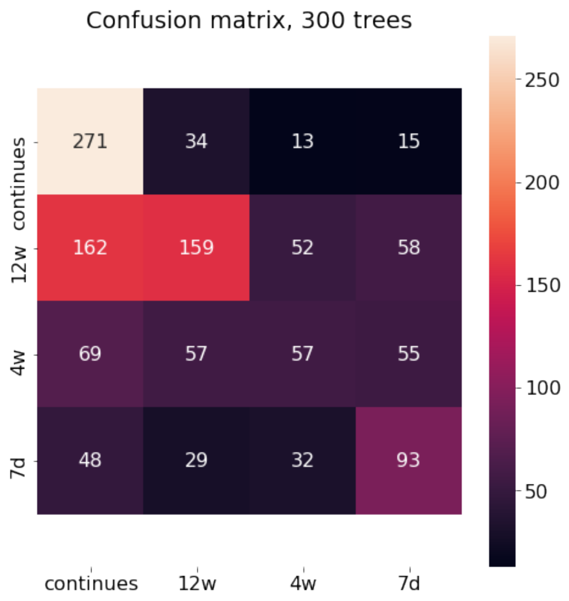

Figure 5.4: Confusion matrix for random forest classifier with 300 trees.

Multi-layer perceptron

For multi-layer perceptron models, the main variable was the size of the hidden layer. For activation function, both rectified linear (ReLU, sklearn default) and hyperbolic tangent were tested, of these two, hyperbolic tangent performed better in most respects. This held for both regression and classification.

For weight optimization solver, the default "adam" (a form of stochastic gradient descent) is recommended for data sets with thousands or more records. Our data set is on the lower end of this, so the "lbfgs" (Limited-memory Broyden Fletcher Goldfarb Shanno) solver intended for small data sets was also tried, but performed worse.

Early stop training was used for all MLP models, with 10% of the training data set aside for validation. Maximum iterations were increased from default 200 to 400 after initial experiments suggested the models had trouble converging in 200

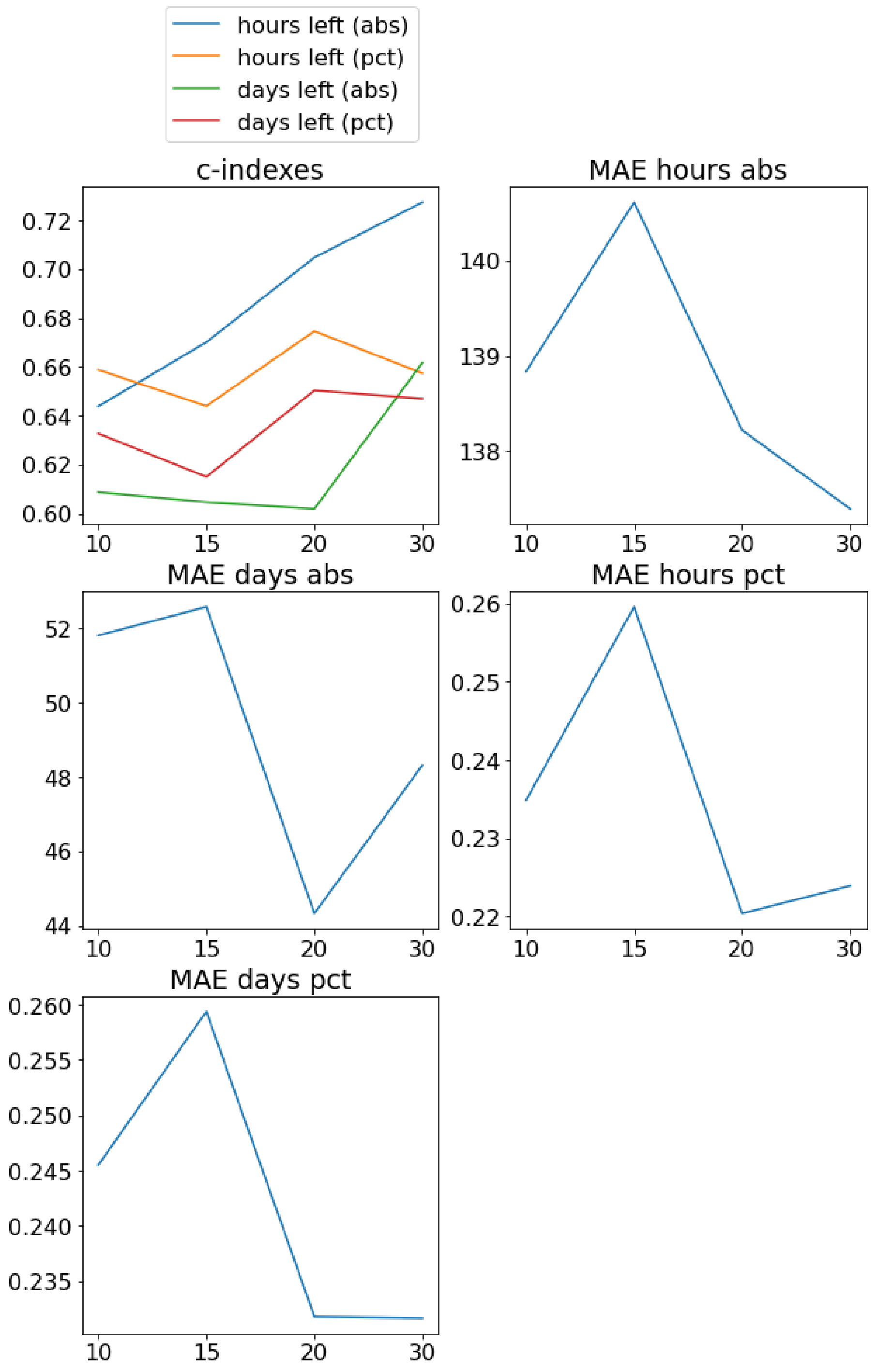

Figure 5.5: Regression accuracy metrics for multi-layer perceptrons with different hidden unit counts. Unit count on X axis, calendar days and work hours as absolute values and percentages on Y axis.

iterations. MLP is not a deterministic model, so random state was again set to zero to make different runs comparable.

In short, the MLP regression and classification results are from models where hidden layer unit count varied, but other model parameters were always the following:

- Activation function: hyperbolic tangent (tanh)
- Solver: adam
- Early stopping: true, with 0.1 validation fraction.
- Maximum iterations: 400
- Random state: 0

Regression results can be seen in figure 5.5. For classification, the accuracy with different hidden layer sizes can be seen in table 5.5.

Hidden layer size	Classification accuracy (%)
10	34.1
15	32.3
20	37.3
30	37.5

Table 5.5: Results of MLP classifier for predicting the categorical "ends_in" target attribute.

The classification accuracies of MLPs with 20 and 30 units in the hidden layer were very close, so again both confusion matrices were inspected to see if any major difference in classification trends could be found. However, again the matrices were similar enough that no particular insight can be drawn from this. The confusion matrix for 30 hidden units can be seen in figure 5.6.

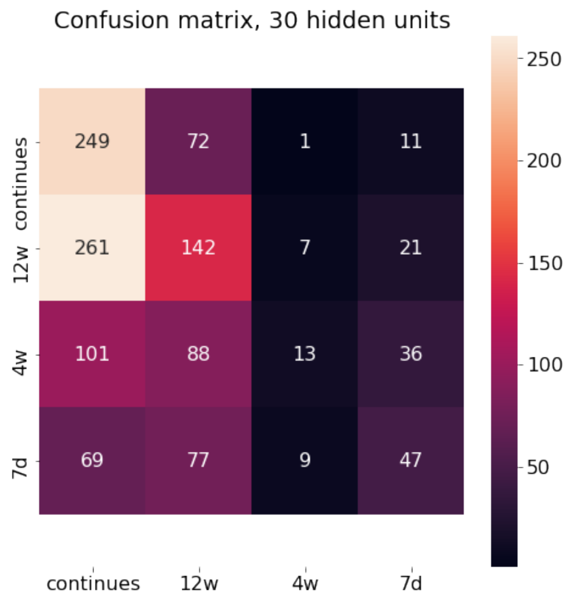

Figure 5.6: Confusion matrix for MLP classifier with 30 hidden units.

5.1.3 Initial results and discussion

Regression

For the regression targets, random forest achieved the best performance. The only exception was the absolute number of days attribute, where all tested multi-layer perceptrons achieved performance comparable to random forest, with the 20 hidden unit model reaching a significantly better result (mean absolute error 44.327) than the best random forest (500-tree model, which achieved 51.090 MAE). With every other regression target attribute, even the worst-performing random forests still outperformed the best models of the other classes.

MLP's performance with predicting the absolute number of calendar days left in a project highlights an important issue with artificial neural networks: The inability to explain their decisions. This model class performed noticeably worse than the others in all other categories measured by mean absolute error, so the performance with this single attribute is quite interesting, but we essentially cannot know why this happened. Furthermore, the c-index results are in the range of 0.60-0.66, which is better than random guessing but still quite poor, which suggests that while the predictions are closer to true values than what other models managed, their internal order is fairly poorly preserved in the predictions.

Classification

In terms of classification, random forest again performed the best, with classification accuracy in the 47-48 percent range for most models. KNN was the worst with accuracy ranging between 27-34 percent, and MLP between these two with an accuracy range of 32-37 percent. A simple majority voter reached an accuracy of approximately 36%, so kNN and MLP can be considered completely useless for classification here. Closer inspection of the confusion matrices drawn for the best model of each class shows that the performance difference mainly comes from the model's ability to correctly predict the less common "4w" and "7d" classes, though in kNN's case its ability to predict the majority class was also considerably weaker than the other types of models' was.

Model usefulness

Overall, random forests are the most promising class of models, but in terms of absolute accuracy measures for both regression and classification, the performance is too poor to likely be of much use in a company's daily operations.

In terms of calendar days, the best models are typically off by almost two months of actual project duration, which is unlikely to be acceptable even in a larger scale project that runs for months or years. In terms of work hours, the best models are off by approximately a week and a half's worth of hours (for a single full-time employee), which would in fact be a very good prediction accuracy for large projects which can run for thousands of hours, but we know a large number of projects in the test set ran for less than 100 hours, so this kind of an error margin is unacceptable.

The insufficient accuracy of absolute hours and days predictions was more or less expected, since as mentioned before, the data set contains projects of vastly different scope. However, the percentage predictions were also quite poor, with mean absolute error being approximately 15% at best for work hours and 17% for calendar days. In order to be useful, these would need to be less than 10%, preferably closer to 5%.

On the other hand, random forests achieved c-index scores ranging from roughly 0.7 for absolute days left to 0.77 for absolute hours left, with percentage attributes evaluated to 0.74-0.75. At first glance this suggests that the order of records is captured quite well, so there might be some potential in trying to predict project completion order, rather than the absolute time an individual project might still need. However, comparison with the expert baseline (table 5.2) shows that this performance is in fact roughly the same or slightly worse than what current expert evaluation achieves. Thus a model would likely have to reach c-index values above 0.8 to be a noticeable improvement.

In terms of mean absolute error, random forest does outperform both baselines in all cases except for the absolute number of days left, where the median baseline result is approximately equal to what random forest achieved. With the percentage-type labels, the random forest models outperformed both baselines to a very large degree (by as much as 30 points in case of hours left percentage, where expert baseline's mean absolute error was 45% versus the model's 15%), but as mentioned before, this is still likely not good enough to consider the model useful in a real-world estimation scenario.

Potential causes for inaccuracy

One potential issue here is that we don't know how valuable each individual feature in our data set actually is. Nearest neighbors method is particularly vulnerable to redundant or counterproductive features, since it cannot internally weight them by usefulness. Random forest on the other can do some feature weighting internally, and it did perform considerably better than kNN.

Additionally, as mentioned in the experiment notes, some features in the set represent lengths of history that don't exist at the point of prediction for many rows, since the projects either haven't run that long at the prediction point, or never lasted that long in the first place.

With these considerations in mind, we'll see if we can improve general model performance by doing some feature selection.

5.1.4 Follow-up: Feature selection

While we can't feasibly try every possible subset of our 41 features, we can pare the feature list down by domain knowledge. In practice, we will do backward feature selection, that is, we'll drop features we suspect to be useless and see whether the model performance improves. Based on domain knowledge, the usefulness of following feature categories will be tested like this:

- Features depicting task categories other than testing and finalization. Depending on how project tasks were defined, the content of these features can be questionable. The categories had to be assigned by hand, and this process consisted of the author's best guess rather than involvement of the actual people who worked in the project, so their accuracy is questionable to begin with. Furthermore, these columns depict project phases that don't indicate the project ending soon, or phases where it can't be said what phase they represent in the first place. This may be unnecessary clutter rather than useful information to base predictions on.

- Features depicting 12-week history (all attributes with the "12w_" prefix. For many projects and prediction points, these are irrelevant or misleading for the aforementioned reasons.

- Keyword match features. Again, it's entirely possible that the simple keyword matches don't actually correlate well with project end. Some of these columns are also quite sparsely populated with any non-default (non-zero) values.

All tested feature subsets can be found in appendix B. For each of these feature subsets, model selection was run as described in 5.1.2.

5.1.5 Results of feature selection

KNN saw the most improvement from feature selection, with the removal of each of the aforementioned subsets resulting in performance increase across all measures. Ultimately, the results still fall somewhat short of what random forest models achieve, but the improvement from non-feature selected data is notable in both regression and classification. The final regression results of kNN with all aforementioned feature categories removed can be seen in figure 5.7. Classification accuracy range was 41-46%, seeing improvement of approximately 10 or more percentage points for each k-value, putting kNN well above majority voter and MLP in this situation.

In other model classes, the feature selection had little to no effect, and in some cases was actually detrimental, though to a very minor degree. The mean absolute error for absolute days left with MLP saw the higher hidden unit counts achieve values under 50 more consistently, but everything else either stayed the same or deteriorated slightly as features were removed. With random forests, classification accuracy improved by two or three percentage points across all models, while regression results stayed the same or deteriorated to a barely noticeable degree.

As random forest models do internal feature selection and can return the feature importances they arrived at, we plotted the feature importances returned by the 500-tree regression and classification forests for comparison. The results can be seen in figures 5.8 and 5.9.

Notice that mean decrease in impurity (MDI) as a metric does favor features

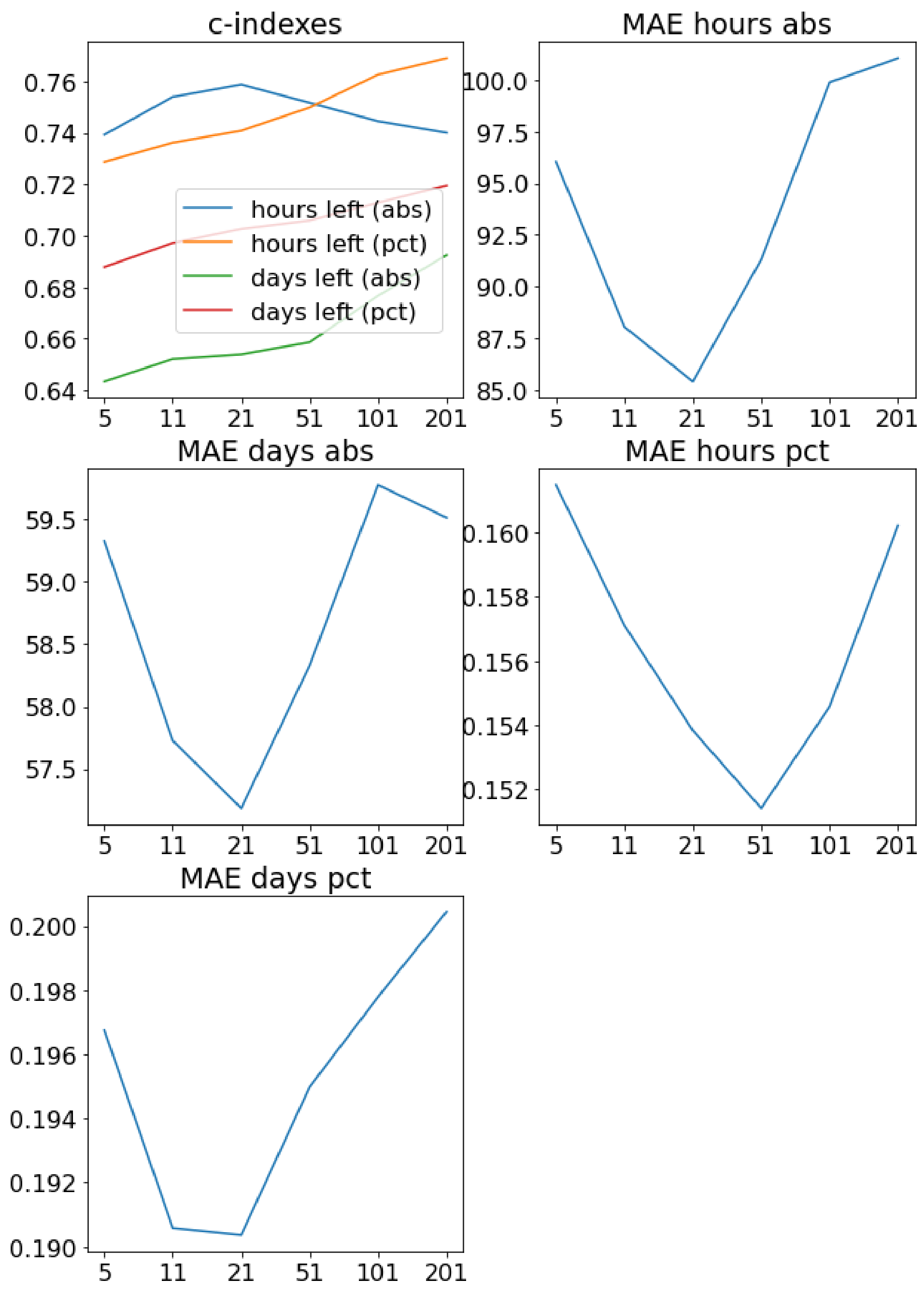

Figure 5.7: Regression accuracy metrics for kNN models trained with various values of k. K-value on X axis, calendar days and work hours as absolute values and percentages on Y axis. Unnecessary features were pruned from the data prior to training and testing.

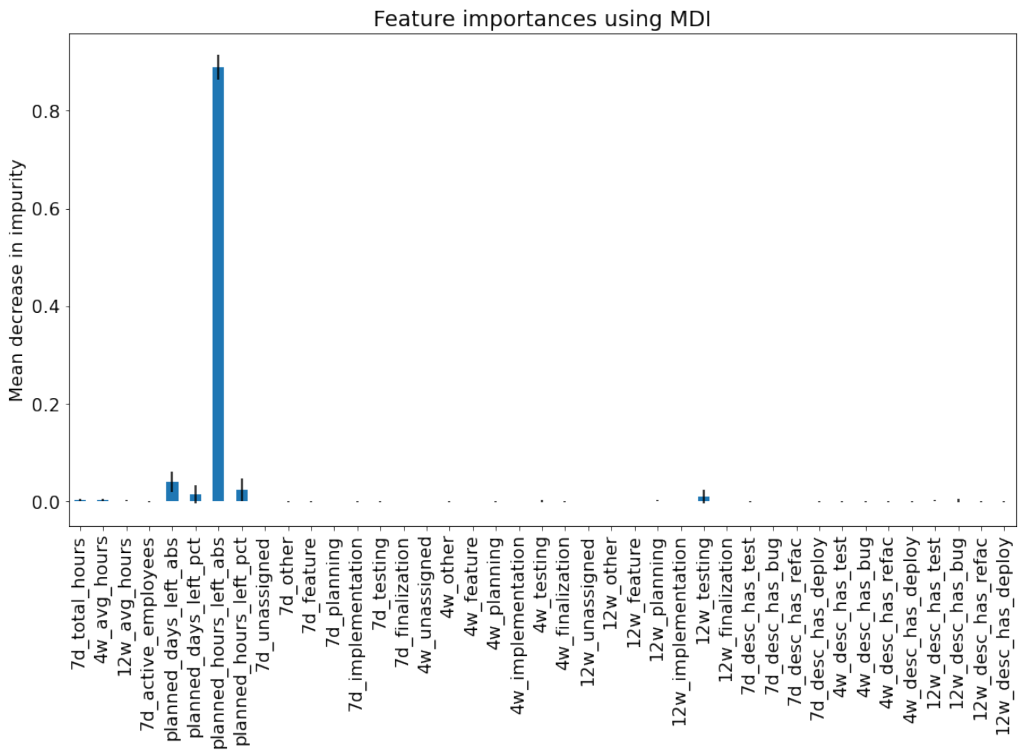

Figure 5.8: Relative feature importances calculated by the 500-tree random forest regressor. The metric for importance is mean decrease in impurity (MDI). The error bars represent one standard deviation.

with high cardinality (large number of different values), and sklearn documentation recommends measuring feature importance by permutation in this case instead. In practice this means randomly shuffling the values of a feature and measuring how much model performance was impacted by this. However, this resulted in near-identical feature importances to what is displayed in figures 5.8 and 5.9, so cardinality bias doesn't appear to be an issue here.

Both feature importance plots suggest that our domain knowledge-based feature selection was on the right track. Especially for regression, only the planned days/hours fields seem to hold any real value, and even of those the absolute planned hours feature is extremely dominant. For classification, the feature importance distribution was more even, though again the planned days/hours fields were the most

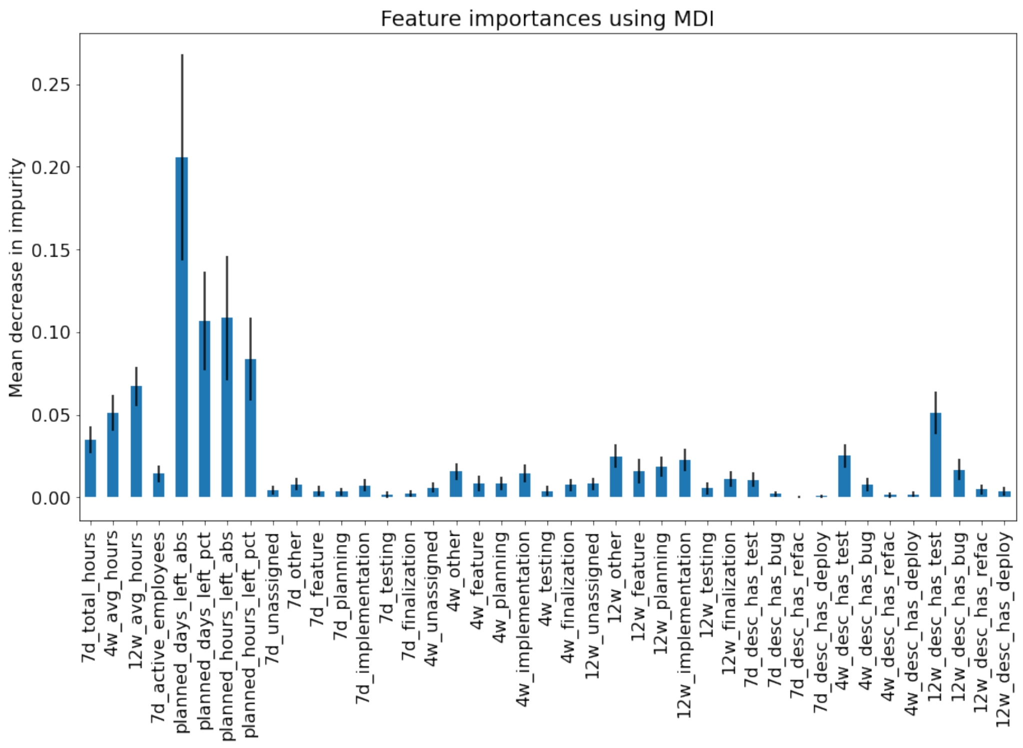

Figure 5.9: Relative feature importances calculated by the 500-tree random forest classifier. The metric for importance is mean decrease in impurity (MDI). The error bars represent one standard deviation.

important.

This suggests that a large quantity of the available features are in fact fairly inconsequential for making predictions, but kNN is the only model that suffers from this to a noticeable degree. In either case, while the performance improvement for kNN was noticeable, it still failed to improve upon what random forest achieved, so random forest remains the most promising model class.

5.2 Limited scope models

As the general models don't appear to offer sufficient performance, we will next try to create models of more limited scope. This means creating models that are built for predicting projects of a particular scope or scale, rather than trying to encompass all of the company's projects in a single model. Furthermore, we will try to address the issue of using features that describe lengths of history that a project may not yet have at the time some predictions are made.

Discussions with ATR management about the results of the general model experiment suggested that while models should of course be as accurate as possible, in some situations there may be a preference for the model's tendency to over- or underestimate the remaining project duration. This is essentially a trade-off between customer satisfaction and the efficiency of personnel assignments to billable work. If a project's remaining duration is underestimated, the company may end up making delivery date promises that cannot be kept, which is problematic for customer relations. On the other hand, if the project duration is overestimated, personnel will potentially be out of billable work for a while as the project completes before a new one can be assigned. While no project management decision will be made solely based on model predictions, with the aforementioned concerns in mind it would be useful to know which way a model is likely to err, so that management can account for that in resource planning.

Furthermore, accurate estimates are most in demand during the last month of a project's run time, as then the issue of re-assigning personnel to new projects is the most acute.

5.2.1 The plan and initial insights

Limiting the scope of a model can be done in several ways, depending on the data we're working with. In this case, an intuitive idea is to plot project durations (in days) against their total work hours, in the hopes of finding some particular clusters of project scope that could be targeted for modeling (the plot itself is omitted by request from ATR). Unfortunately, no particular clusters were revealed by this. However, a number of projects did appear as clear outliers in terms of scope. These will be addressed in experiment setup.

As no clear clusters were found, we will instead attempt to create project-specific models that account for general project scale as well as the elapsed project run time at the point of time where the prediction is made. The fundamental idea is to create 1-3 models for predicting each project at different parts of its run time. The models are trained on data which contains only past projects of similar scale as the target project. Up to three different models per project are trained with different feature sets based on how long the project has been running at the time the prediction is made. The first model represents the situation where a project has run for less than 4 weeks. This model is trained on a feature set where the 4- and 12-week history features are omitted, since the project hasn't got that much history yet. The next model represents a project run time of over 4 but less than 12 weeks. This model is trained on a feature set where the 12-week history features are omitted. The final model is trained on a full feature set including all history features, this represents a project that has run for at least 12 weeks.

The general model experiment showed that on the scale of data we're operating in, models are quite fast to fit, so creating project-specific models is within the available processing resources.

We will use planned project work hours as a scale metric. Planned project duration in calendar days was also considered, possibly in combination with work hours. However, ultimately limiting the relevant training projects solely by planned work hours seemed more prudent for two reasons:

- **Reliability of data:** It is known that the planned end date field does not actually always represent what the field name suggests. Sometimes this field is used to mark the project end date after the fact. Thus it doesn't reliably represent information that would've actually been available at the time a prediction was made. Planned hours is less vulnerable to this issue; while it gets updated if additional work hours are agreed with the customer, it's not updated in response to a project falling short or running past initial estimates like end date occasionally is.

- **Data set size concerns:** Limiting the training data based on both work hours and duration would potentially reduce the available training data per project to an unacceptable degree. For a larger set of data, this might be feasible, but the ATR works data set is too small for this.

In practice, the scaling will be implemented in the following way: For each testing project, we will calculate a lower and upper bound of planned hours. All training data must be from projects that fall between these bounds. Bounds are calculated by multiplying the testing project's planned hours with some multiplier. The multipliers can be set separately, so if necessary, the models can be biased towards under- or overestimating remaining project effort by simply adjusting the multipliers to include more or less data from smaller or larger projects.

An initial investigation into what kind of training data set sizes we can expect to get per project was conducted for various upper and lower bound values. This investigation suggested that for the vast majority of projects, at least hundreds of training data rows would be available for bounds on the order of $\pm 50\%$ of target project planned hours. This was deemed good enough.

5.2.2 Experiment setup

- **Data:** Data is split into training and testing sets along a cut-off date of 1.4.2020. Data from projects that ended before that date forms the training set, and data from after that date forms the testing set. Furthermore, data from projects that were identified as outliers in terms of scope in the initial analysis is excluded, this resulted in the discarding of approximately

1000 training data rows and 100 testing data rows. The remaining train-test split was deemed acceptable. The feature columns are z-score standardized to ensure they all have comparable magnitudes.

Different feature subsets will be used for modeling depending on project run time at the point of prediction. The different subsets are listed in appendix C.

Initially, each testing project's training set will consist of projects whose scale is between 0.5 and 1.5 times the testing project's planned hours (ie. within 50 percent). Different upper and lower bounds may be tested to determine the resulting bias in predictions.

- **Models:** As random forests performed the best in the general model experiment, we will be using only them in this experiment. In the previous experiment, increasing tree count above 300 yielded negligible performance improvement, so in this experiment we will use only 300-tree models. At this number, the training time is still reasonable.

  As explained above, up to three different models will be trained per project. These represent project run-times of less than 4 weeks, 4-12 weeks, and 12 weeks or more.

  Random state is set to zero for all models, to ensure repeatable, comparable results over multiple runs.

- **Goals:** We will attempt to predict the values for all of the five target attributes listed in A.4. Four of these are numerical, and require a regression model, the last is categorical and requires a classification model. The prediction will be made for every aggregated booking row in the test set, to simulate predicting project duration at an arbitrary point in time.

  However, in contrast to the general model experiment, in this experiment each single prediction will be made with a model built specifically for that project and run time category (ie. elapsed project time less than 4 weeks, 4-12 weeks or over 12 weeks) that applies at point of prediction.

- **Evaluation:** C-index and mean absolute error will be calculated for the whole testing data set for the regression target attributes. For the classification target attribute, classification accuracy and confusion matrix are calculated.

  As we are operating three distinct categories of models here, we want to know how models perform by category as well. We will thus split the result set from the experiment into subsets by the category of model used, and calculate c-indexes, mean absolute errors and classification accuracies separately for each category. This is to see whether model performance improves as project moves forward and more history data becomes available for analysis.

  Regression results will be compared to baselines established in tables 5.1 and 5.2. Classification results will be compared to the majority voter score.

5.2.3 Results and discussion

Regression

Basic performance metrics for the project-specific models (PSM) can be seen in table 5.6.

C-index values follow a similar trend to what was seen with the best random forest models in the general model experiment. That is, the c-index for absolute amount of days left in project is noticeably lower (sub-0.7) while for the other label attributes the c-indexes are in the 0.75-0.77 range. The expert baseline (table 5.2) generally achieves higher c-index values, with the only exception being absolute amount of work hours left, where expert baseline is 0.720, ie. slightly lower than what the PSM achieves. As with the general model, it appears that the PSM is reasonably good at keeping the internal order of test records intact, but is still

Measurement	Attribute	Result
C-index	Hours left (abs)	0.743
C-index	Days left (abs)	0.683
C-index	Hours left (pct)	0.752
C-index	Days left (pct)	0.774
MAE	Hours left (abs)	34.8
MAE	Days left (abs)	48.4
MAE	Hours left (pct)	0.152
MAE	Days left (pct)	0.147

Table 5.6: Overall regression performance for whole testing data set, with training data scale bounds set as described in experiment setup.

slightly inferior to the expert estimation in use. Median baseline (table 5.1) is of course exceeded considerably.

With the mean absolute error scores, the PSM outperforms the best general model and both of the baselines, and with a considerable margin in some cases. The only exception is the percentage of hours left, where the best general model random forest achieved the same result (within measurement precision) as the PSM did. For absolute calendar days left, the best general model, median baseline and PSM were also very close, though PSM was the best by a slight margin. The largest improvement was seen with absolute amount of hours left. Here the PSM achieved a mean absolute error of 34.8, a drop of approximately 25 units compared to both baselines, and a drop of 17 units compared to the best general model. Percentage of days left achieved some improvement as well, though the difference was only 2 percentage points from the best general model. Baselines of course have much worse scores for the MAEs of both percentage attributes.

Overall, PSM achieved either similar or slightly better performance than the best general models, and generally outperformed both baselines in terms of mean absolute error. C-index scores were equal or slightly worse than what the expert baseline achieved.

Classification

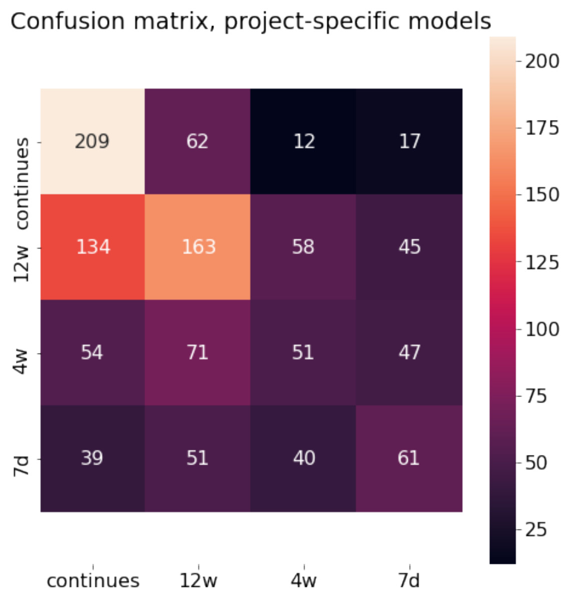

Figure 5.10: Confusion matrix for the project-specific models over the full test set.

The PSM overall classification accuracy was 43.4%. This is a drop of approximately 5 percentage points from what the best general model was able to achieve (48.2%). A majority voter achieved an accuracy of 39.1% with this test data set. The difference between PSM and majority voter is small enough that PSM can be considered essentially useless as a classification tool in this scenario. The confusion matrix for PSM was also plotted, and can be seen in figure 5.10. The general trend there is fairly similar to what was seen in the random forest general model (figure 5.4), with the noticeably worse ability to correctly predict the "continues" label appearing to be the main reason why the overall classification accuracy is lower here.

Model usefulness

The considerable improvement in predicting the absolute amount of hours left suggests that the project-specific models may in fact be a worthwhile additional tool for project planning. This improvement needs to be understood in the context of the other performance measures however. The c-indexes suggest that PSM isn't necessarily better at predicting the order of project completions than the current expert estimations, and in terms of predicting remaining calendar days, the improvement offered by PSM over the expert baseline is also fairly minor.

This suggests that the primary use of the project-specific models would be effort estimation, with duration estimation being much less useful. At this level of accuracy, the model would still be simply an additional tool for management, rather than a complete solution for the problem however.

In order to apply such a tool more efficiently, some more detailed looks into the produced results are merited.

5.2.4 Follow-up: Model performance by type

The project-specific models are made up of three distinct types of models, with different feature subsets formed on the basis of elapsed project run time at the point of prediction. As the previous experiment uncovered some differences in feature usefulness, a look into how different PSM types perform is merited.

This is carried out by separating the PSM result data into model type -specific subsets and calculating the basic regression performance measures (c-indexes, mean absolute errors and classification accuracy) for each subset.

Regression

The regression performance metrics can be seen in tables 5.7 - 5.9.

It appears that as projects run longer and more features can be incorporated

Measurement	Attribute	Result
C-index	Hours left (abs)	0.810
C-index	Days left (abs)	0.707
C-index	Hours left (pct)	0.786
C-index	Days left (pct)	0.719
MAE	Hours left (abs)	34.6
MAE	Days left (abs)	44.7
MAE	Hours left (pct)	0.170
MAE	Days left (pct)	0.178

Table 5.7: Performance measures for project-specific models targeted at projects that have less than 4 weeks of elapsed run time.

Measurement	Attribute	Result
C-index	Hours left (abs)	0.776
C-index	Days left (abs)	0.727
C-index	Hours left (pct)	0.702
C-index	Days left (pct)	0.716
MAE	Hours left (abs)	34.6
MAE	Days left (abs)	46.5
MAE	Hours left (pct)	0.179
MAE	Days left (pct)	0.160

Table 5.8: Performance measures for project-specific models targeted at projects that have run for at least 4 weeks, but less than 12.

into the analysis, most performance metrics actually deteriorate to some degree. A notable exception are the percentage-based hours and days left attributes, these actually see a considerable (around 5 percentage point) performance boost once the

Measurement	Attribute	Result
C-index	Hours left (abs)	0.689
C-index	Days left (abs)	0.645
C-index	Hours left (pct)	0.685
C-index	Days left (pct)	0.695
MAE	Hours left (abs)	35.1
MAE	Days left (abs)	51.4
MAE	Hours left (pct)	0.125
MAE	Days left (pct)	0.124

Table 5.9: Performance measures for project-specific models targeted at projects that have at least 12 weeks of elapsed run time.

full 12-week feature set is used.

The mean absolute error of the absolute hours left prediction is remarkably consistent across all model types, but its c-index with the most limited feature set is notably high (0.810), suggesting a significant improvement over the expert baseline (0.720) in ordering projects by effort early on in the project run time.

This further supports the feasibility of the idea of using PSM to predict required effort for a project, particularly in the early part of project run time. Additionally, while absolute hours/days predictions produced later on in the project run time may be of questionable value, the percentage predictions approach useful accuracy once the full feature set is in use.

Classification

The breakdown of classification accuracies (see table 5.10) is quite interesting. The shorter run-time models perform much better than baseline, and in fact reach comparable accuracies to what the best general models achieved. The long run-time

Model type (elapsed project run time)	Type-specific Acc%	Baseline Acc%
$<$ 4 weeks	46.2	33.6
4-12 weeks	49.1	37.1
$>$ 12 weeks	38.5	36.2

Table 5.10: Classification accuracies for each model type (project run time at point of prediction). Baseline (majority voter) score for each test data subset also included for comparison.

models however perform much worse, with an accuracy approximately 10 percentage points lower than that of the other types, and only slightly above the baseline.

This is interesting because in the previous experiment several 12-week features were identified by random forest internal scores as fairly relevant (see figure 5.9), yet here their absence improves the classification accuracy to a significant degree. Furthermore, since we're using random forests here as well, the internal feature selection should be able to mostly negate counterproductive features. Unfortunately, further analysis of this result is not feasible here, as this classification accuracy is the composite result of a total of 49 random forest models, which would have to be analyzed separately for insights into the causes of poor classification accuracy in this bracket.

It is also possible that this result is simply some artefact of the particular test data subsets, since we're already dealing with fairly limited amounts of data.

5.2.5 Follow-up: Biasing the regression model

As mentioned in experiment introduction, there is some interest in knowing whether the model is more likely to over- or under-predict the remaining days and hours. The ability to set the lower and upper bounds of scale for projects included in the training data for a given model could possibly be used to bias the model in one direction or another. A simple way to measure this is to count the number of predictions that were higher than the true value, and comparing the counts between modeling runs with different values of lower and upper bounds.

The initial bounds were 0.5 lower, 1.5 upper. Two other sets of bounds were tested, and the results can be seen in table 5.11. There are a total of 1114 records in the test set, so any value greater than 557 suggests a model is more likely to overestimate the remaining hours/days.

Lower and upper bounds	Predicted hours > actual hours	Predicted days > actual days
0.3 - 1.25	624	714
0.5 - 1.5	669	744
0.75 - 2.0	703	744

Table 5.11: Number of times the models with different training data scale bounds overestimated the remaining hours and days.

It turns out that the initial, balanced scale bounds (0.5 - 1.5) produce a model that is significantly more likely to overestimate a project's work hours and duration than underestimate it. Even scale bounds that significantly favor smaller training projects (0.3 - 1.25) still produce a model that will more likely overestimate a project than underestimate it, though in terms of work hours this model is closer to an even split.

In terms of work hours, the trend is clear here: The more room is given for projects larger than the one being estimated, the more likely the model is to overestimate. This suggests that if biasing the model in one direction or another is desirable, it can be done by simply directing the training data selection via the scale bounds.

For remaining days, the result is more ambiguous. Increasing the bounds from the initial balanced scale didn't produce a change in the number of overestimations, but this could be caused by the distribution of the training data set, as we know from the initial project scale plotting that data does get more sparse as project sizes increase. The lower bounds did produce a decrease in overestimations compared to the balanced bounds, though the model remained very heavily biased towards overestimation.

In either case, the distribution of the scale of the projects in the data set does put limits on how far the biasing can be taken. If a given test project is already among the largest in the data set, the estimates for it can't be biased upwards by much simply because there isn't any significant amount of larger scale projects to train the model with. The same holds true for biasing estimates downwards for projects that are among the smallest ones in the data set. A good understanding of the distribution of project scale in the data set is essential to obtain desirable results from altering the scale bounds.

5.3 Analysis of modeling results

Overall, we are able to produce models that outperform the baselines, at least to some degree. The ability to outperform the expert baseline (table 5.2) in particular is an important milestone, because it represents an improvement over the current estimation process in use at ATR. However, in absolute terms even the best model performances achieved still aren't good enough to exclusively rely on in project planning, though they can potentially provide useful information to support project estimation tasks.

Generally, it seems preferable to frame project estimation as a regression problem, rather than classification. Classification models commonly had trouble outperforming even a simple majority voter by a significant degree, and even the best models only predicted the correct class roughly 50% of the time. In practice this means that the predicted classification can't be trusted, and the confusion matrices (best general model: figure 5.4 and PSM: figure 5.10) suggest there isn't much useful information to be derived from a wrong classification either. The only clear misclassification pattern is models predicting that projects which actually end in 12 weeks will continue past that, which isn't useful given the estimation goals stated by ATR management.

The most promising modeling approach is estimating the remaining project hours with a project-specific model, trained on data filtered to only contain projects of roughly similar scale. With this modeling approach, both the remaining absolute and percentage amounts of work hours reached considerably better mean absolute error values than the expert baseline, with comparable c-index values suggesting that the internal order of the predicted records was also retained quite well.

The performance improvement achieved by PSM over general models in this regard suggests that applying scale filters to the training data is a good way to improve regression accuracy. It should be noted however, that the accuracy gain was mainly seen in the remaining hours prediction. The remaining days prediction improved only marginally, and indeed this minor improvement may have been a result of the outlier removal or some other aspect of the project-specific models, rather than the training data scaling. This suggests that there is a strong correlation between planned and actual work hours, but weak or nonexistent relation between planned hours and project duration.

5.3.1 Other findings

Usefulness of the project task breakdown features

An important aspect of the data preparation and modeling process was the idea of predicting project end by looking at what kind of a task category breakdown each project has at a given time. However, the results of the feature selection follow-up in the general model section (see 5.1.5) suggest the task breakdown attributes are quite inconsequential for regression in that scenario. In classification, some of the task breakdown attributes appear more useful, but classification as a whole performs quite poorly.

The type-specific performance data for project-specific models (see 5.2.4) also suggests that in some cases adding features is actually detrimental, though in this case the feature selection was being done based on the length of project history the attributes were describing, rather than the task breakdown attribute subsets themselves.

This calls into question the usefulness of the whole idea of using task category breakdowns to predict remaining project days/work hours. However, that judgement cannot be conclusively made based on this data, because there are some known data quality issues that could skew the findings here. Most importantly, the task category data was not actually part of the ATR Works data set, it was manually added after the fact, based on the author's best judgement about what broader category a given task might belong to. While this is fairly straightforward for projects whose tasks followed a common planning-implementation-testing-etc. pattern, even with these projects there is the question of how the tasks were actually used in the hours logging process. For an example, ATR personnel may have only been involved in the development part of a project, so the testing tasks were never used.

In other projects the tasks may have been defined based on features needed in the system being worked on. In such a case, there likely has been some prioritization involved, but that information isn't logged into ATR Works, so there was no possibility to categorize these based on which part of the project life cycle they would be done in. In this case they were simply collected in the "feature" category, which perhaps unsurprisingly doesn't appear to be of much use for prediction of remaining work.

In small projects especially it was quite typical to have only a single "project work" type task defined, in which case again there usually isn't any obvious project life cycle implication in the task.

The results here are sufficiently ambiguous that it can't be said with any certainty whether the concept is useful nor not.

Extended downtime in tail-end of project hours data

During the implementation of the project-specific models, manual inspection of some testing data revealed that there were a fairly substantial amount of rows which represented weeks during which no hours were logged to the project. Further investigation revealed that this was commonly caused by a project essentially becoming complete at some point, but not being closed then (especially if there were still planned/contracted hours left in it). Then weeks or even months later, some minor tweaks would be made to the project and those hours would be logged to the original project.

This causes the project to appear to have taken much longer in terms of time than in actually did, since there's a considerable amount of downtime included in the timeline between the first and last booking. As we aggregated the booking data by splitting the timeline between the project's first and last booking to weeks and filling each row with the booking data from that week, the end result is the aforementioned situation where many aggregated booking rows actually represent project downtime.

Downtime or periods of lesser activity being represented in the data are to be expected, but this kind of extended downtime at the tail-end of a project may be quite problematic for analysis conducted based on the task category breakdown concept. In practice, it means that a number of projects have long tail-ends consisting of almost nothing but downtime, which can greatly skew a model's perception of what kind of task category breakdown precedes project end.

Semantic ambiguity of the project end field

As mentioned before, it turned out during the analysis process that the project end date field is initially filled as planned project end date, but the value often gets updated to represent the actual project end/closure date, once the project finishes. Fields with a dual meaning like this are quite problematic from a data analysis point of view, since it means that during the analysis we cannot really know whether the field means planned or actual end at any given time. Ultimately it was decided to use the field in the planning sense throughout the analysis, with the mutual understanding between author and ATR Soft that this may impact the reliability of duration estimations.

The project planned hours field suffers of this problem to a lesser degree. It is only updated to represent any extra hours agreed with the customer, so it doesn't suffer from the semantic dual meaning that the end date does, but again there is the possibility that the planned hours value has changed during the project.

The exact impact of these issues cannot be quantified with the data we have at hand.

5.4 Future work

There are a number of potential avenues for future work.

5.4.1 Modeling approaches

To be able to effectively utilize and improve the models, a still deeper understanding of their behavior is necessary. While the project-specific models were able to significantly outperform the expert baseline in predicting the absolute amount of hours left in a project, we don't know how large the error is in proportion to the project's estimated/actual scale. An error of 30 hours in the estimation is acceptable for a project whose scale is on the order of 1000 hours total, but for a 20-hour project it would be unacceptable.

Analyzing the behavior of the expert baseline predictions in more detail and comparing the results of that to the model performance measures obtained in this work would also be useful. This might more clearly highlight situations where data-driven estimation could be a useful addition to the estimation process.

Additionally, we know from discussions with ATR management that accurate estimations are particularly valuable during the last month of a project's run time, but we currently don't know how our models' performance changes as the project's end approaches. Thus investigating the prediction accuracy trends as project end approaches is something that would need to be done before the models could be considered for deployment.

The type-specific results of project-specific models (see 5.2.4) suggest some interesting trends in modeling results when longer time span history features are added, but thorough analysis of the causes of this was beyond the scope of this work. A more detailed look into this topic could offer some insight into feature usefulness, with potential tie-ins to the aforementioned interest in model performance trends over project duration.

With project-specific models, we sought to ensure the training data for a given model would consist of projects that are similar to the one being modeled. Here the training data filtering was based on very simple logic, so it might be possible to improve this process by finding new ways to measure similarity between projects. The simplest addition would be to scale the training projects by duration in addition to work hours. No unsupervised learning was conducted in this work, but some kind of a clustering process might also be utilized here to find similar training projects.

Training data set size concerns have to be kept in mind in such endeavors however. A good definition of similarity is not useful if it ends up reducing the size of the training data set too far.

The utilization of the booking description fields was limited to some simple keyword matching, but some kind of more complex text comprehension could be applied to try and extract information out that data as well. This is a more open-ended problem however, that would likely require a much larger scale of effort than the previous suggestions.

The interplay of outlier detection and the test set representativeness concerns related to the use of cut-off dates brought up a potentially interesting research topic that is entirely separate from the concrete modeling issues dealt with above. Namely, what kind of projects can a given organization expect to be able to accurately estimate based on the project data that the organization produces? In ATR's case, projects whose scale exceeds certain limits likely can't ever be accurately estimated by data-driven methods. This is because the organization doesn't work on such projects often enough to accumulate a useful amount of data about them before the oldest data becomes outdated (due to process changes etc). Analysis of an organization's general project profile might make it possible to determine what kind of projects the organization could feasibly estimate with data-driven methods. This would assist in picking the right estimation tools for each project, potentially improving estimation quality considerably.

Another broader research topic would be to implement some of the classical effort estimation tools and methods mentioned in section 2.3 in a proof-of-concept fashion, and comparing the results from that to those obtained in this thesis.

5.4.2 Data collection approaches

A number of data quality issues were identified during the modeling process. While some of these could possibly be tackled in some way during the data pre-processing and modeling, these should also be addressed in the wider work hours collection process and data model to better support potential future analysis projects.

The task categories were applied manually after the fact in this work, but adding them to the ATR Works data model would likely improve the quality of this data considerably. While the overall project management process at ATR Soft requires the project task definitions to remain at the discretion of the relevant project manager, a mandatory categorization field could be added to tasks, to indicate where in the project life cycle that task is likely to reside.

The project end date field's dual meaning should be addressed. This could be done by adding a separate field for planned project end. This might still be updated if contracts are modified during project run time (like the planned hours field is handled), but it would still be an improvement over the current dual meaning the project end field has. Another option would be to use the existing field as the planned end date, and simply show the date of the project's last booking as its actual end date once the project has been closed.

Some way is needed to identify the cases where a project has essentially ended, but it remains open for months until the final tweaks are requested and delivered. This is an issue that could be identified and addressed during data pre-processing as well, though it likely would need some manual work to judge whether a given stretch of project downtime represents this issue or not. Alternatively, this issue could be addressed in the project management process by closing the original project when the initial delivery is made, and making a separate project with the remaining hours to be used for later post-delivery support tasks. This could be implemented entirely as a process change, though it might be worthwhile to support this case directly in

ATR Works by creating a new category of project for this situation, and giving users the option to automatically create a post-delivery support project during original project closure.

6 Conclusions

6.1 Answers to research questions

How is machine learning currently utilized to support project management / software engineering work?

Applying machine learning to various parts of the software engineering process is an idea that has seen fairly consistent interest for several decades already [2] [8]. As the industry has grappled with the problem of increasing complexity of software for most of its existence, there has been a constant demand for better tools, methods and processes for managing the complexity and the uncertainty resulting from it at all levels of software development.

The recent surge of interest in AI has brought the concept of various AI-powered development tools to the forefront [28] [29]. In addition to tasks like code completion, bug detection and testing, AI is being explored for higher-level architecture tasks like detecting code smells and assisting in UI design [30] [31]. For lower-level day-to-day development tasks, at least larger organizations have already made practical deployments of assorted AI-powered tools [29], but the higher-level tools appear to be still some ways away from being ready for practical deployment [31].

For project estimation and management tasks, the interest in AI-based tools has been more consistent over recent decades, and classical machine learning methods like decision trees, the nearest neighbor method and various forms of curve fitting have been explored and applied since the 1980s at least. In the field of software project estimation, these form a part of an estimation method category called data-driven methods. However, while such methods have existed for decades already, the software engineering industry still mostly uses some form of expert evaluation to estimate project effort requirements. This is because relevant project data needed by data-driven methods tends to be quite scarce in most organizations, so methods requiring it are often not feasible [3] [10]. While machine learning research is certainly still conducted with the goal of improving software development effort and duration estimation capabilities [1], this fundamental issue of data availability will likely continue to limit the practical deployments of data-driven estimation methods for the foreseeable future.

The existing work in the field of software development effort estimation is mostly focused on estimating things at the project level and early on in the planning phases of a project. Classical effort estimation methods do often support project tracking and the refining of estimations later on in the project life cycle as well (with separate and/or refined models in some cases), but usually this isn't the main focus [1] [3] [10]. While estimating the availability of individual developers is not unheard of [4], and project-level methods will of course account for the characteristics of the relevant development teams and team members at least to some degree, the particular scenario of predicting employee workloads from their logged work hours appears to be a fairly scarcely covered topic. For these reasons, it was determined that the ATR Works data would need to be analyzed with a custom machine learning solution, rather than a ready-made tool or product.

Is it possible to predict an employee's workload from their reported hours data?

The goal of directly predicting employee workloads from their logged work hours was determined to be unfeasible early on in the analysis process. There was no indication that any useful pattern could be found while looking at the data for individual employees, so the focus of the analysis was shifted to the project level.

Aggregating the work hours data to provide project-level insight into effort and time requirements could still indirectly assist with the employee workload estimation issue, while also providing improved project planning and tracking tools. Ultimately it was determined that there is some promise in this approach, as some of the models trained here were able to outperform the baseline formed by estimation methods currently in use at ATR Soft. A considerable amount of further work would be required before a practical deployment of the developed methods would be feasible however.

How might the hours logging process be improved to better support data analysis projects in the future?

As expected, a number of concrete improvement suggestions to the work hours logging process were discovered during the practical work part of this thesis. The specific suggestions can be found in section 5.4.2. These mostly deal with removing various ambiguities in the logged hours data, as well as seeking to formalize certain patterns by making them an explicit part of the hours logging system's data model, rather than something applied in an ad-hoc manner with existing structures. The suggested improvements should improve both the quality and quantity of available data.

A considerable amount of data had to be excluded from the analysis due to missing values and inconsistencies that couldn't be addressed in the analysis process.

Ensuring that data is recorded without such issues to begin with would help to minimize the amount of data that needs to be pruned from the set during pre-processing.

6.2 Other findings

Even with the aforementioned issues, the scarcity of relevant project data that commonly hampers data-driven effort estimation methods was less of a problem with the approach used in this work. The typical problem is that there aren't enough similar historical projects to base an estimate on, but work hours data from individual employees is relatively plentiful in comparison. The time-series nature of the data still caused some data set size and representation concerns, but analysis was still able to be conducted, even though aggregation and pruning various missing values and outliers shrunk the initial data set considerably. As this kind of data is commonly collected in many other industries as well, there might be some broad cross-industry appeal for analysis along the same lines.

References

[1] P. Pospieszny, B. Czarnacka-Chrobot, and A. Kobyliński, “An effective approach for software project effort and duration estimation with machine learning algorithms”, *Journal of Systems and Software*, vol. 137, Nov. 2017. DOI: 10.1016/j.jss.2017.11.066.

[2] S. Kumar, B. A. Krishna, and P. S. Satsangi, “Fuzzy systems and neural networks in software engineering project management”, *Journal of Applied Intelligence*, vol. 4, pp. 31–52, 1994. DOI: 10.1007/BF00872054.

[3] A. Trendowicz, J. Münch, and R. Jeffery, “State of the practice in software effort estimation: A survey and literature review”, vol. 4980, Sep. 2011, pp. 232–245, ISBN: 978-3-642-22385-3. DOI: 10.1007/978-3-642-22386-0_18.

[4] R. Sawarkar, N. K. Nagwani, and S. Kumar, “Predicting available expert developer for newly reported bugs using machine learning algorithms”, in *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, 2019, pp. 1–4. DOI: 10.1109/I2CT45611.2019.9033915.

[5] K. Schwalbe, *Information Technology Project Management.* Cengage Learning, 2016.

[6] A. Alami, “Why do information technology projects fail?”, *Procedia Computer Science*, vol. 100, pp. 62–71, Dec. 2016. DOI: 10.1016/j.procs.2016.09.124.

[7] *Chaos report 2015*, `https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf`, Accessed: Nov 23, 2020.

[8] D. Partridge, *Artificial Intelligence and Software Engineering: Understanding the Promise of the Future*. Glenlake Publishing Company, Ltd., 1998.

[9] M. Usman, E. Mendes, F. Neiva, and R. Britto, "Effort estimation in agile software development: A systematic literature review", Sep. 2014. DOI: `10.1145/2639490.2639503`.

[10] A. Trendowicz and R. Jeffery, *Software Project Effort Estimation: Foundations and Best Practice Guidelines for Success*. Springer Publishing Company, Incorporated, 2014, ISBN: 3319036289.

[11] U. S. B. of Naval Weapons. Special Projects Office, *PERT (Program Evaluation Research Task): Summary Report, Phase 1*. 1958. [Online]. Available: `https://books.google.fi/books?id=Nn88zQEACAAJ`.

[12] B. W. Boehm, *Software Engineering Economics*, 1st. USA: Prentice Hall PTR, 1981, ISBN: 0138221227.

[13] J. Grenning, *Planning poker or how to avoid analysis paralysis while release planning*, `https://wingman-sw.com/papers/PlanningPoker-v1.1.pdf`, Accessed: May 10, 2021, 2002.

[14] M. R. Berthold, C. Borgelt, F. Höppner, and F. Klawonn, *Guide to Intelligent Data Analysis: How to Intelligently Make Sense of Real Data*, 1st. Springer Publishing Company, Incorporated, 2010, ISBN: 1848822596.

[15] L. C. Briand, K. El Emam, and F. Bomarius, "COBRA: A hybrid method for software cost estimation, benchmarking, and risk assessment", in *Proceedings of the 20th International Conference on Software Engineering*, 1998, pp. 390–399. DOI: `10.1109/ICSE.1998.671392`.

[16] *General data protection regulation legal text*, https://eur-lex.europa.eu/eli/reg/2016/679, Accessed: Jan 27, 2021.

[17] S. Marsland, *Machine Learning: An Algorithmic Perspective*, 2nd. Chapman & Hall/CRC, 2014, ISBN: 1466583282.

[18] E. Alpaydin, *Introduction to Machine Learning*. The MIT Press, 2014, ISBN: 0262028182.

[19] S. Franklin, *Artificial Minds*. Cambridge, MA, USA: MIT Press, 1995, ISBN: 0262061783.

[20] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.", *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958, ISSN: 0033-295X. DOI: 10.1037/h0042519.

[21] M. Minsky and S. A. Papert, *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 1969, ISBN: 0262630222.

[22] O. Simeone, *A brief introduction to machine learning for engineers*, 2018. arXiv: 1709.02840 [cs.LG].

[23] E. Fix and J. Hodges, *Discriminatory Analysis: Nonparametric Discrimination: Consistency Properties*. USAF School of Aviation Medicine, 1951.

[24] T. Cover and P. Hart, "Nearest neighbor pattern classification", *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967. DOI: 10.1109/TIT.1967.1053964.

[25] L. Breiman, "Random forests", *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001, ISSN: 0885-6125. DOI: 10.1023/A:1010933404324.

[26] M. A. Little, G. Varoquaux, S. Saeb, L. Lonini, A. Jayaraman, D. C. Mohr, and K. P. Kording, "Using and understanding cross-validation strategies. Perspectives on Saeb et al.", *GigaScience*, vol. 6, no. 5, Mar. 2017, ISSN: 2047-217X. DOI: 10.1093/gigascience/gix020.

[27] A. Silva, *Concordance index as an evaluation metric*, `https://medium.com/analytics-vidhya/concordance-index-72298c11eac7`, Accessed: Apr 27, 2021.

[28] M. Harman, "The role of artificial intelligence in software engineering", in *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*, 2012, pp. 1–6. DOI: `10.1109/RAISE.2012.6227961`.

[29] D. Schatsky and S. Bumb, *AI is helping to make better software*, `https://www2.deloitte.com/us/en/insights/focus/signals-for-strategists/ai-assisted-software-development.html`, Accessed: Mar 26, 2021.

[30] V. Potluri, T. Grindeland, J. E. Froehlich, and J. Mankoff, "AI-assisted UI design for blind and low-vision creators", in *ASSETS'19 Workshop: AI Fairness for People with Disabilities*, Pittsburgh, PA, 2019.

[31] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis", *Information and Software Technology*, vol. 108, pp. 115–138, 2019, ISSN: 0950-5849. DOI: `10.1016/j.infsof.2018.12.009`.

[32] L. Bao, Z. Xing, X. Xia, D. Lo, and S. Li, "Who will leave the company?: A large-scale industry study of developer turnover by mining monthly work report", in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 170–181. DOI: `10.1109/MSR.2017.58`.

[33] *Anaconda software distribution*, version 2020.07, 2020. [Online]. Available: `https://anaconda.com/`.

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine

learning in Python", *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

Appendix A Table of aggregated data attributes

The following tables briefly describe all attributes that were included in / generated for the aggregated project booking data during the data preparation phase of the analysis process. Training/prediction attributes are marked as "feature" in the category field of these tables, to separate them from the attributes intended to be used for tracking or as prediction targets (labels).

