From Prototype to Production

The tools we've discussed in this book are great for many machine learning applications, and allow very quick analysis and prototyping. Python and `scikit-learn` are also used in production systems in many organizations—even very large ones like international banks and global social media companies. However, many companies have complex infrastructure, and it is not always easy to include Python in these systems. That is not necessarily a problem. In many companies, the data analytics teams work with languages like Python and R that allow the quick testing of ideas, while production teams work with languages like Go, Scala, C++, and Java to build robust, scalable systems. Data analysis has different requirements from building live services, and so using different languages for these tasks makes sense. A relatively common solution is to reimplement the solution that was found by the analytics team inside the larger framework, using a high-performance language. This can be easier than embedding a whole library or programming language and converting from and to the different data formats.

Regardless of whether you can use `scikit-learn` in a production system or not, it is important to keep in mind that production systems have different requirements from one-off analysis scripts. If an algorithm is deployed into a larger system, software engineering aspects like reliability, predictability, runtime, and memory requirements gain relevance. Simplicity is key in providing machine learning systems that perform well in these areas. Critically inspect each part of your data processing and prediction pipeline and ask yourself how much complexity each step creates, how robust each component is to changes in the data or compute infrastructure, and if the benefit of each component warrants the complexity. If you are building involved machine learning systems, we highly recommend reading the paper "Machine Learning: The High Interest Credit Card of Technical Debt", published by researchers in Google's machine learning team. The paper highlights the trade-off in creating and maintaining machine learning software in production at a large scale. While the issue of technical debt *is* particularly pressing in large-scale and long-term projects, the lessons learned can help us build better software even for short-lived and smaller systems.

Testing Production Systems

In this book, we covered how to evaluate algorithmic predictions based on a test set that we collected beforehand. This is known as *offline evaluation*. If your machine learning system is user-facing, this is only the first step in evaluating an algorithm, though. The next step is usually *online testing* or *live testing*, where the consequences of employing the algorithm in the overall system are evaluated. Changing the recommendations or search results users are shown by a website can drastically change their behavior and lead to unexpected consequences. To protect against these surprises, most user-facing services employ *A/B testing*, a form of blind user study. In

A/B testing, without their knowledge a selected portion of users will be provided with a website or service using algorithm A, while the rest of the users will be provided with algorithm B. For both groups, relevant success metrics will be recorded for a set period of time. Then, the metrics of algorithm A and algorithm B will be compared, and a selection between the two approaches will be made according to these metrics. Using A/B testing enables us to evaluate the algorithms "in the wild," which might help us to discover unexpected consequences when users are interacting with our model. Often A is a new model, while B is the established system. There are more elaborate mechanisms for online testing that go beyond A/B testing, such as *bandit algorithms*. A great introduction to this subject can be found in the book *Bandit Algorithms for Website Optimization* by John Myles White (O'Reilly).

Building Your Own Estimator

This book has covered a variety of tools and algorithms implemented in `scikit-learn` that can be used on a wide range of tasks. However, often there will be some particular processing you need to do for your data that is not implemented in `scikit-learn`. It may be enough to just preprocess your data before passing it to your `scikit-learn` model or pipeline. However, if your preprocessing is data dependent, and you want to apply a grid search or cross-validation, things become trickier.

In Chapter 6 we discussed the importance of putting all data-dependent processing inside the cross-validation loop. So how can you use your own processing together with the `scikit-learn` tools? There is a simple solution: build your own estimator! Implementing an estimator that is compatible with the `scikit-learn` interface, so that it can be used with `Pipeline`, `GridSearchCV`, and `cross_val_score`, is quite easy. You can find detailed instructions in the `scikit-learn` documentation, but here is the gist. The simplest way to implement a transformer class is by inheriting from `BaseEstimator` and `TransformerMixin`, and then implementing the `__init__`, `fit`, and `predict` functions like this: