Andriy Burkov

# MACHINE LEARNING ENGINEERING

*"An optimist sees a glass half full. A pessimist sees a glass half empty. An engineer sees a glass that is twice as big as it needs to be."*
*— Unknown*

*"Death and taxes are unsolved engineering problems."*
*— Unknown*

The book is distributed on the "read first, buy later" principle.

# 3 Data Collection and Preparation

Before any machine learning activity can start, the right data has to be collected and prepared for machine learning. The data available to the analyst is not always "right" and is not always in a form that a machine learning algorithm can use. In this chapter, I consider the first machine learning engineering stage of the machine learning project life cicle, as shown below:

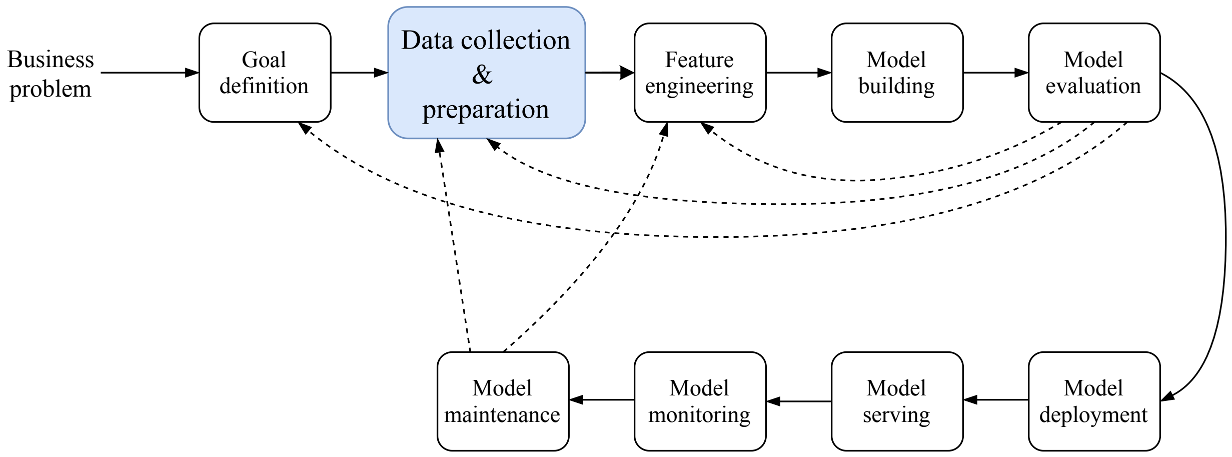


Figure 1: Machine learning project life cycle.

In particular, I focus on the properties of good quality data, typical problems a dataset can have, and ways to prepare and store data for machine learning.

## 3.1 Questions About the Data

Now that you have a machine learning goal with well-defined model input, output and success criteria, you can start collecting the data needed to train your model. However, before you start collecting the data, there are some questions to answer first.

### 3.1.1 Is the Data Accessible?

Does the data you need already exist? If yes, is it accessible (physically, contractually or from the cost perspective)? If you are purchasing or re-using someone else's data sources, have you considered how that data might be used or shared? Do you need to negotiate a new license with the original supplier?

If the data is accessible, is it protected by copyright or other legal norms? If so, have you established who owns the copyright in your data? Might there be joint copyright?

Is the data sensible (concerns your organization's projects, clients or partners, or is classified by the government) and are there any potential privacy issues to care about? If so, have you discussed data sharing with the respondents from whom you collected the data? Can you preserve for the long-term, personal information so that it can be used in the future?

Do you need to share the data along with the model? If so, do you need to get written consent from owners or respondents?

Do you need to anonymize data, for example, to remove **personally identifiable information** (PII), during analysis or in preparation for sharing?

Even if it's physically possible to get the data you need, don't work with it until all above questions are resolved.

### 3.1.2 Is the Data Sizeable?

The question for which you would like to have a definitive answer is whether there's enough data. However, as I already mentioned, it's usually not known in advance how much data is needed to reach your goal, especially if the minimum accuracy requirement is very strict.

If you have doubts about the immediate availability of the sufficient amount of data, find out how frequently the new data gets generated. For some projects, you can start with a small initially available dataset and, while you are working on feature engineering, modeling, and solving other relevant technical problems, the new data might gradually come in. The new data can come in either naturally, as a result of some observable or measurable process, or be gradually provided by your data labeling experts or a third-party data provider.

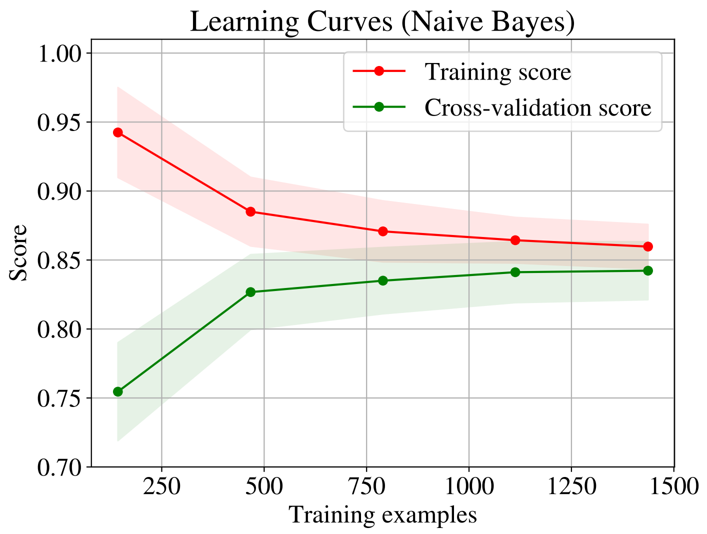


Figure 2: Learning curves for the Naïve Bayes learning algorithm applied to the standard *digits* dataset of scikit-learn.

Consider the estimated time needed to accomplish the project. Will a sufficiently large dataset be gathered during this time? Base your answer on the past experience working on similar projects or results reported in the literature.

One practical way to find out if you have collected sufficient data is to plot **learning curves**. More specifically, plot the training and validation scores of your learning algorithm for varying numbers of training examples as shown in Figure 2.

By looking at the learning curves, you will see that the performance of your model will plateau after you reach a certain number of training examples. After reaching that number of training examples, you will begin to experience diminishing returns from adding each additional example.

If you observe that the performance of the learning algorithm plateaued, it *might be* a sign that collecting more data will not help in building a better model. Why I say "might be"? Because two other explanations are possible:

- You don't have enough informative features that your learning algorithm can leverage to build a more complex model, or
- You use a learning algorithm incapable of building a complex enough model using the data you have.

In the former case, you might think about engineering additional features by combining the existing features in some clever ways or by using information from indirect data sources, such as lookup tables and gazetteers.

In the latter case, one approach would be to use an ensemble learning method or train a deep neural network, though neural networks usually require more training data compared to shallow learning algorithms.

Some practitioners use rules of thumb to estimate the number of training examples needed for a problem. Usually, they are using scaling factors applied to either:

- number of features, or
- number of classes, or
- number of trainable parameters in the model.

These rules of thumb often work, but they are different for different problem domains, and each analyst adjusts the numbers based on experience. While each analyst discovers by experience those "magical" scaling factors that work for them, the most frequently cited numbers in various online sources are: 10 times the number of features (this often exaggerates the size of the training set but works well as an upper bound), 100 or 1000 times the number of classes (this often underestimates the size of the training set), or 10 times the number of trainable parameters.

Keep in mind that you might not need all your big data. In fact, just because you have big data does not mean that you should use all of it. A smaller sample of big data can give good results in practice and accelerate the search for a better model. It's important to ensure, though, that the sample is representative of the whole big dataset. Such sampling strategies as **stratified** and **balanced sampling** can lead to better results. We consider data sampling strategies later in this chapter.

### 3.1.3 Is the Data Useable?

The quality of the data is one of the major factors affecting the performance of the model. Imagine that you want to build a model that predicts a person's gender given the name of that person. You might have a budget to acquire a dataset of people with gender information. If you blindly use this dataset you might realize later that no matter how hard you try to improve the performance of your model, its accuracy on new data is low. What is the reason for such a low performance? The answer could be that in the dataset you acquired the gender information was not factual but obtained using a statistical classifier of rather low quality. In this case, the best you can achieve with your own model is the level of performance of that low-quality classifier.

If the dataset comes in the form of a spreadsheet, the first thing to check is if the data in the spreadsheet is tidy. The dataset usable for machine learning has to be tidy. If it's not the case for your data, you will have to transform it into tidy data using, as already mentioned, feature engineering.

A tidy raw dataset can have **missing values**. If it's the case for your data, you can consider **data imputation** techniques to fill the missing values. We will consider several such techniques later in this chapter.

One frequent problem with datasets compiled by humans is that people can decide to indicate missing values with some **magic number** like 9999 or $-1$. During visual data analysis, such situations have to be spotted and these magic numbers have to be replaced using an appropriate data imputation technique.

Another property to validate is whether the dataset contains **duplicates**. Usually, duplicates are removed, unless you added them on purpose to balance an **imbalanced problem**. We consider this problem and methods to alleviate it later in this chapter.

Data can be **expired** or significantly not up to date. For example, your goal is to build a model that recognizes abnormality when a complex piece of electronic appliance, such as a printer, misbehaves. You have measurements taken during normal and abnormal functioning of a printer. However, these measurements have been recorded for a previous generation of printers, while the new generation since then has received several significant upgrades. The model built using such outdated data coming from an older printer generation might perform worse when deployed on the new generation of printers.

Finally, data can be **incomplete** or **unrepresentative** of the phenomenon of the study. For example, a dataset of pictures of various animals can only contain pictures taken during the summer. A dataset of pedestrians for the self-driving car systems could be created by engineers with other engineers playing pedestrians; in such a dataset, most situations on the road would involve younger men, while children, women and elderly people would be underrepresented or not present at all. A company working on a facial expression recognition model could have the research and development office in a predominantly white location, so the dataset they would build might contain many faces of white men and women, while black or Asian people would be underrepresented. Engineers in another company working on a posture recognition model intended to be deployed in a camera could build the training dataset by taking pictures of people indoors, while the customers would use the camera predominantly outdoors.

In many practical cases, data can only become useable for modeling after preprocessing. Whence the importance of visual analysis of the dataset before you start modeling. Let's say you work on a problem of predicting the topic in the news articles. It's likely that you will get your data by scraping news websites. It's also likely that the names of authors of articles and the date when the text was downloaded would be saved in the same document as the text of news articles. Imagine also that the data engineer responsible for scraping decided to loop over news topics mentioned on the news websites and scrape news one topic at a time. So, on Monday the arts-related articles were scraped, on Tuesday — sports, on Wednesday — technology, and so on. It is also likely that names of article authors present in the text reveal the topic of the article because typically the same person writes on one or a few topics.

If you don't preprocess such data by removing the download dates and author names, the model can learn the correlation of specific download dates and names present in the text with the topics. Of course, such a model will be of no use in practice.

### 3.1.4 Is the Data Understandable?

As I already demonstrated above on the example of gender prediction, it is very important to understand where each attribute in the dataset came from. It is equally important to understand what exactly each attribute represents. One frequent problem observed in practice is when the variable which the analyst tries to predict is found among the features in the feature vector. How can this happen?

Imagine that you work on the problem of predicting the price of a house from its attributes such as the number of bedrooms, surface, location, year of construction, and so on. The attributes of each house were provided to you by the client, which is a big online real estate sales platform. The data has the form of an Excel spreadsheet. Without spending too much time on analyzing each column in the spreadsheet, you remove the transaction price from the list of attributes and use that value as the target that you want to model. Very quickly you realize that the model is almost perfect: it predicts the transaction price with the accuracy near 100%. You deliver the model to the client, they deploy it in production and the tests show that the model is wrong most of the time. What happened?

What happened is called the **data leakage** (also known as the **target leakage**). After a careful investigation of the dataset, you realize that one of the columns in the spreadsheet contained the real estate agent commission. Of course, the model easily learned to perfectly convert this attribute into the house price. However, this information is not available in the production environment before the house is sold, because the commission depends on the selling price. Below, we will consider the problem of data leakage in more detail.

### 3.1.5 Is the Data Reliable?

The reliability of a dataset varies depending on the procedure that was used to gather that dataset. Can you trust the labels? If the data was produced by the workers on Mechanical Turk (so-called "turkers") then the reliability of such data might be very low. In some cases, the labels assigned to feature vectors might be obtained as a majority vote (or an average) of several turkers. If it's the case, then the data can be considered more reliable. However, even in that latter case, it's better to do an additional validation of quality on a small random sample of the dataset.

On the other hand, if the data represents measurements made by some measuring devices, you can find the details on the accuracy of each measurement in the technical documentation of the corresponding measuring device.

The reliability of labels can also be affected by the **delayed** or **indirect** nature of the label. The label is considered delayed when the feature vector to which the label was assigned represents some event that happened significantly earlier in time than the time of label observation. To be more concrete, take the **churn prediction** problem for example. In churn prediction, we have a feature vector describing a customer and we want to predict whether the customer will leave at some point in the future (typically 6 months to 1 year from now). The feature vector represents what we know about the customer *now*, but the label (customer left or stayed) will be obtained in the future. This is an important property because between now and the future, many events, not reflected in our feature vector, can happen which can affect the customer's decision to stay or leave. Therefore delayed labels make our data less reliable.

Whether a label is direct or indirect also affects reliability, depending, of course, on what we are trying to predict. For example, let's say our goal is to predict whether the website visitor will be interested in a webpage. We might acquire a certain dataset containing information about users, webpages and labels "interested"/"not_interested" reflecting whether a specific user was interested in a specific webpage. A direct label would indeed indicate the interest, while an indirect label could *suggest some* interest. For example, if the user pressed the "Like" button, we have the direct indicator or the interest. However, if the user only clicked at the link, this could be an indicator of *some* interest, but it's an indirect indicator. The user could click by mistake or because the link text was a clickbait, we cannot know for sure. If the label is indirect, this also makes such data less reliable. Of course, it's less reliable for predicting the interest, but can be perfectly reliable for predicting clicks.

Finally, another source of unreliability in the data is **feedback loops**. A feedback loop is a property in the system design when the data used to train the model was obtained using the model itself. Again, imagine that you work on a problem of predicting whether a specific user of a website will like the content, and you only have indirect labels – clicks. If the model is already deployed on the website and the users click on links recommended by the model, this means that the new data indirectly reflects not only the interest of users to the content but also how intensively the model recommended that content. If the model decided that a specific link is very important to recommend to many users, it's likely that more users will click on that link, especially if the recommendation was made repeatedly during several days or weeks.

## 3.2 What Is Good Data

Good data *contains enough information* that can be used for modeling. For example, if you want to build a model that predicts whether the customer will buy a specific product you will need to possess both the properties of the product in question and the properties of the products customers bought in the past. If you only have the properties of the product and a customer's location and name, then the predictions will be likely the same for all users from the same location. If you have enough training examples, then, potentially, the model can derive the gender and ethnicity from the name and make different predictions for men, women, locations, and ethnicities, but not to each customer individually.

Good data *has good coverage* of what you want to do with the model. For example, if you want to use the model to classify web pages by topic and you have a thousand topics of interest then your data has to contain examples of documents on each of the thousand topics in a quantity sufficient for the algorithm to be able to learn the difference between topics from the data. Imagine a different situation. Let's say that for a certain topic, you only have one or a couple of documents. Let each document contain a unique ID in the text. In such a scenario, the learning algorithm will not be sure what exactly it has to look at in each document to understand to which topic it belongs. Maybe its IDs? They look like good differentiators. If the algorithm decides to use IDs as the main feature to separate these couple examples from the rest of the dataset, then the learned model will not be able to generalize: it will not see any of those IDs ever again.

Good data *reflects real inputs* that the model will see in production. For example, if you build a system that recognizes cars on the road and all pictures you took were taken during the working hours, then it's unlikely that you will have many examples of night pictures. Once you deploy the model in production, pictures will come from all times during the day and your model will more frequently make errors on night pictures. Also, remember the problem of a cat, a dog, and a raccoon discussed above: if your model doesn't know anything about racoons, it will predict their pictures as either dogs or cats.

Good data *is as unbiased as possible*. This property can look similar to the previous one, but bias can actually be present in both the data you use for training and the data that the model is applied to in the production environment. A famous example is looking for associations for words using embeddings trained with an algorithm like word2vec. The model predicts that $king - man + women = queen$ but at the same time it predicts that $programmer - man + woman = homemaker$. We will discuss bias in data and how to deal with it later in this chapter.

Another source of bias can come from human nature. For example, you want to build a model that predicts whether a book will be liked by the readers. You can use the rating users gave to similar books in the past. However, unhappy users tend to give disproportionally low ratings, so the data will be biased towards too many very low ratings as compared to the quantity of mid-range ratings, as shown in Figure 3.

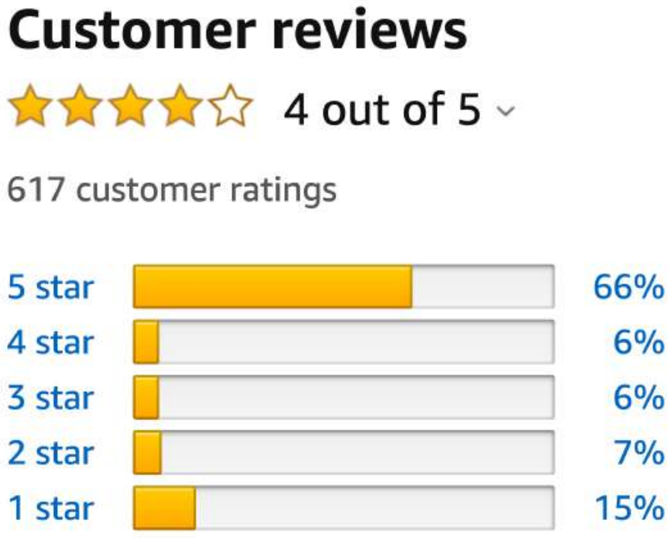


Figure 3: The distribution of ratings given by the readers to a popular AI book on Amazon.

A different example of bias coming from human nature is when you put a poll on the website and ask users to rate something. The ratings you receive will be biased because they will represent the opinion of people who tend to participate in online polls and not the general population.

A user interface can also be a source of bias. For example, you want to predict the popularity of a news article and you use the click rate as a feature. If some news article was displayed on the top of the page, the number of clicks it got would often be higher compared to another news article displayed on the bottom, even if the latter is more engaging.

Good data *is not a result of the model* itself. This echoes the problem of the feedback loop discussed above. For example, you cannot build a model that predicts the gender of a person from their name and then use the prediction as a new training example.

Alternatively, if you use the model to decide which email messages are important to the user and you highlight those important messages, then you should not directly take the clicks on those emails as a signal that the email is important. The user might have clicked on them because they were highlighted by the model.

Good data *has consistent labels.* Inconsistency in labeling can come from several sources:

- Different people do labeling according to different criteria. Even if people believe that they use the same criteria, different people often interpret the same criteria differently.
- The definition of some classes evolved over time. This results in a situation when two very similar feature vectors receive two different labels.

- Misinterpretation of user's motives. For example, assume that the user ignored a recommended news article. As a consequence, this news article receives a negative label. However, the motive of the user for ignoring this recommendation might be that they already knew the story and not that they are not interested in the topic of the story.

Good data *is big enough* to allow generalization. Sometimes, nothing can be done to increase the accuracy of the model. No matter how much data you throw on the learning algorithm: the information that is contained in the data has low predictive power for your problem. However, and it happens more often in practice, you can get a very accurate model if you pass from thousands of examples to millions or hundreds of millions. You cannot know how much data you need until you start working on your problem.

For the convenience of future reference, let me once again repeat the properties of good data:

- it contains enough information that can be used for modeling,
- it has good coverage of what you want to do with the model,
- it reflects real inputs that the model will see in production,
- is as unbiased as possible,
- is not a result of the model itself,
- it has consistent labels,
- it is big enough to allow generalization.

## 3.3 Dealing With Interaction Data

**Interaction data** is the data you can collect from the interactions of the user with the system your model supports. You are considered lucky if you can gather good data from interactions of the user with the system.

Good interaction data contains information on three aspects:

- *context* of interaction,
- *action* of the user in that context,
- *outcome* of interaction.

As an example, assume that you build a search engine and your model reranks search results for each user individually. A reranking model takes as input the list of links returned by the search engine based on keywords provided by the user and outputs another list in which the items of the input list changed order. Usually, a reranked model "knows" something about the user and their preferences and can reorder the generic search results for each user individually according to that user's learned preferences. The context here is the search query and the ten documents presented to the user in a specific order. The action is a click of the user on specific document link. The outcome is how much time the user spent reading the document and whether the user hit "back". Another action is the click on the "next page" link. The intuition is that the ranking was good if the user clicked on some link and spent a significant time reading the document. The ranking was not so good if the user clicked on a link to a document and then hit "back" quickly. The ranking was bad if the user clicked on the "next page" link. This data can be used to improve the ranking algorithm and make it more personalized.

## 3.4 Common Problems With Data

The data you work with can have several problems. In this section, I cite the most important of these problems and what you can do to alleviate them.

### 3.4.1 High Cost

Getting unlabeled data can be expensive; however, labeling data is the most expensive work, especially if the work is done manually.

Getting unlabeled data becomes expensive when the data has to be gathered specifically for your problem. Let's say your goal is to know where different types of commerce are located in a city. The best solution would be to buy this data from a government agency. However, for various reasons it can be complicated or even impossible or the data in the government database can be incomplete or outdated. To get the up-to-date data, you may decide to send cars equipped with cameras on the streets of a given city and those cars would take pictures of all buildings on the streets.

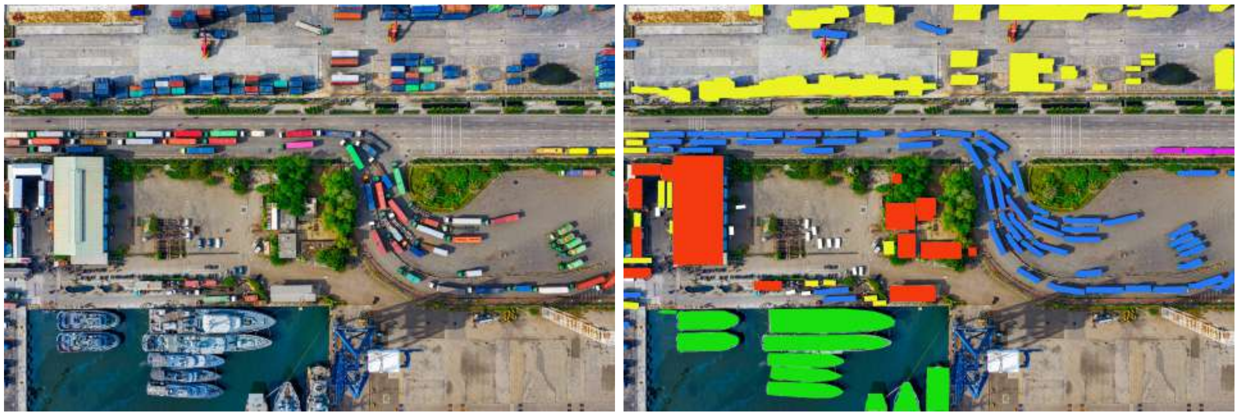
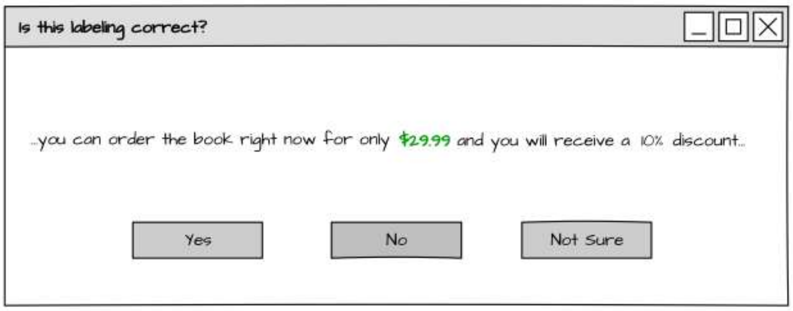


Figure 4: The unlabeled and labeled aerial photo. Photo credit: Tom Fisk.

As you can imagine, such an enterprise is not cheap. However, having pictures of all the buildings in the city is not enough. We want to know the type of commerce in every building. Now we need labeled data: pictures of building facades with labels: "coffee house", "bank", "grocery", "drug store", "gas station", etc. These labels have to be assigned manually and paying someone for doing that work is equally expensive. By the way, Google has got a clever idea to outsource this labeling work to anonymous people as part of its free reCAPTCHA service. reCAPTCHA thus solves two problems: reducing spam on the Web and providing cheap labeled data to Google.

In Figure 4, you can see how much work has to be done in certain cases to label one image. In this task, the goal is to segment a picture by assigning to every pixel labels from the following list: "heavy truck", "car or light truck", "boat", "building", and "container" (everything unlabeled will be considered as "other"). To label this example I spent about 30 minutes. You can imagine that if there were more kinds of objects in the list of labels (for example "motorcycle", "tree", "road", etc), the time would be even longer, and, thus, the cost would be higher.