Name	Category	Type	Description
proj_id	Tracking	Numeric	Id of the project that this row represents
proj_category	Tracking	Numeric	Category id of the project
proj_status	Tracking	Numeric	Status id of the project
proj_actual_start	Tracking	Datetime	The date when first booking was made to the project
proj_actual_end	Tracking	Datetime	The date when last booking was made to the project
proj_planned_days	Tracking	Numeric	How many days was the project planned to last
proj_planned_hours	Tracking	Numeric	How many work hours the project was planned to need
proj_actual_hours	Tracking	Numeric	How many work hours were needed to complete the project
proj_actual_days	Tracking	Numeric	How many calendar days were needed to complete the project
proj_spent_days	Tracking	Numeric	How many calendar days have been spent at the end of this span
span_start	Tracking	Datetime	The beginning of the time span that this row depicts
span_end	Tracking	Datetime	The end of the time span that this row depicts
7d_total_hours	Feature	Numeric	The total hours logged in this time span

Table A.1: Features in aggregated data

Name	Category	Type	Description
4w_avg_hours	Feature	Numeric	Average weekly hours from last 4 weeks
12w_avg_hours	Feature	Numeric	Average weekly hours from last 12 weeks
7d_active_employees	Feature	Numeric	How many employees have made bookings in this time span
planned_days_left_abs	Feature	Numeric	How many days of planned duration are left
planned_days_left_pct	Feature	Numeric	Percentage of planned days left
planned_hours_left_abs	Feature	Numeric	How many planned work hours are left
planned_hours_left_pct	Feature	Numeric	Percentage of planned work hours left
7d_unassigned	Feature	Numeric	Percentage of logged hours with named task category in this time span
7d_other	Feature	Numeric	As above
7d_feature	Feature	Numeric	As above
7d_planning	Feature	Numeric	As above
7d_implementation	Feature	Numeric	As above
7d_testing	Feature	Numeric	As above
7d_finalization	Feature	Numeric	As above
4w_unassigned	Feature	Numeric	Percentage of logged hours with named task category in the last 4 weeks

Table A.2: Features in aggregated data

Name	Category	Type	Description
4w_other	Feature	Numeric	As above
4w_feature	Feature	Numeric	As above
4w_planning	Feature	Numeric	As above
4w_implementation	Feature	Numeric	As above
4w_testing	Feature	Numeric	As above
4w_finalization	Feature	Numeric	As above
12w_unassigned	Feature	Numeric	Percentage of logged hours with named task category in the last 12 weeks
12w_other	Feature	Numeric	As above
12w_feature	Feature	Numeric	As above
12w_planning	Feature	Numeric	As above
12w_implementation	Feature	Numeric	As above
12w_testing	Feature	Numeric	As above
12w_finalization	Feature	Numeric	As above
7d_desc_has_test	Feature	Numeric	Percentage of bookings in this time span whose description included keyword "test"
7d_desc_has_bug	Feature	Numeric	As above for keyword "bug"
7d_desc_has_refac	Feature	Numeric	As above for keyword "refactor"
7d_desc_has_deploy	Feature	Numeric	As above for keyword "deploy"
4w_desc_has_test	Feature	Numeric	Percentage of bookings in last 4 weeks whose description included keyword "test"

Table A.3: Features in aggregated data

Name	Category	Type	Description
4w_desc_has_bug	Feature	Numeric	As above for keyword "bug"
4w_desc_has_refac	Feature	Numeric	As above for keyword "refactor"
4w_desc_has_deploy	Feature	Numeric	As above for keyword "deploy"
12w_desc_has_test	Feature	Numeric	Percentage of bookings in last 12 weeks whose description included keyword "test"
12w_desc_has_bug	Feature	Numeric	As above for keyword "bug"
12w_desc_has_refac	Feature	Numeric	As above for keyword "refactor"
12w_desc_has_deploy	Feature	Numeric	As above for keyword "deploy"
actual_hours_left_abs	Target	Numeric	How many work hours are left until project completion
actual_hours_left_pct	Target	Numeric	Percentage of work hours left until project completion
actual_days_left_abs	Target	Numeric	How many calendar days are left until project completion
actual_days_left_pct	Target	Numeric	Percentage of calendar days left until project completion
ends_in	Target	Categorical	Classification by the closeness of project end. Values represent project ending in 7 days / 4 weeks / 12 weeks / continuing for the foreseeable future

Table A.4: Features in aggregated data

Appendix B Tested general model feature subsets

Each of these feature subsets was tested as part of the feature selection follow-up in the general models section (5.1.4).

Full feature set

This is the original set that contains all 41 features. The list of features is:

7d_total_hours, 4w_avg_hours, 12w_avg_hours, 7d_active_employees, planned_days_left_abs, planned_days_left_pct, planned_hours_left_abs, planned_hours_left_pct, 7d_unassigned, 7d_other, 7d_feature, 7d_planning, 7d_implementation, 7d_testing, 7d_finalization, 4w_unassigned, 4w_other, 4w_feature, 4w_planning, 4w_implementation, 4w_testing, 4w_finalization, 12w_unassigned, 12w_other, 12w_feature, 12w_planning, 12w_implementation, 12w_testing, 12w_finalization, 7d_desc_has_test, 7d_desc_has_bug, 7d_desc_has_refac, 7d_desc_has_deploy, 4w_desc_has_test, 4w_desc_has_bug, 4w_desc_has_refac, 4w_desc_has_deploy, 12w_desc_has_test, 12w_desc_has_bug, 12w_desc_has_refac, 12w_desc_has_deploy

Subset 1

This subset was created by removing features that represent task category prevalences other than testing and finalization from the full feature set. The resulting list of features is:

7d_total_hours, 4w_avg_hours, 12w_avg_hours, 7d_active_employees, planned_days_left_abs, planned_days_left_pct, planned_hours_left_abs, planned_hours_left_pct, 7d_testing, 7d_finalization, 4w_testing, 4w_finalization, 12w_testing, 12w_finalization, 7d_desc_has_test, 7d_desc_has_bug, 7d_desc_has_refac, 7d_desc_has_deploy, 4w_desc_has_test, 4w_desc_has_bug, 4w_desc_has_refac, 4w_desc_has_deploy, 12w_desc_has_test, 12w_desc_has_bug, 12w_desc_has_refac, 12w_desc_has_deploy

Subset 2

This set was created by removing the 12-week history columns from subset 1. The resulting list of features is:

7d_total_hours, 4w_avg_hours, 7d_active_employees, planned_days_left_abs, planned_days_left_pct, planned_hours_left_abs, planned_hours_left_pct, 7d_testing, 7d_finalization, 4w_testing, 4w_finalization, 7d_desc_has_test, 7d_desc_has_bug, 7d_desc_has_refac, 7d_desc_has_deploy, 4w_desc_has_test, 4w_desc_has_bug, 4w_desc_has_refac, 4w_desc_has_deploy

Subset 3

This set was created by removing the columns representing keyword matches to words "deploy" and "refactor" from subset 2. The resulting list of features is:

7d_total_hours, 4w_avg_hours, 7d_active_employees, planned_days_left_abs, planned_days_left_pct, planned_hours_left_abs, planned_hours_left_pct, 7d_testing, 7d_finalization, 4w_testing, 4w_finalization, 7d_desc_has_test, 7d_desc_has_bug, 4w_desc_has_test, 4w_desc_has_bug

Subset 4

This set was created by removing the columns representing keyword matches to words "test" and "bug" from subset 3. The resulting list of features is:

7d_total_hours, 4w_avg_hours, 7d_active_employees, planned_days_left_abs, planned_days_left_pct, planned_hours_left_abs, planned_hours_left_pct, 7d_testing, 7d_finalization, 4w_testing, 4w_finalization

Appendix C Limited scope model feature subsets

Each of these feature subsets were used for modeling projects at various stages of their run time.

Subset 1

This is the full feature set, used on project models that represent a situation where the project has run for at least 12 weeks at the point of prediction.

7d_total_hours, 4w_avg_hours, 12w_avg_hours, 7d_active_employees, planned_days_left_abs, planned_days_left_pct, planned_hours_left_abs, planned_hours_left_pct, 7d_unassigned, 7d_other, 7d_feature, 7d_planning, 7d_implementation, 7d_testing, 7d_finalization, 4w_unassigned, 4w_other, 4w_feature, 4w_planning, 4w_implementation, 4w_testing, 4w_finalization, 12w_unassigned, 12w_other, 12w_feature, 12w_planning, 12w_implementation, 12w_testing, 12w_finalization, 7d_desc_has_test, 7d_desc_has_bug, 7d_desc_has_refac, 7d_desc_has_deploy, 4w_desc_has_test, 4w_desc_has_bug, 4w_desc_has_refac, 4w_desc_has_deploy, 12w_desc_has_test, 12w_desc_has_bug, 12w_desc_has_refac, 12w_desc_has_deploy

Subset 2

This subset was created by removing the 12-week history features from the full set. This depicts the situation where a project has run for more than 4 but less than 12 weeks.

7d_total_hours, 4w_avg_hours, 7d_active_employees, planned_days_left_abs, planned_days_left_pct, planned_hours_left_abs, planned_hours_left_pct, 7d_unassigned, 7d_other, 7d_feature, 7d_planning, 7d_implementation, 7d_testing, 7d_finalization, 4w_unassigned, 4w_other, 4w_feature, 4w_planning, 4w_implementation, 4w_testing, 4w_finalization, 7d_desc_has_test, 7d_desc_has_bug, 7d_desc_has_refac, 7d_desc_has_deploy, 4w_desc_has_test, 4w_desc_has_bug, 4w_desc_has_refac, 4w_desc_has_deploy

Subset 3

This subset has had both the 4- and 12-week history features omitted. This represents the situation where the project has run for less than 4 weeks at the time a prediction is made.

7d_total_hours, 7d_active_employees, planned_days_left_abs, planned_days_left_pct, planned_hours_left_abs, planned_hours_left_pct, 7d_unassigned, 7d_other, 7d_feature, 7d_planning, 7d_implementation, 7d_testing, 7d_finalization, 7d_desc_has_test, 7d_desc_has_bug, 7d_desc_has_refac, 7d_desc_has_deploy