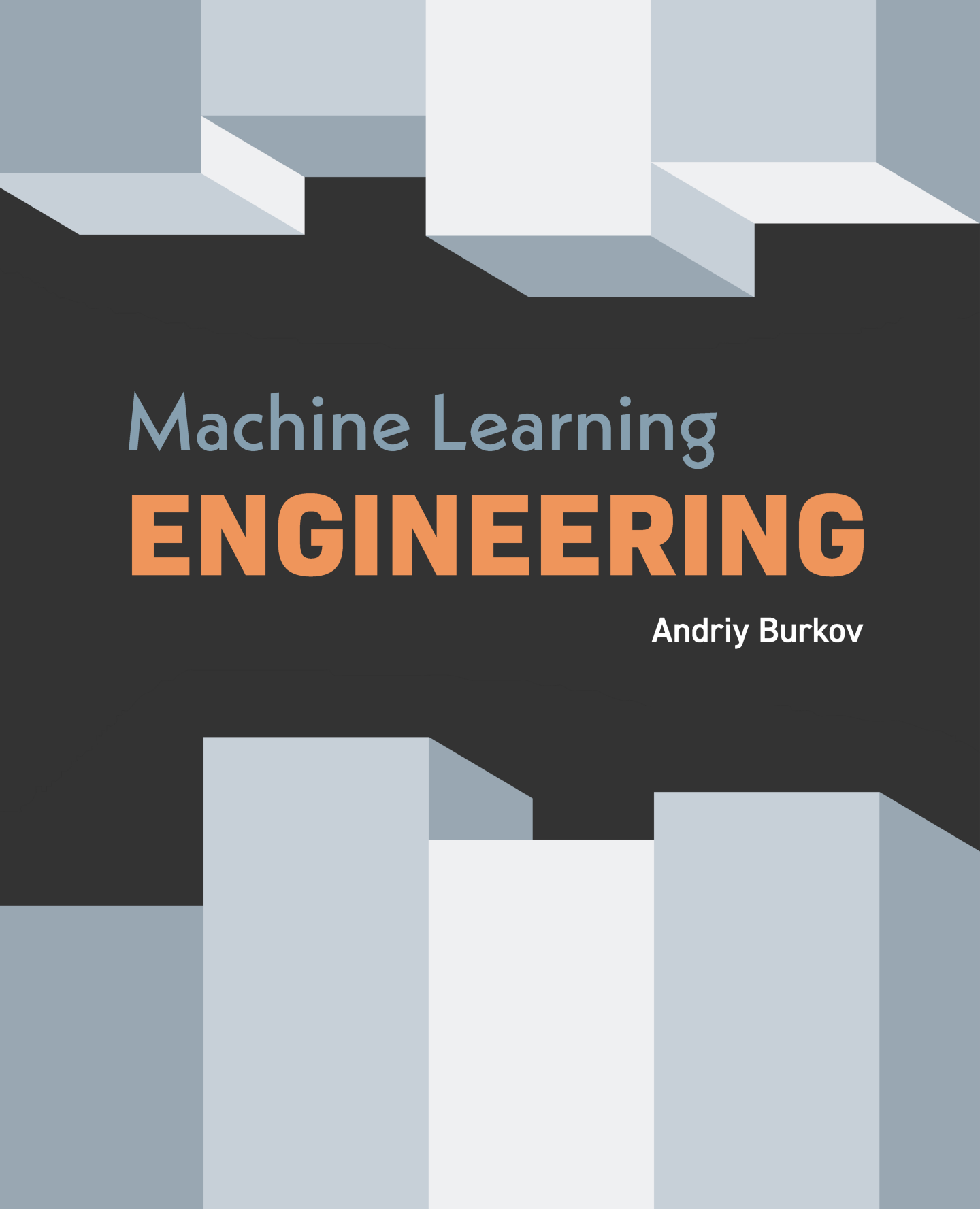
Machine Learning
ENGINEERING
Andriy Burkov

“In theory, there is no difference between theory and practice. But in practice, there is.”

— Benjamin Brewster

“The perfect project plan is possible if one first documents a list of all the unknowns.”

— Bill Langley

“When you’re fundraising, it’s AI. When you’re hiring, it’s ML. When you’re implementing, it’s linear regression. When you’re debugging, it’s printf().”