Is this labeling correct?

...you can order the book right now for only $29.99 and you will receive a 10% discount...

Figure 5: An example of simple labeling interface.

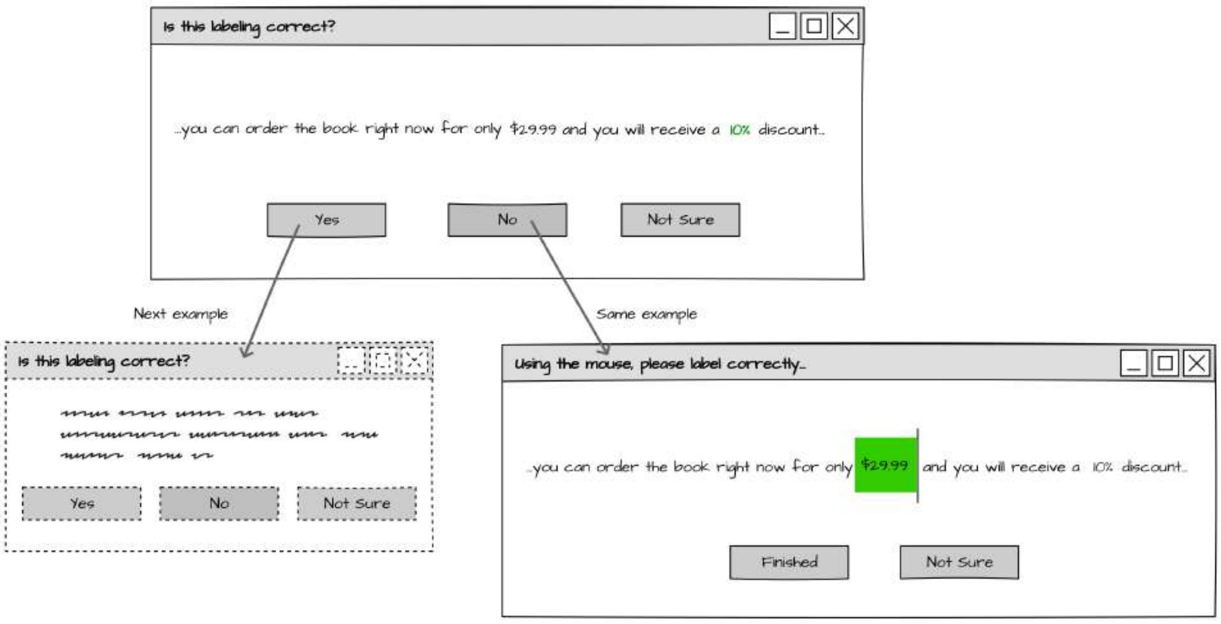

Is this labeling correct?
...you can order the book right now for only $29.99 and you will receive a 10% discount...
Yes
No
Not Sure
Next example
Same example
Is this labeling correct?
Yes
No
Not Sure
Using the mouse, please label correctly...
...you can order the book right now for only $29.99 and you will receive a 10% discount...
Finished
Not Sure

Figure 6: An example of noisy pre-labeling workflow.

Well-designed labeling tools that minimize mouse use (and menus activated by mouse clicks) as well as maximize the use of hotkeys enable cost reduction by increasing the speed of data labeling.

Whenever possible, reduce the decision the labeler has to make to a yes/no decision. Instead of asking "Find all prices in this text.", extract all numbers from the text and then display each number to the labeler one by one in its context by asking "Is this number a price?" as shown in Figure 5. If the labeler clicks the "Not Sure" button, you can save this example to analyze later or simply don't use such examples for training the model.

Another trick allowing for accelerated labeling is **noisy pre-labeling** consisting of pre-labeling the example using the current best model. In this scenario, you start by labeling a certain quantity of examples "from scratch" (that is without using any support). Then you build the first model that works reasonably well using this initial set of labeled examples. Then you use the current model to label each new example in place of the human[1] and ask the human labeler whether that automatically assigned label is correct. If the labeler clicks on "Yes", you save this example as usual. If the labeler clicks on "No", then you ask the labeler to label this example manually. See the workflow chart illustrating that process in Figure 6. The goal of good labeling process design is to make the labeling as streamlined as

[1]This is why it's called a "noisy" pre-labeling: the labels assigned to examples by using a sub-optimal model would not all be accurate and, thus, require a human validation.

possible. Engagement of labelers is also a key consideration: for instance, showing progress in the number of labels added, as well as the value of the performance metric of the current best model is a way to engage the labeler by showing purpose to the labeling task.

### 3.4.2 Bad Quality

I already mentioned that the quality of the data is one of the major factors affecting the performance of the model. I cannot stress it strongly enough.

Data quality has two components: raw data quality and labeling quality.

Some common problems with raw data are noise, bias, low predictive power, outdated examples, outliers, and leakage.

### 3.4.3 Noise

**Noise** in data is a corruption of examples. Images can be blurry or incomplete. Text can lose formatting which makes some words concatenated or split. Audio data can literally have noise on the background. Poll answers can be incomplete or have missing attributes, such as the responder's age or sex. Noise is usually seen as a random process that corrupts each example independently of other examples in the collection. Sometimes noise is added to feature vectors on purpose, for example in **denoising autoencoders**.

If raw examples have the form of a row in a spreadsheet with missing attributes, **data imputation** techniques can help in guessing values for those attributes. We will consider data imputation techniques later in this chapter.

Blurred images can be deblurred using specific image deblurring algorithms, though deep machine learning algorithms such as neural networks can learn to deblur if needed. The same can be said about noise in audio data: it can be algorithmically suppressed.

Noise is more a problem when the dataset is relatively small (thousands of examples or less) because the presence of noise can lead to **overfitting**: the algorithm may learn to model the noise contained in the training data, which is undesirable. In the big data context, on the other hand, noise, if it's randomly applied to each example independently of other examples in the dataset, is typically "averaged out" over multiple examples. In that latter context, noise can actually bring a regularization effect as it prevents the learning algorithm from relying too much on a small set of input features[ˆch3_4].

[ˆch3_4] This is, by the way, the rationale behind the increase in performance brought by the dropout regularization technique in deep learning.

### 3.4.4 Bias

**Bias** in data is inconsistency with the phenomenon it represents. This inconsistency may occur for a number of reasons (which are not mutually exclusive).

#### 3.4.4.1 Types of Bias

**Selection bias** is the tendency to skew your choice of data sources to those that are easily available, convenient and cost-effective for your purposes. For example, you want to know the opinion of the readers on your new book. You decide to send several initial chapters of the book to the mailing list subscribers of your previous book and ask them whether they like the new book. It's very likely that the readers of your previous book subscribed to the mailing list will like your new book, however, this information doesn't tell you anything about a general reader.

**Self-selection bias** is a form of selection bias where you get the data from sources that "volunteered" themselves to provide you that data. Most poll data has this type of bias. For example, you want to train a model that predicts the behavior of successful entrepreneurs. You decide to ask questions of entrepreneurs by letting them respond whether they are successful or not. Then you only keep the data obtained from the entrepreneurs who declared themselves successful. The problem here is that, most likely, really successful entrepreneurs don't have time to answer your questions, while those who declare themselves successful can be wrong on that matter.

**Omitted variable bias** happens when your featurized data doesn't have a feature important for accurate prediction. For example, let you work on a churn prediction model and you want your model to predict whether a customer of your organization's service will abandon the subscription within a six months period. You build a model and it's accurate enough, however, several weeks after deployment you see many false negatives, much more than expected. You investigate the reason of such a decreased model performance and discover that a new competitor to your organization now offers a very similar service for a lower price. This information wasn't available to your model in the form of a feature, therefore important information for accurate prediction was missing.

**Sponsorship or funding bias** affects the data produced by a sponsored agency. For example, let a famous video game company sponsor a news agency to produce news about the video game industry. If you try to make a prediction about the video game industry, you might include in your data the news produced by this sponsored agency. It often happens, however, that sponsored news agencies tend to suppress bad news about their sponsor and exaggerate the achievements of their sponsor's products. In this case, your model's performance will be biased because of that sponsorship bias in the data.

**Sampling bias** occurs when the distribution of the examples used to train the model doesn't reflect the distribution of the inputs the model will receive in the production environment. This type of bias is frequently observed in practice. For example, if you are working on a system that classifies documents according to a taxonomy of several hundred topics. You might decide to create a collection of documents, in which each topic is represented by an equal amount of documents. Once you finish working on the model, you observe the accuracy of 95% which is considered good for your purposes. However, soon after the deployment, you see that the wrong topic is assigned to about 30% of documents (and not to 5% as you expected). Why did this happen? One of the possible reasons is the sample bias: it might happen that one or two frequent topics in production data account for about 80% of all input documents. If your model doesn't perform well for these specific frequent topics, then your system will make many more errors in production than you initially expected.

**Prejudice or stereotype bias** is often observed in data obtained from historical sources, such as books or photo archives, or from the online activity of people, such as social media, online forums and comments to online publications. For example, if you use photos from a photo archive to train a model that distinguishes men from women. Historically, men were frequently taken on a picture in work or outdoor contexts, while women were more often captured at home and indoor. If you use such biased data, your model will have more difficulty in recognizing a woman outdoors and a man at home.

**Systematic value distortion** is a type of bias that most often occurs when there's a problem with the device making measurements or observations. This type of bias can result in a machine learning model making suboptimal predictions when deployed in the production environment. For example, the training data could be gathered using a camera with a particular setup of white balance which makes white look yellowish. In production, on the other hand, engineers might decide to put a higher quality camera which "sees" white as white. Because your model was trained on yellowish lower quality pictures, the predictions in production for higher-quality input will most likely be suboptimal. This problem is different from the presence of noise in the data. As I mentioned above, noise is considered to be a result of a random process that distorts the data. That means that when you have a sufficiently large dataset, noise can become less of a problem because it might average out, as I mentioned above. On the other hand, if the measurements are consistently skewed in one direction all the time, then it can damage data used for training the model, and ultimately generate a poor quality model.

Generally speaking, **confirmation bias** is the tendency to search for, interpret, favor, and recall information in a way that affirms one's prior beliefs or hypotheses. Applied to machine learning, confirmation bias occurs when each example in the dataset is obtained from the answers given by a particular person to a survey, one example per person. Assume that each survey contains multiple questions. The form of those questions can significantly affect the responses. The simplest way for a question to affect the response is to provide response options: "Which kind of pizza do you like: pepperoni, all meats, or vegetarian?". The above way to ask a question doesn't leave the choice of giving an answer different from the three given options. Alternatively, a survey question can be constructed with a particular slant. Instead of asking "Do you recycle?" the analyst with a confirmation bias could ask "Do you dodge from recycling?". In the former case, the person would rather give an honest answer, but less likely in the latter.

Confirmation bias often happens when the analyst was briefed in advance to support a particular conclusion (for example the one in favor of doing "business as usual"). In that situation, analysts can intentionally exclude particular variables from the analysis as unreliable or noisy, for example.

**Labeling bias** happens when labels are assigned to unlabeled examples by a biased process or person. For example, if you ask several people (labelers) to assign a topic to a document by reading the document, some labelers can indeed read the document entirely and assign well-thought labels, while others could just try to scan the text, spot some keyphrases and choose the topic that corresponds the best to the spotted keyphrases. Because each person's brain pays more attention to key phrases from a specific domain or domains and less to others, the labels assigned by labelers who scan the text without reading will be biased. Alternatively, some labelers would be more interested in reading documents on some topics that they personally prefer. In that case, they might skip uninteresting documents and the latter will be underrepresented in your data.

**3.4.4.2 Ways to Avoid Bias** It is usually impossible to know exactly what biases are present in a dataset. Furthermore, avoiding biases is a challenging task. First of all, be prepared not to be capable of avoiding them entirely. A good habit is to *question everything*: who has created the data you have access to, what were their motivation and quality criteria, and more importantly how and why the data was created. If the data is a result of some research, question the research method and make sure that that research method doesn't contribute to any of the biases described above.

**Selection bias** is can be avoided by systematically questioning the reason why a specific data source was chosen. If the reason is the simplicity to get the data or the low cost, then you have to pay careful attention to the coverage by that data source of the most important use cases of the model you try to build. If your model has to predict whether a specific customer will subscribe to your new offering, then building the model using *only* the data about your current customers is likely a bad idea, because your existing customers are more loyal to your brand than a random potential customer. Therefore your model will be overly optimistic.

**Self-selection bias** cannot be completely eliminated. Because it usually happens to survey-like data, the mere consent of the responder to answer the survey's question represents self-selection bias. Usually, the longer the survey, the fewer the chances that the respondent will agree to answer all the questions with a high degree of attention. Therefore, keep your survey short and give an incentive to the responders to give quality answers to the questions. Furthermore, pre-select responders to reduce self-selection. In our example of survey of successful entrepreneurs, don't ask the responder whether they consider themselves successful. Rather, build a list of successful entrepreneurs based on reference from experts or publications in literature, and then only contact people on that list.

It's very hard to completely avoid the **omitted variable bias**, because, as you know, *we don't know what we don't know.* One possible approach is to use all available information, that is to include into your feature vector as many features as possible, even those you deem unnecessary. This could theoretically make your feature vector very wide (i.e., many dimensions) and sparse (i.e, the values in most dimensions are zero) but if you use a well-tuned regularization, your model will "decide" which features are important and which ones aren't.

Alternatively, if you think that a specific variable is important and leaving it out of your regression model could result in an omitted variable bias, but at the same time you do not have data for it, you can try to use a proxy variable in lieu of the omitted variable. For instance, if you want to build a model that predicts the price of a used car and you cannot have the age of the car, you can use the time of ownership of the car by its current owner. The amount of time the car was owned by the current owner can be taken as a proxy for the age of a car.

**Sponsorship bias** can be reduced by carefully investigating the source of the data, more specifically the incentive for the owner of the source to provide the data. For example, it's known that publications on tobacco and pharmaceutical drugs are very often sponsored by either tobacco and pharmaceutical companies or their opponents. The same can be said about news companies, especially those that depend on the advertisement or have an undisclosed business model.

**Sampling bias** can be avoided by doing research on the real proportion of various properties in the data that will be observed in production and then sampling the data for training by keeping similar proportions.

**Prejudice or stereotype bias** can be controlled. For our example of distinguishing women from men on pictures, a data analyst could choose to under-sample the number of pictures of women indoor or oversample the number of men at home. In other words, prejudice or stereotype bias is reduced by exposing the learning algorithm to a more even-handed distribution of examples.

**Systematic value distortion** bias can be alleviated by having multiple measuring devices or hiring humans trained to compare the output of measuring or observing devices.

**Confirmation bias** can be avoided by letting multiple people validate the questions asked in the survey. The question to ask: "Do I feel uncomfortable or constrained answering this question?". Furthermore, despite more difficulties in analysis, prefer open-ended questions rather than yes/no or multiple-choice questions. If you still prefer to give responders a choice of answers, include the option "Other" and a place to write a different answer.

**Labeling bias** can be avoided by asking different labelers to label the same example. In case the labels assigned by different labelers to the same example are different, you can ask the labelers why they decided to assign a specific label to an example. If you see that some labelers refer to keyphrases rather than try to paraphrase the whole document, you can identify the labelers who are quickly scanning texts instead of reading them. You can also compare the frequency of skipped document for different labelers. If you see that some labeler skips document more often than the average, you could ask them why they skipped specific documents: is that because they have a technical problem or they are simply not interested in some topics.

You cannot completely avoid bias in data. There's no silver bullet. As a general rule, keep a human in the loop, especially if your model affects people's lives.

There is a temptation among the data analysts to assume that machine learning models are inherently fair because they make decisions based on evidence and math as opposed to messy human judgments. This is, unfortunately, not always the case: inevitably, a model trained on biased data will produce biased results. It is the duty of people building the model to ensure that its outputs are fair. But what's fair, you may ask? Unfortunately, again, there is no silver bullet measurement that would always detect unfairness. Choosing an appropriate definition of model fairness is always problem-specific and requires human judgment.

To keep a human in the loop in all stages of data gathering and preparation is the best approach to make sure that the damage caused by a statistical model is minimized.

### 3.4.5 Low Predictive Power

Low predictive power is that kind of problem which you don't see until you spend too much time fruitlessly trying to build a good model. The difficulty to see that problem is due to the fact that you typically don't know why your model doesn't perform well. Is it because the model is not expressive enough or the data doesn't contain enough information to learn from? You don't know.

Let your problem be to predict whether a user of a music streaming service will like a new song. Your data is the name of the artist, the title of the song and song lyrics, as well as whether the user has this song in their playlist. While such data may at first seem sufficient, the model you will build based on that data alone will be far from perfect.

First of all, the model will very unlikely score high songs from artists that are not in the user's playlist. Furthermore, many users of streaming services only add to their playlist *some* songs of a specific artist. The preferences of a music lover can be significantly influenced by the song arrangement, choice of instruments, sound effects, tone of voice, and subtle changes in tonality, rhythm, and beat. These are properties of the song that cannot be found in its lyrics, title and the name of the artist. They have to be extracted from the sound file. On the other hand, extracting relevant features from a sound file is challenging. Even with modern neural networks, recommending songs based on how they sound is still considered a hard task for AI. Typically, song recommendations are given by comparing playlists of different users and finding those that are similar in composition.

Another example of data with low predictive power is the photos of the sky made by a telescope. Let's say we want to build a model that would predict where to point the telescope to observe something interesting. Our data are photos of various regions of the sky where something interesting was observed in the past. You can imagine that based on the photos alone it's very unlikely that we will be able to train a model that accurately predicts that something interesting will happen. However, if we add to this data the measurements of various sensors, such as the sensors measuring radiofrequency of the signals coming from different parts of the sky or particle bursts, it is more likely that we will be able to make better predictions.

As I said above, defining that your data has low predictive power for your problem is very hard. It's especially hard if you work with that dataset for the first time. If you cannot obtain good model no matter how complex it becomes, I recommend engineering as many additional features as possible (apply your creativity!), especially by using indirect data sources to enrich feature vectors with additional information.

### 3.4.6 Outdated Examples

Once you build the model and deploy it in production, it usually performs well for some time. The length of this time depends entirely on the phenomenon you are modeling.

Typically, a certain model quality monitoring procedure is deployed in the production environment. Once an erratic behavior is detected, new training data is added to fix the behavior of the model; the model is then re-trained and re-deployed.

Often, the cause of an error is explained by the finiteness of the training set. In this case, adding additional training examples only solidifies the model. However, in many practical scenarios, the model can start to make errors because of so-called **concept drift**. Concept drift is a fundamental change in the properties of the phenomenon your data represents.

To be more specific, imagine that your model predicts whether a given user will like a given piece of content on a website. You have a history of content consumption for each user as well as some indicator of whether they liked the content or not. This indicator can be direct, such as a "like", or indirect, such as the time the user spent "consuming" the content. Once a user consumes some piece of content, you add a new example to your training data and periodically rebuild the model for that user.

During some time, the model improves with additional training examples and the predictions for each user improve too. However, at the same time, the preferences of some users change. This can happen because of aging, or because a user discovers something new with time and starts liking it more. (I didn't listen to jazz three years ago, now I do!) After the user's preferences change — we can also say, after the concept drift happens, — the examples added to the training data before the concept drift become outdated and actually start hurting the model performance rather than contributing to it.

You can detect concept drift by regularly building a new test set of recent examples from the production environment and comparing the performance of your model on the recent examples with the historical performance. If you see a decreasing trend in performance on new data, it's an indicator of the concept drift.

In order to remove the outdated examples from the training data, you can first sort your training examples in the order of recency and define an additional hyperparameter — the percentage of the most recent examples to use for rebuilding the model. You can use **grid**

**search** or another hyperparameter tuning technique to find the best value for that additional hyperparameter by maximizing the performance of the model on a sufficiently representative test set composed of recent examples from the production environment. (We will consider hyperparameter tuning in Chapter 5.)

### 3.4.7 Outliers

Outliers are examples that look dissimilar to a "typical" example from the dataset. It's up to the data analyst to define "dissimilar". Typically, dissimilarity is measured by some distance metric, such as Euclidean distance. In that case, an outlier is an example that lies at an abnormal distance from other examples in the dataset.

In many practical situations, however, what seems to be an outlier in the original feature vector space can be a typical example in a feature vector space transformed using a **kernel function**. Feature space transformation is often explicitly done by a kernel-based algorithm, such as **support vector machine** (SVM), or implicitly by a deep neural network.

Shallow algorithms, such as linear or logistic regression, and some ensemble methods, such as AdaBoost, are particularly sensitive to outliers. SVM has one definition that is less sensitive to outliers: a special penalty hyperparameter regulates the influence of misclassified examples (which often happen to be outliers) on the decision boundary. If this penalty value is low, the SVM algorithm may completely ignore outliers from consideration when drawing the decision boundary. Though, if it's too low, even some normal examples can end up on the wrong side of the decision boundary. The best value for that hyperparameter has to be found using a hyperparameter tuning technique.

A sufficiently complex neural network can learn to behave differently for each outlier in the dataset and, at the same time, still work well for the normal examples, though it's not the desired outcome because the model is unnecessarily complex for the task. More complexity can result in longer training and prediction time and poorer generalization in the production environment.

Whether to exclude outliers from the training data or to use machine learning algorithms robust to outliers is a debated question. Deletion of examples from a dataset is not considered scientifically or methodologically sound, especially in small datasets. In the big data context, on the other hand, outliers don't typically have a significant influence on the model.

From a practical standpoint, if excluding some training examples results in better performance of the model on the holdout data, the exclusion may be justified. Which examples to consider for exclusion can be decided based on a certain similarity measure. A modern approach to getting such a measure is to build an **autoencoder** and use the reconstruction error as the measure of (dis)similarity: the higher the reconstruction error for a given example, the more dissimilar it is to the dataset. The hyperparameters of the autoencoder are tuned to minimize the reconstruction error of the holdout data.

### 3.4.8 Leakage

**Data leakage**, also called **target leakage**, is a problem that affects several stages of the machine learning life cycle, from data collection to model evaluation. In this chapter, I will only describe how this problem manifests itself at the data collection and preparation stage. In the subsequent chapters, I will talk about other forms of the problem of data leakage.

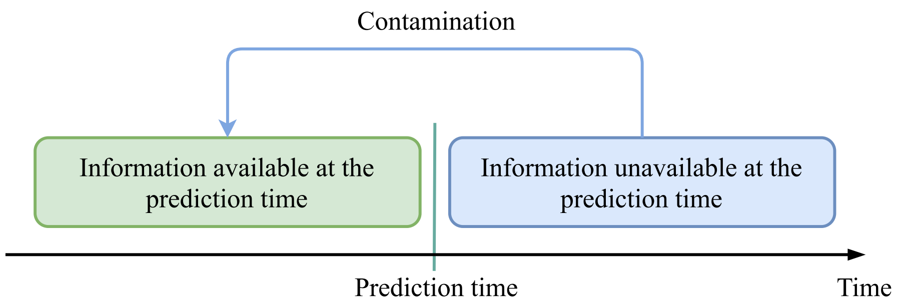

Contamination
Information available at the prediction time
Information unavailable at the prediction time
Prediction time
Time

Figure 7: Data leakage in a nutshell.

Data leakage in supervised learning is an unintentional introduction of information about the target which should not be available to learn from. It manifests itself in the emergence of unexpected additional information in the training data due to the so-called "contamination" (Figure 7). Training on contaminated data leads to overly optimistic expectations about the model performance in production.

The contamination of the training data can happen at one or several stages of the machine learning life cycle including the stage of data collection and preparation, which we focus on in this chapter. Below, I consider three typical causes of contamination that can happen at that stage.

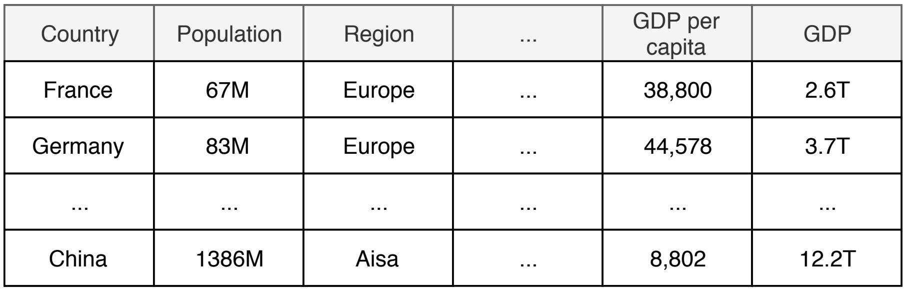
## 3.5 Causes of Data Leakage

Let's discuss the three most frequent causes of data leakage at the stage of data collection and preparation.

### 3.5.1 Target is a Function of a Feature

Let our goal be to predict the GDP of the country based on various attributes describing that country: area, population, geographic region, and so on. An example of such data is shown in Figure 8. If you don't do a careful analysis of each attribute and how it's related to GDP you might let a leakage happen: in the data in Figure 8, two columns, Population and GDP per capita, multiplied give GDP. The model you will build will most likely perfectly predict

GDP by looking at these two columns only. The fact that you let GDP be a part of features though in a slightly modified form (devised by the population) constitutes contamination and is data leakage.



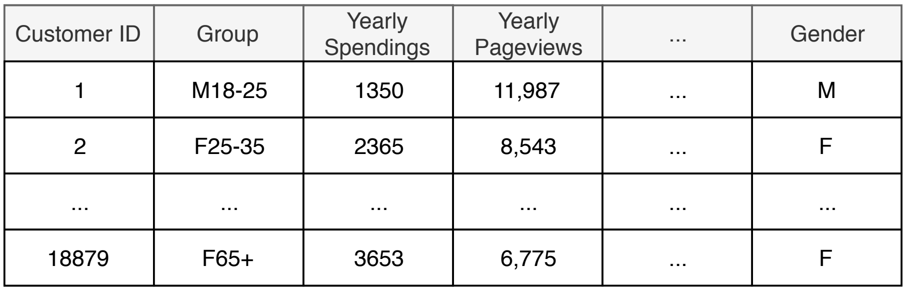
| Country | Population | Region | ... | GDP per capita | GDP |
|---|---|---|---|---|---|
| France | 67M | Europe | ... | 38,800 | 2.6T |
| Germany | 83M | Europe | ... | 44,578 | 3.7T |
| ... | ... | ... | ... | ... | ... |
| China | 1386M | Aisa | ... | 8,802 | 12.2T |

Figure 8: An example of the target being a simple function of two features: Population and GDP per capita.

A simpler example of the same type of data leakage is when you have the target among features; it's just given in a different format. For example, imagine that you build a model to predict the yearly salary given an employer's attributes. The data you use is a table that contains both monthly and yearly salary among many other attributes of an employee. If you forget to remove the monthly salary from the list of features, this attribute alone will perfectly predict the yearly salary making you believe that your model is perfect until, in production, it doesn't receive information about a person's monthly salary, which is likely: otherwise the modeling would not be needed.

### 3.5.2 Feature Hides the Target

Sometimes the target is not a function of one or more features but rather is "hidden" in one of the features. Consider the dataset shown in Figure 9.



| Customer ID | Group | Yearly Spendings | Yearly Pageviews | ... | Gender |
|---|---|---|---|---|---|
| 1 | M18-25 | 1350 | 11,987 | ... | M |
| 2 | F25-35 | 2365 | 8,543 | ... | F |
| ... | ... | ... | ... | ... | ... |
| 18879 | F65+ | 3653 | 6,775 | ... | F |

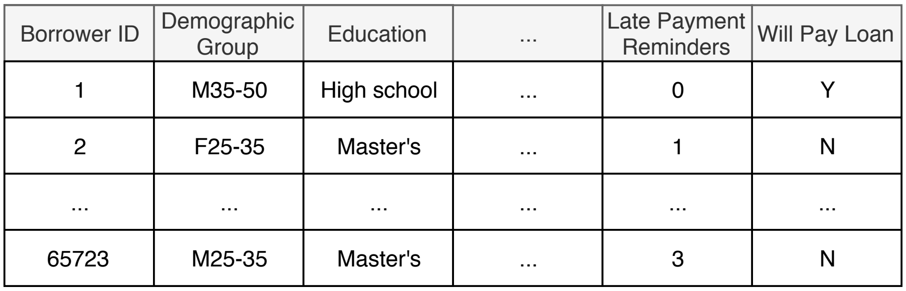
Figure 9: An example of the target being hidden in one of the features.

In this scenario, you use the data about a customer to predict their gender. Look at the column Group. If you closely investigate the data in this column, you will see that Group represents a demographic group to which each existing customer was related in the past. If the data about a customer's gender and age is factual (as opposed to being guessed by another model that might be available in production) then the column Group constitutes a form of data leakage, when the value you want to predict is "hidden" in the value of a feature.

If values in the column Group are predictions provided by another, possibly less accurate, model then you can use this attribute to build a possibly stronger model. This is called **model ensembling** and we will consider this topic in Chapter 5.

### 3.5.3 Feature From the Future

"Feature from the future" is a kind of data leak that is hard to catch if you don't have a clear understanding of the business goal. Imagine a client asked you to build a model that predicts whether a borrower will pay the loan based on the borrower's attributes such as age, gender, education, salary, marital status, number of children, and so on. An example of such data is shown in Figure 10.



| Borrower ID | Demographic Group | Education | ... | Late Payment Reminders | Will Pay Loan |
|---|---|---|---|---|---|
| 1 | M35-50 | High school | ... | 0 | Y |
| 2 | F25-35 | Master's | ... | 1 | N |
| ... | ... | ... | ... | ... | ... |
| 65723 | M25-35 | Master's | ... | 3 | N |

Figure 10: An example of a feature unavailable at the prediction time: Late Payment Reminders.

If you don't make an effort to understand the business context, in which your model will be used, you might decide to use all available attributes to predict the value in the column Will Pay Loan, including the data from the column Late Payment Reminders. Your model will most likely work very well and you will send it to the client who will later report that the model doesn't work very well in the production environment.

After close investigation, you will find out that in the production environment, the value of the feature that you created based on the column Late Payment Reminders is always zero. This makes sense because the client uses your model *before* the borrower gets the credit, so no reminders have yet been made to the borrower! However, your model most likely learned to make the negative prediction once the value of this feature is positive and pays less attention to the values of the other features.

Understanding the business context, in which the model will be used is, thus, very important to avoid this type of data leakage.

## 3.6 Data Partitioning

As you already know from Chapter 1, in practical machine learning, we typically use three disjoint sets of examples: training set, validation set, and test set.

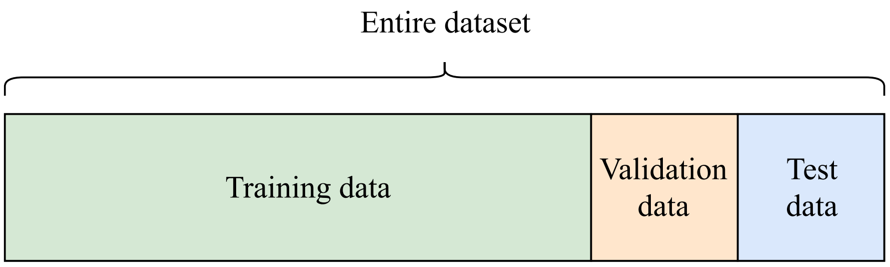


Figure 11: The entire dataset partitioned into a training, validation and test sets.

The **training set** is used by the machine learning algorithm to build the model.

The **validation set** is needed to find the best values for the hyperparameters of the machine learning system. The analyst tries different combinations of hyperparameters one by one, builds a model by using each combination and then selects the model whose performance is the best when applied to the validation set

The **test set** is used for reporting: once you have your best model, you test its performance on the test set and report the results.

Validation and test set are often referred to as **holdout sets**: they contain the examples that the learning algorithm is not allowed to see.

To obtain a good partition of your entire dataset into these three disjoint sets, as shown in Figure 11, the process of partitioning has to satisfy several conditions discussed below.

**Condition 1: Split was applied to raw data.**
Once you have access to raw examples, and before everything else, do the split. This will allow avoiding data leakage, as I will show below.

**Condition 2: Data was randomized before split.**
Randomly shuffle your examples first, then do the split.

**Condition 3: Validation and test sets follow the same distribution.**
In some cases, as we will discuss in Chapter 5, you might prefer to use different datasets for training and test: one — bigger and more easily available — for training and another — smaller but more similar to the production data — for test. In this situation, shuffle all available data for test, and then split it into validation and test sets.

**Condition 4: Leakage was avoided.**
Data leakage can happen even during the data partitioning. Below, I show what form of leakage can happen during that stage.

There is no ideal ratio for the split. In older literature (pre-big data), you might find either 70%/15%/15% or 80%/10%/10% recommended splits (numbers refer to the sizes of training, validation, and test set respectively given in proportion to the entire dataset). On the other hand, today, in the era of the Internet and cheap labor available on such online platforms as Mechanical Turk or via crowdsourcing, organizations, scientists and even enthusiasts at home can get access to millions of training examples. In that latter case, it would be wasteful to only use 70% or 80% of the available data for training.

Indeed, the validation and test data is only used to calculate statistics reflecting the performance of the model. These two sets just need to be big enough to provide *reliable statistics*. How much is big enough, is debatable. As a rule of thumb, having a dozen examples per class is a desirable minimum. If you can have a hundred examples per class in each of the two holdout sets, you can typically consider that you have a solid setup and the statistics calculated based on such sets are reliable.

The percentage of the split can also be dependent depending on the chosen machine learning algorithm or model. For instance, it is well known that deep learning models tend to improve to a greater extent when provided more training data while shallow algorithms and models may not. Also, your choice for the split may depend size of the dataset in your possession. For example, if you have a small dataset (less than a thousand examples) you would want to use 90% of the data for training trying to produce the best model possible. In the latter case, you might decide to not have a validation set and simulate it using the **cross-validation** technique. We will talk more about that in Chapter 5.

It's worth mentining that when you split **time series data** into the three datasets you have to execute the split in such a way that the order of observations in each example is preserved and not shuffled. Otherwise, for most predictive problems your data will be broken and no learning will be possible.

### 3.6.1 Leakage During Partitioning

As I said above, data leakage can happen at any stage from data collection to model evaluation. The stage of data partitioning is no exception.

One form of leakage that can happen during the partitioning of the data is called **group leakage**. Imagine the following problem: you have magnetic resonance images of the brain of multiple patients. Each image is labeled with certain brain disease. One particularity of this dataset is that the same patient can be represented by several images taken in different moments. If you apply the partitioning technique discussed above (i.e., shuffle then split) then images of the same patient could get into both training and holdout data. This can result in the following undesirable property of the model: the model could learn from the particularities of the patient rather than of the disease. For example, the model can remember that patient A's brain has a specific form of brain convolutions, and if patient A has a specific disease in the training data, then the model will successfully predict this disease in the validation data if it recognizes patient A from the brain convolutions.

The solution to group leakage is **group partitioning**. It consists of keeping all examples from one group (i.e., one patient in our illustrative example discussed above) in one set: either training or holdout. Again, you can see how important for the data analyst is to know as much as possible about the data they work with.

## 3.7 Dealing with Missing Attributes

In some cases, the data comes to the analyst in the tidy form, such as an Excel spreadsheet[2]. In some examples, the values of some attributes can be missing. That often happens when the dataset was handcrafted, and the person working on it, for example, forgot to fill some values or didn't get them measured at all.

The list of typical approaches of dealing with missing values for an attribute include:

- removing the examples with missing attributes from the dataset (that can be done if your dataset is big enough so you can safely sacrifice some data);
- using a learning algorithm that can deal with missing values of attributes (such as the decision tree learning algorithm);
- using a **data imputation** technique.

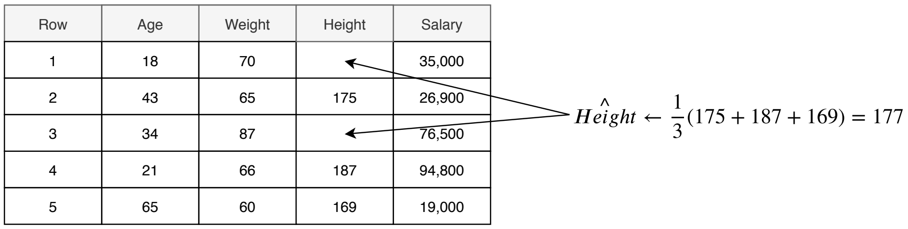
### 3.7.1 Data Imputation Techniques

To impute the value of a missing numerical attribute, one technique consists in replacing the missing value by an average value of this attribute in the dataset. Mathematiclly it looks as follows. Let $j$ be an attribute that is missing in some examples in the original dataset, and let $\mathcal{S}^{(j)}$ be the set of size $N^{(j)}$ that contains only those examples from the original dataset in which the value of the attribute $j$ is present. Then the missing value $\hat{x}^{(j)}$ of the attribute $j$ is given by,

$$\hat{x}^{(j)} \leftarrow \frac{1}{N^{(j)}} \sum_{i \in \mathcal{S}^{(j)}} x_i^{(j)},$$

[2]The fact that your raw dataset is contained in an Excel spreadsheet doesn't guarantee that the data is tidy. One property of tidiness is that each row represents one example.

where $N^{(j)} < N$ and the summation is made only over those examples, in which the value of the attribute $j$ is present. An illustration of this technique is given in Figure 12, where two examples (at row 1 and 3) have the Height attribute missing.



| Row | Age | Weight | Height | Salary |
|---|---|---|---|---|
| 1 | 18 | 70 |  | 35,000 |
| 2 | 43 | 65 | 175 | 26,900 |
| 3 | 34 | 87 |  | 76,500 |
| 4 | 21 | 66 | 187 | 94,800 |
| 5 | 65 | 60 | 169 | 19,000 |

$$\widehat{Height} \leftarrow \frac{1}{3}(175 + 187 + 169) = 177$$

Figure 12: Replacing the missing value by an average value of this attribute in the dataset.

Another technique is to replace the missing value with a value outside the normal range of values. For example, if the normal range is $[0, 1]$, then you can set the missing value to 2 or $-1$; or, if the attribute is categorical, such as days of the week, then a missing value can be replaced by the value "Unknown". The idea is that the learning algorithm will learn what is best to do when the attribute has a value different from regular values. If the attribute is numerical, an alternative technique consists of replacing the missing value by a value in the middle of the range. For example, if the range for an attribute is $[-1, 1]$, you can set the missing value to be equal to 0. Here, the idea is that the value in the middle of the range will not significantly affect the prediction.

A more advanced technique is to use the missing value as the target variable for a regression problem. (In this case, we assume that all attributes are numerical.) You can use all remaining attributes $[x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(j-1)}, x_i^{(j+1)}, \ldots, x_i^{(D)}]$ to form a feature vector $\hat{\mathbf{x}}_i$, set $\hat{y}_i \leftarrow x^{(j)}$, where $j$ is the attribute with a missing value. Then you build a regression model to predict $\hat{y}$ from $\hat{\mathbf{x}}$. Of course, to build training examples $(\hat{\mathbf{x}}, \hat{y})$, you only use those examples from the original dataset, in which the value of attribute $j$ is present.

Finally, if you have a significantly large dataset and just a few attributes with missing values, you can add an additional attribute to your dataset, a binary indicator attribute for each attribute with missing values. Let's say that examples in your dataset are $D$-dimensional and attribute at position $j = 12$ has missing values. For each example $\mathbf{x}$, you then add the attribute at position $j = D + 1$ which is equal to 1 if the value of the attribute at position 12 is present in $\mathbf{x}$ and 0 otherwise. The missing value then can be replaced by 0 (or a different number if the attribute is numerical) or any value of your choice.

At prediction time, if your example is not complete, you should use the same data imputation technique to fill the missing values as the technique you used to complete the training data.

Before you start working on the learning problem, you cannot tell which data imputation technique will work the best. Try several techniques, build several models and select the one that works the best (using the validation set to compare models).

### 3.7.2 Leakage During Imputation

If you use the imputation techniques that compute some statistic of one attribute (by finding the average, and out-of-range or mid-range value) or several attributes (by solving the regression problem) the leakage happens if you use the whole dataset to compute this statistic. By using all available examples, you "communicate" to the training data some information obtained from the validation and test examples.

This constitutes a contamination of the training data and, thus, leakage. This type of leakage is not as significant as other types discussed earlier, but you still have to be aware of it and avoid it by partitioning first and only then computing the imputation statistic on the training set only.

## 3.8 Data Augmentation

For some types of data, it's quite easy to get more labeled examples without additional labeling. The strategy is called **data augmentation** and it's the most effective when applied to images. It consists of applying simple operations, such as crop or flip, to the original images to obtain new images.

### 3.8.1 Data Augmentation for Images

In Figure 13, you can see examples of operations that can be easily applied to a given image to obtain one or more new images: flip, rotation, crop, color shift, noise addition, perspective change, contrast change, and information loss.

Flipping, of course, has to be done only with respect to the axis for which the meaning of the image is preserved. If it's a soccer ball, you can flip with respect to both axes[ˆch3_5], but if it's a car or a pedestrian, then you should only flip with respect to the vertical axis.

[ˆch3_5] Unless the context, like grass, makes flipping according to the horizontal axis irrelevant.

Rotation should be applied with slight angles to simulate an incorrect horizon calibration. You can rotate an image in both directions.

Crop can be randomly applied multiple times to the same image by keeping a significant part of the object (or objects) of interest in the cropped images.

In color shift, nuances of RGB are slightly changed to simulate the effect that different lighting conditions can have on a picture taken with a photo camera. This effect can be applied multiple times to the same image to simulate multiple lighting conditions. Contrast change and noise addition can also be applied multiple times to the same image, by changing the degree of contrast (both decreasing and increasing) and the intensity of the Gaussian noise.

By randomly removing parts of image we can simulate situations when an object is recognizable but not entirely visible because of some visual obstacle.

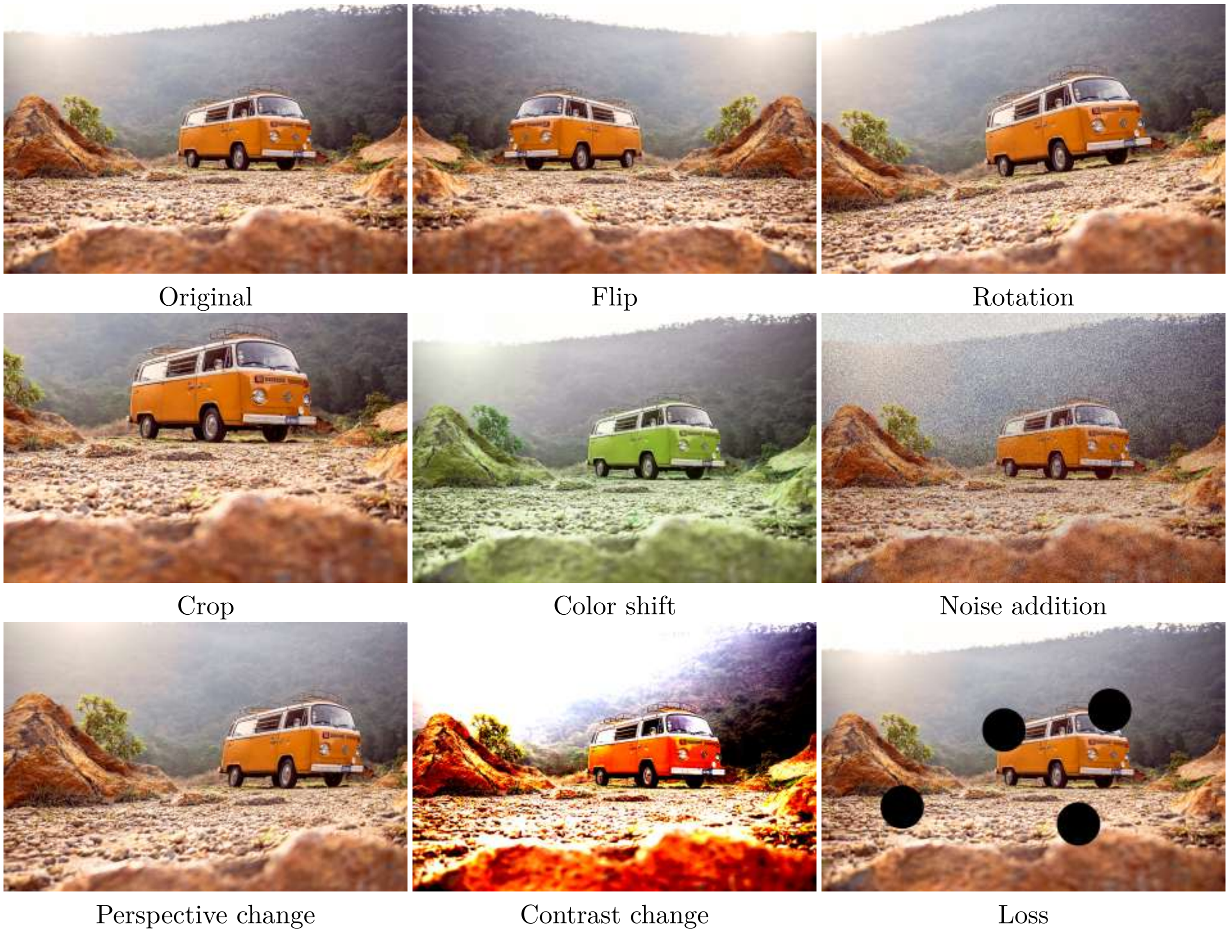


Figure 13: Examples of data augmentation techniques. Photo credit: Alfonso Escalante.

In addition to the techniques shown in Figure 13, if you expect that the input images in your production system can come overcompressed, you can simulate the effect of overcompression by using some frequently used lossy compression methods and file formats, such as JPEG or GIF.

Only training data undergoes augmentation. Of course, it's impractical to generate all these additional examples in advance and store them. In practice, the data augmentation techniques are applied to the original data on-the-fly during training.

### 3.8.2 Data Augmentation for Text

When it comes to text data augmentations, it is not as straightforward to find appropriate transformation techniques that would preserve the contextual and grammatical structure of natural language texts.

One technique consists of replacing random words in a sentence with their exact or very close synonyms. For example, we can obtain several equivalent sentences from the following one: *The car stopped near a shopping mall.* Some examples are:

- The automobile stopped near a shopping mall.
- The car stopped near a shopping center.
- The auto stopped near a mall.

A similar technique consists of using **hypernyms** instead of synonyms. A hypernym is a word that has more general meaning. For example, "mammal" is a hypernym for "whale" and "cat"; "vehicle" is a hypernym for "car" and "bus". From our example sentence above, by using hypernyms, we could obtain the following sentences:

- The vehicle stopped near a shopping mall.
- The car stopped near a building.

If you represent words or documents in your dataset using word or document embeddings, you can apply slight Gaussian noise to randomly chosen features of an embedding to hopefully obtain a variation of the same word or document. You can tune the number of features to modify and the intensity of noise as hyperparameters by using the validation data.

Alternatively, to replace a given word $w$ in the sentence, you can find $k$ nearest neighbors to the word $w$ in the embedding space and generate $k$ new sentences by replacing the word $w$ by its respective neighbor. The nearest neighbors can be found using a metric such as **cosine similarity** or **Euclidean distance**. The choice of the metric, as well as the value of $k$, can be tuned as hyperparameters.

Similarly, if your problem is document classification and you have a large corpus of unlabeled documents and only a small corpus of labeled documents, you can do as follows. First, build **document embeddings** for all documents in your corpus. To do that you can use **doc2vec** or any other technique of document embedding. Then, for each labeled document $d$ in your dataset, find $k$ closest unlabeled documents in the embedding space and label them with the same label as $d$. Again, tune $k$ on the validation data.

Another text data augmentation technique that works well is **back translation**. To obtain a new example from sentence text $t$ in English (it can be a sentence or a document), you translate it into language $l$ using a machine translation system and then translate it back from $l$ into English. If the text obtained by back translation is different from the original text, you add it to the dataset by assigning the same label as the label of the original text.

There are also techniques of data augmentation for other types of data, such as audio and video. For example, noise addition, shifting an audio or a video clip in time, slowing it down or accelerating, changing pitch for audio and color balance for video, and so on. Describing these techniques in detail is out of the scope of this book. You should just be aware that data augmentation can be applied to any media data, and not just images and text.

## 3.9 Dealing With Imbalanced Data

**Class imbalance** is a condition in the data that can significantly affect the performance of the model, independently of the chosen learning algorithm. The problem is a very uneven distribution of labels in the training data[3].

This is the case, for example, when your classifier has to distinguish between genuine and fraudulent e-commerce transactions: the examples of genuine transactions are much more frequent. Typically, a machine learning algorithm tries to classify most training examples correctly. The algorithm is pushed to do so because it needs to minimize a cost function that typically assigns a positive loss value to each misclassified example. If the loss is the same for misclassification of an example from the minority class as it is for the misclassification of an example from a majority class then it's very likely that the learning algorithm decides to "give up" on many minority class examples in order to make fewer mistakes on the examples of the majority class.

### 3.9.1 Class Weighting

Some algorithms, such as support vector machines, allow the data analyst to provide weights for each class. The loss in the cost function is typically multiplied by the weight; the data analyst can, for example, provide higher weight to the minority class and, by doing so, to make it harder for the learning algorithm to give up on the examples of the minority class, because it would result in much higher cost than without class weighting.

### 3.9.2 Oversampling

If a learning algorithm doesn't allow weighting classes, you can try the technique of **oversampling**. It consists of increasing the importance of examples of some class by making multiple copies of the examples of that class (Figure 14a).

[3]While there is no formal definition imbalanced data, consider the following rule of thumb. If there are two classes, then balanced data would mean the half of the dataset represents each class. A slight class imbalance is usually not a problem. So, if 60% examples belong to one class and 40% belong to the other class and you use a popular machine learning algorithm in its standard formulation, it should not cause any significant performance degradation. However, when the class imbalance is high, for example when 90% examples are of one class and 10% are of the other class, using the standard formulation of the learning algorithm that usually equally weights errors made in both classes may not be as effective and would need modification.

Oversampling can be performed by making exact copies of the existing examples or by generating synthetic examples by combining existing examples of the minority class together in a certain way.

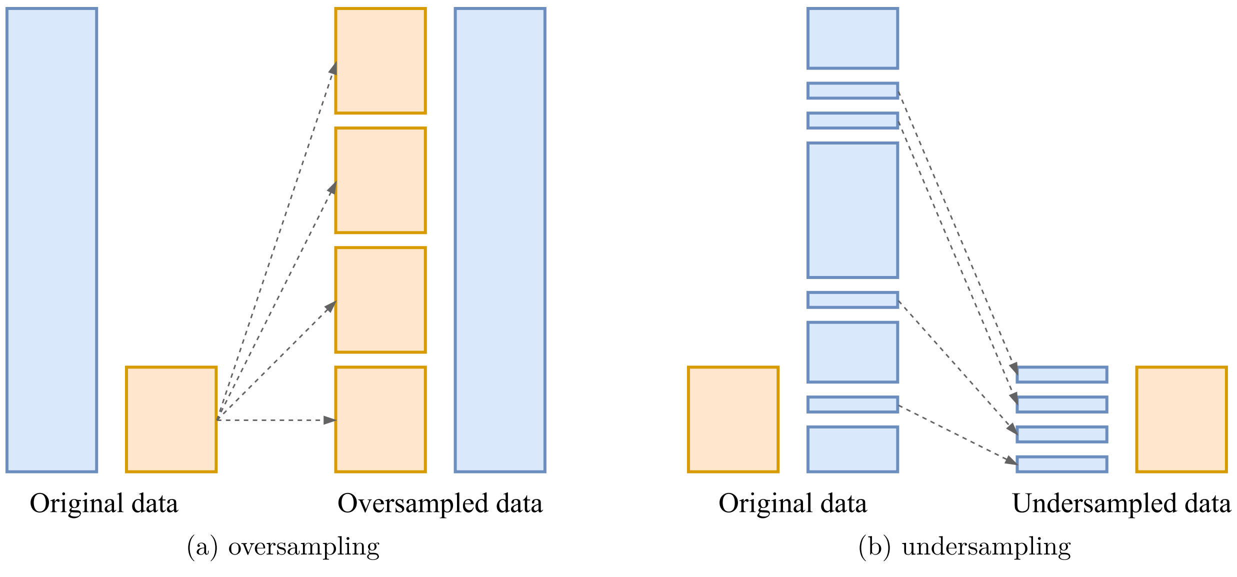
There are two popular algorithms that oversample the minority class by creating synthetic examples: **SMOTE** (for synthetic minority oversampling technique) and **ADASYN** (for adaptive synthetic sampling method).

SMOTE and ADASYN work similarly in many ways. For a given example $\mathbf{x}_i$ of the minority class, they pick $k$ nearest neighbors of this example (let's denote this set of $k$ examples $\mathcal{S}_k$) and then create a synthetic example $\mathbf{x}_{new}$ as $\mathbf{x}_i + \lambda(\mathbf{x}_{zi} - \mathbf{x}_i)$, where $\mathbf{x}_{zi}$ is an example of the minority class chosen randomly from $\mathcal{S}_k$. The interpolation hyperparameter $\lambda$ is a random number in the range $[0, 1]$.

Both SMOTE and ADASYN randomly pick all possible $\mathbf{x}_i$ in the dataset. In ADASYN, the number of synthetic examples generated for each $\mathbf{x}_i$ is proportional to the number of examples in $\mathcal{S}_k$ which are not from the minority class. Therefore, more synthetic examples are generated in the area where the examples of the minority class are rare.

### 3.9.3 Undersampling

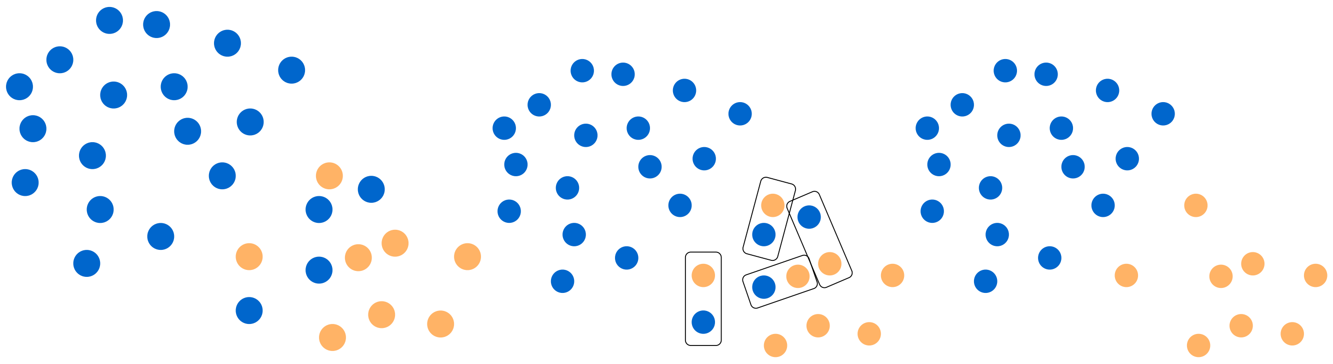
An opposite approach, **undersampling**, is to remove from the training set some examples of the majority class (Figure 14b).



(a) oversampling (b) undersampling

Figure 14: Undersampling and oversampling.

The undersampling can be done randomly, that is the examples to remove from the majority class can be chosen at random. Alternatively, examples to remove from the majority class can be selected based on some property. One such property is **Tomek links**. A Tomek link exists between two examples $\mathbf{x}_i$ and $\mathbf{x}_j$ belonging to two different classes if there's no other example $\mathbf{x}_k$ in the dataset closer to either $\mathbf{x}_i$ or $\mathbf{x}_j$ than the latter two are to each other. The closeness can be defined using a metric such as **cosine similarity** or **Euclidean distance**.



(a) original data (b) Tomek links (c) undersampled data

Figure 15: Undersampling with Tomek links.

In Figure 15, you can see how removing examples from the majority class based on Tomek links helps to establish a wider margin between examples of two classes. A wide margin makes it simpler to classify the data and helps generalization.

**Cluster-based undersampling** works as follows. Decide on the number of examples you want to have in the majority class as the result of undersampling. Let that number be $k$. Run a centroid-based clustering algorithm *on the majority examples only* with $k$ being the desired number of clusters. Then replace all examples in the majority classes with the $k$ centroids. An example of a centroid-based clustering algorithm is **k Nearest Neighbors**.

### 3.9.4 Hybrid Strategies

You can develop your own hybrid strategies (by combining both over- and undersampling) and possibly get better results. One such strategy consists of using SMOTE or ADASYN to oversample and then Tomek links to undersample.

Another possible strategy consists of combining cluster-based undersampling with SMOTE or ADASYN.

## 3.10 Data Sampling Strategies

When you have a large data asset, so-called big data, it's not always practical or necessary to work with the entire data asset. You would rather prefer to draw a smaller sample of data that, despite being much smaller, contains enough information for learning.

Similarly, when you try to solve the data imbalance problem by undersampling the majority class, you would prefer that the smaller sample of the data was representative of the entire data from the majority class. In this section, we discuss several sampling strategies, their properties, advantages, and drawbacks.

Sampling strategies can be of two families: probability sampling and nonprobability sampling (Figure 16). In probability sampling, all examples have a chance to be selected to be included in a sample. These techniques involve randomness.

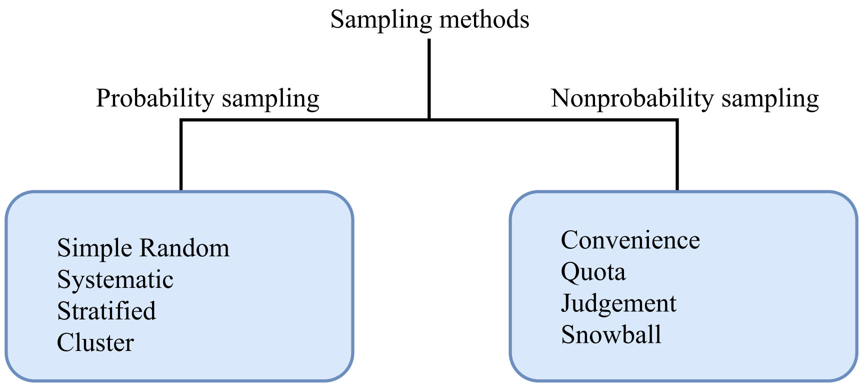


Figure 16: Sampling methods.

Nonprobability sampling is not random. To build a sample, it follows a fixed deterministic sequence of heuristic actions. This means that some examples don't have a chance of being selected, no matter how many samples you build.

Nonprobability sampling methods are more easy to execute manually by a human, though. However, this advantage is not significant for a data analyst working on a computer and using a software or programming code that greatly simplifies sampling of examples even from a very large data asset. The main drawback of nonprobability sampling techniques is that they provide non-representative samples and might systematically exclude important examples from consideration. This drawback outweighs the possible advantages of nonprobability sampling methods. Therefore, in this book I will only present the probability sampling techniques.

### 3.10.1 Simple Random Sampling

**Simple random sampling** is the most simple strategy and the one I refer to when I say "sample randomly". Here, each example from the entire dataset is chosen purely by chance; each example has an equal chance, or probability, of being selected.

One way of obtaining a simple random sample is to assign to each example a number, and then use a random number generator to decide which examples to select. For example, if your entire dataset contains 1000 examples, tagged from 0 to 999, use groups of three digits from the random number generator to select your example. So, if the first three numbers from the random number generator were 0, 5 and 7, select the example numbered 57, and so on.

The advantage of this sampling method is that it's simple. Any programming language has a random number generator, so you can implement this method easily. A disadvantage of simple random sampling is that you may not select enough examples that would have a certain property of interest. If, for example you sample from a large collection of examples, but some of those examples belong to a minority class, there are chances that in your smaller sample will not contain any example of the minority class.

### 3.10.2 Systematic Sampling

To implement **systematic sampling** (also known as **interval sampling**), you create a list containing all examples. From that list, you randomly select the first example from the first $k$ elements on the list. Then, you select every $k^{\text{th}}$ element on the list. You choose such a value of $k$ that will give you a sample of the desired size.

One advantage of the systematic sampling over the simple random sampling is that it works faster. It also draws examples from the whole range of values, while the simple random sampling, as we discussed above, may result in examples with some specific properties underrepresented in the sample. However, systematic sampling is inappropriate if the list of examples has periodicity or some kind of repetitive pattern. In that latter case, the obtained sample can exhibit a bias. However, if the list of examples is randomized, then systematic sampling often results in a better sample compared to simple random sampling.

### 3.10.3 Stratified Sampling

If you know about the existence of several groups of examples in your data, you would like to have, in your sample, examples from each of those groups. For example, in a dataset of people, groups can be defined by gender, location, or age. In **stratified sampling**, you first divide your dataset into groups (called strata) and then randomly (like in simple random sampling) select examples from each stratum. The number of examples to select from each stratum is proportional to the size of the stratum.

Stratified sampling often improves the representativeness of the sample by reducing its bias; in the worst case, it gives the sample no worse than when using simple random sampling. However, to define strata, the analyst has to have a good understanding of the properties of the dataset. Furthermore, it can be difficult to decide based on which attributes to define strata.

Stratified sampling is the slowest of the three methods due to the additional overhead of working with several independent strata. However, its potential benefit of producing a less biased sample typically outweighs its drawbacks.

### 3.10.4 Cluster Sampling

In **cluster sampling**, the whole dataset is first partitioned into distinct clusters. Then a number of clusters are randomly selected and all examples from the selected clusters are then added to the sample. This is different from the stratified sampling where examples are selected from each stratum; in cluster sampling, if a cluster was not selected, none of the examples from this cluster will get to the sample.

This technique improves the representativeness of the sample if each cluster is representative of the whole dataset. For instance, let examples in your dataset come from a variety of time periods. You can put in one cluster examples from the same time period. In this case, by applying cluster sampling you include all examples from selected time periods into the sample. Alternatively, an airline company can decide to first randomly choose planes, and then add all passengers from the selected planes to the sample.

The main drawback of this sampling method is similar to that of the stratified sampling: to define clusters that each are representative of the entire dataset, the analyst has to have a good understanding of the properties of the dataset.

## 3.11 Storing Data

Keeping data safe is insurance for your organization's business: if you lose a business-critical model for any reason, such as a disaster or human mistake (the file with the model was accidentally erased or overwritten), having the data will allow you to rebuild that model easily.

Keeping data safe becomes critical if your model has to be frequently updated. While in some cases, an existing model can be upskilled (additionally trained) by using only the new training examples, a best practice is to retrain the model from scratch every time by using the old and the new training examples together.

In many cases, especially when you work with the data about people or sensitive data provided to you by your customers or business partners, the data has to be stored in not just a safe but also a secure location. Jointly with the DBA or DevOps engineers, access to such sensitive data can be restricted by username and, if needed, IP address. Access to some data stored in a relational database might also be limited on the per row and per column basis.

It's also recommended to limit the access to data to read and add operations only (by restricting write and erase operations to specific users only).

If the data is collected on mobile devices, it might be necessary to store it on the mobile device for some time until the owner of the mobile device connects to wifi. This data might need to be encrypted so that other applications could not access it. Once the user is connected to wifi, the data has to be synchronized with a secure server by using cryptographic protocols, such as **Transport Layer Security** (TLS). Each data element on the mobile device has to be marked with a timestamp to allow its proper synchronization with the data on the server.

### 3.11.1 Data Formats

Data for machine learning can be stored in various formats. Data used indirectly, such as dictionaries or gazetteers are usually stored as a table in a relational database, a collection in a key-value store, or a structured text file, typically a file of comma-separated values (CSV) or tab-separated values (TSV). The tidy data is usually stored in a CSV or TSV file (all examples in one file) or a collection of XML or JSON files (one example per file).

In addition to general-purpose formats, certain popular machine learning packages use proprietary data formats to store tidy data. Other machine learning packages often provide APIs to one or several such proprietary data formats. The most frequently supported formats are ARFF (Attribute-Relation File Format used in the Weka machine learning package) and the LIBSVM format, which is the default format used by the LIBSVM and LIBLINEAR packages. The data in the LIBSVM format consists of one file containing all examples. Each line of that file represents a labeled feature vector using the following format:

```
label index1:value1 index2:value2 ...
```

where index$X$:value$Y$ specifies the value $Y$ of the feature at position $X$. If the value at some position is zero, it can be omitted. This data format is especially convenient for **sparse data**, which is the data that consists of examples in which the values of most features in the feature vectors are zero.

Furthermore, different programming languages come with **data serialization** capabilities. The data ready to be used by a specific machine learning package can be persisted on the hard drive by using a serialization object or function provided by the programming language or a library. When needed, the data can be deserialized in its original form. For example, in Python, a popular general-purpose serialization modeule is **pickle**. Additionally, different data analysis packages can offer their own serialization/deserialization tools.

In Java, any object that implements the *java.io.Serializable* interface can be serialized into a file and then deserialized when needed.

### 3.11.2 Data Storage Levels

Before deciding how and where to store the data, it's important to choose the appropriate **storage level**. Storage can be organized in different levels of abstraction: from the lowest level, the filesystem, to the highest level, such as data lake.

**Filesystem** is the foundational level of storage. The fundamental unit of data on that level is a **file**. A file can be text or binary, it is not versioned, can be easily erased or overwritten.

A filesystem can be local or networked. A networked filesystem can be simple or distributed.

A **local filesystem** can be as simple as a locally mounted disk containing all the files needed for your machine learning project.

A **networked filesystem**, such as NFS or Amazon S3, can be accessed over the network by multiple physical or virtual machines. A networked filesystem can be **simple**, such as a Network Attached Storage, or NAS, or **distributed**, such as HDFS or Amazon S3. A distributed networked filesystem is stored and accessed over multiple machines in the network.

Despite its simplicity, filesystem-level storage can be appropriate for a variety of use cases, including:

**File sharing**
The simplicity of filesystem-level storage and support for standard protocols allows you to store and share data with a small group of colleagues with minimal effort.

**Local archiving**
Filesystem-level storage is a cost-effective option for archiving data thanks to the availability and accessibility of scale-out NAS solutions.

**Data protection**
Filesystem-level storage is a viable data protection solution thanks to built-in redundancy and replication.

Parallel access to the data on the filesystem level is fast for retrieval access but slow for storage, so it's an appropriate storage level for smaller teams and data.

An **object storage** is an application programming interface, or API, defined over a filesystem. Using an API, you can programmatically execute such operations on files as GET, PUT, or DELETE without worrying where the files are actually stored. The API is typically provided by an **API service** available on the network and accessible by HTTP or, more generally, TCP/IP or a different communication protocol suite.

The fundamental unit of data in an object storage level is an **object**. Objects are usually binary: images, sound, or video files, and other data elements having a certain **format**.

Such features as versioning and redundancy can be built into the API service. The access to the data stored on the object storage level can often be done in parallel, but the access is not as fast as on the filesystem level.

A canonical example of object storage is Amazon S3. It is not only the defacto standard way of storing data on Amazon Web Services (AWS) but many other object storage providers offer an S3 compatible API as well. Alternatively, Ceph is a storage platform, which implements object storage on a single distributed computer cluster and provides interfaces for both object- and filesystem-level storage. Ceph is often used as an alternative to S3 in on-premises computing systems.

The **database** level of data storage allows persistent, fast and scalable storage of **structured data** with fast parallel access for both storage and retrieval.

A modern database management system (DBMS) stores data in RAM, but software ensures that data is persisted (and operations on data are logged) to disk and never lost.

The fundamental unit of data at this level is a **row**. A row has a unique ID and contains values in columns. In a relational database, rows are organized in **tables**. Rows can have references to other rows in the same or different tables.

Databases are not especially well suited for storing binary data, though rather small binary objects sometimes can be stored in a column in the form of a **blob** — a Binary Large OBject &mdash. Blob is a collection of binary data stored as a single entity. More often though, a row stores references to binary objects stored elsewhere — in a filesystem or object storage.

The four most frequently used DBMS in the industry are Oracle, MySQL, Microsofr SQL Server and PostgresSQL. They all support SQL, an interface for accessing and modifying data stored in the databases, as well as create, modify and erase databases[4].

A **data lake** is a repository of data stored in its natural or raw format, usually in the form of object blobs or files. A data lake is usually an unstructured aggregation of data from multiple sources, including databases, logs, or intermediary data obtained as a result of expensive transformations of the original data.

Because the data is saved in the data lake in its raw format, including the structured data, in order to read data from a data lake, the analyst typically should be able to write the programming code that would read and parse the data stored in a file or a blob. Writing a script to parse the data file or a blob is an approach called **schema on read**, as opposed to **schema on write** in DBMS. In a DBMS, the schema of data is defined beforehand and, at each write, the DBMS makes sure that the data corresponds to the schema.

### 3.11.3 Data Versioning

If data is held and updated in multiple places, you might need to keep track of versions. Versioning of the data is also needed when you frequently update the model by collecting more data, especially in an automated way. This can happen, for example, when you work on such problems as automated driving, spam detection, or personalized recommendations. The

[4]Microsoft SQL Server indeed uses its proprietary Transact SQL (T-SQL) while Oracle uses Procedural Language SQL (PL/SQL).

data in the above three problems come either from a human driving a car, or the users using a specific service such as electronic mail or video streaming. Sometimes, after an update of the data, the new model exhibits a worse performance than the previous one and you would like to investigate why that happens by switching from one version of the data to another one.

Versioning of the data is also a critical element in the supervised learning scenario when the labeling is done by multiple labelers. Some labelers can assign very different labels to similar examples, which typically hurts the performance of the model. You would like to keep the examples annotated by different labelers separately and only merge them together when you build the model. If, after a careful analysis of the model performance, you find out that one or several labelers didn't provide quality or consistent labels, you would prefer to exclude such data from the training data, or relabel it. Data versioning would allow you to do so with minimal effort.

Data versioning can be implemented in several levels of complexity, from the most basic to the most elaborate.

**Level 0: data is unversioned.**
At this level, data may reside on a local filesystem, object storage or in a database. The advantage of having unversioned data is the speed and simplicity of dealing with the data, but that advantage is outweighed by potential problems you might encounter when working on your model. The first problem which you will most likely encounter is an impossibility to make versioned deployments. As we will discuss in later chapters, model deployments must be versioned. A deployed machine learning model is a mix of code and data. If the code is versioned, the data must be too. Otherwise, the deployment will be unversioned.

If you don't version deployments, you will not be able to get back to the previous level of performance in case of any problem with the model. Therefore, unversioned data is not recommended.

**Level 1: data is versioned as a snapshot at training time.**
At this level, data is versioned by storing, at training time, a snapshot of everything needed to build a model. Such an approach allows you to version deployed models and to get back to past performance. You should keep track of each version in some document, typically an Excel spreadsheet. That document should describe the location of the snapshot of both code and data, the values of hyperparameters and other metadata needed to reproduce the experiment if needed. If you don't have many models and don't update them too frequently, this level of versioning could be a viable versioning strategy. Otherwise, it's not recommended.

**Level 2: both data and code are versioned as one asset.**
At this level of versioning, small data assets, such as dictionaries, gazetteers, and small datasets, are stored jointly with the code in a version control system, such as **Git** or **Mercurial**. Large files are stored in an object storage, such as Amazon S3, with unique IDs. The training data is stored as JSON, XML or another standard format, referring

to these IDs and includes relevant metadata such as labels, the identity of the labeler, time of labeling, the tool used to label the data, and so on.

Tools like **Git Large File Storage** (LFS) automatically replaces large files such as audio samples, videos, large datasets, and graphics with text pointers inside Git, while storing the file contents on a remote server.

The version of the dataset is defined by the git signatures of the code and the data file. It can also be helpful to add a timestamp to easily identify a needed version.

**Level 3: using or building a specialized data versioning solution.**
Such data versioning software as **DVC**, **Pachyderm**, and **Quilt** provide additional tools for versioning of the data. They typically interoperate with code versioning software, such as Git.

Level 2 of versioning is a recommended way of implementing versioning for most projects. If you feel like Level 2 is not sufficient for your needs, explore Level 3 solutions or consider building your own. Otherwise, I don't recommend that approach, as it adds additional complexity to what already is a complex engineering project.

### 3.11.4 Documentation and Metadata

While you are actively working on a machine learning project, you are often capable of keeping important detail about the data in your head. However, once the project goes to production and you switch to another project, this information in your head will eventually become less detailed.

Before you switch to another project, you should make sure that others are able to understand your data and use it properly.

If the data is self-explanatory, then you might leave it undocumented. However, it's rather rare that someone who didn't create data can easily understand it and know how to use it just by looking at it.

Documentation has to accompany any data asset that was used to build a model. This documentation has to contain the following details:

- what that data means,
- how it was collected or methods used to create it,
- what format is used to store it,
- types of attributes or features (which values are allowed for each atribute or feature),
- possible values for labels or the allowable range for a numerical target.

#### 3.11.5 Data Lifecycle

Some data can be stored indefinitely. However, in some business contexts, you might be allowed to store some data for a specific time and then you might have to erase it. If such restrictions apply to the data you work with, you have to make sure that a reliable alerting system is in place. That alerting system has to contact the person responsible for the data erasure and have a backup plan in case that person is not available for some reason. Don't forget that the consequences for the organization for not erasing data can in some cases be very serious.

For every sensitive data asset, a **data lifecycle document** has to describe the asset, the circle of persons who have access to that data asset, both during the project development and after the work on the project is over. The document has to describe how long will that data asset be stored and whether or not it has to be explicitly destroyed.

## 3.12 Data Manipulation Best Practices

**Reproducibility** should be an important concern in everything you do, including data collection and preparation. You should avoid transforming data manually or by using the tools included into powerful text edititors or comand line shells, such as regular expressions, ad hoc **awk**, **sed** and piped expressions.

Usually, the data collection and transformation activities consist of multiple stages, for example downloading data from web APIs or databases, replacing multiword expressions by unique tokens, removing stop-words and noise, imputation of missing values, and so on. Each state in this multistage process has to be implemented as a programming script, such as Python, awk or R script with its inputs and outputs. If you are organized like that in your work, it will allow you keep all traces of all changes the data undergoes, and if, at any stage something wrong happens to the data, you can always fix the script and run the entire data processing pipeling from scratch.

On the other hand, manual interventions can be hard to reproduce, apply to an updated data or scale for much more data (once you can afford getting more data or a different dataset).

## 3.13 Contributors

I'm grateful to the following people for their valuable contributions to the quality of this chapter: Marko Peltojoki, Gregory V., Win Pet, Yihwa Kim, Timothée Bernard, Christopher Thompson, Kelvin Sundli, Marwen Sallem, **Alexander Sack**, Daniel Bourguet, Aliza Rubenstein, Brad Ezard, Niklas Hansson, Sylvain Truong, Tim Flocke, Alice O, Fernando Hannaka, Oliver Proud, Ana Fotina, and Abhijit Kumar.