— Baron Schwartz

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

5 Supervised Model Training (Part 1)

Model training (or modeling) is the fourth stage in the machine learning project life cycle:

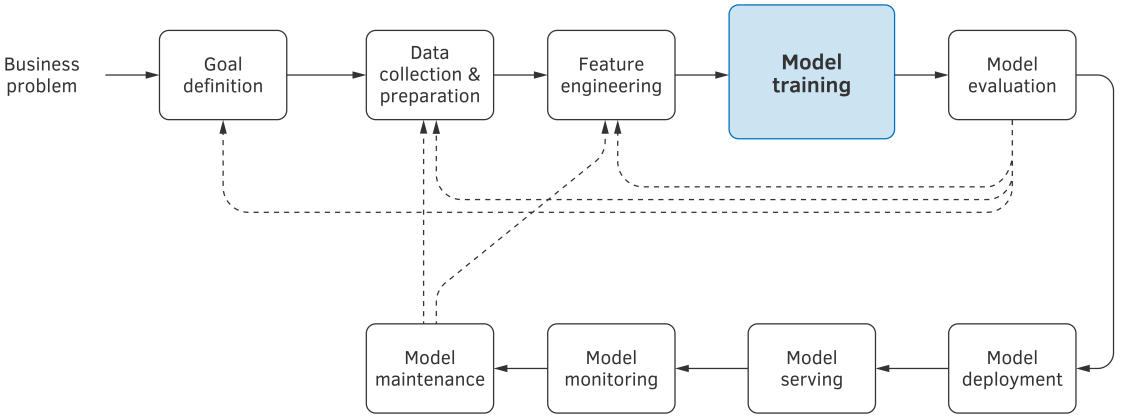

Figure 1: Machine learning project life cycle.

It's clear that without training, no model will be built. However, model training is one of the most overrated activities in machine learning. On average, a machine learning engineer spends only $5 - 10\%$ of their time on modeling, if at all. Successful data collection, preparation, and feature engineering are more important. Usually, modeling is simply applying an algorithm from scikit-learn or R to your data, and randomly trying several combinations of hyperparameters. So, if you skipped the preceding two chapters and jumped directly into modeling, please go back and read those chapters, they are important.

As indicated by this chapter's title, I have divided supervised model training into two parts. In this first part, we will consider learning preparation, choosing the learning algorithm, a shallow learning strategy, assessing model performance, bias-variance tradeoff, regularization, the concept of the machine learning pipeline, and hyperparameter tuning.

5.1 Before You Start Working on the Model

Before working on the model, you should validate schema conformity, define an achievable level of performance, choose a performance metric, and make several other decisions.

5.1.1 Validate Schema Conformity

First, ensure the data conforms to the schema, as defined by the **schema file**. Even if you initially prepared the data, it's likely the original data and the current data are not the same. This difference can be explained by various factors, most probably:

- the method used to persist the data, to hard drive or to database, contains an error;
- the method you used to read the data, from where it was persisted, contains an error;
- someone else may have changed the data, or the schema, without informing you.

These schema errors must be detected, identified, and corrected just as when a programming code error is detected. If needed, the entire data collection and preparation pipeline should be run from scratch, as we discussed at the end of Chapter 3 when talked about **reproducibility**.

5.1.2 Define an Achievable Performance Level

Defining an achievable performance level is a crucial step. It gives you an idea of when to stop trying to improve the model. Here are some guidelines:

- if a human can label examples without too much effort, math, or complex logic derivations, then you can hope to achieve human-level performance with your model;
- if the information needed to make a labeling decision is fully contained in the features, you can expect to have near-zero error;
- if the input feature vector has a high number of signals (such as pixels in an image, or words in a document), you can expect to come close to near-zero error;
- if you have a computer program solving the same classification or regression problem, you can expect your model to perform at least as well. Often the machine learning model performance can improve as more labeled data comes in; and,
- if you observe a similar, but different system, you can expect to get a similar, but different machine learning model performance.

5.1.3 Choose a Performance Metric

We will talk about assessing the model performance later. For now, there are several ways — metrics — to estimate the level of model performance (its quality). There's no single best metric you can use for every project. You will choose based on your data and the problem.

It is recommended to choose one, and only one, **performance metric** before you start working on the model. Then, compare different models and track the overall progress by using this one metric.

In Section 5.5, you will read about the most popular and handy model performance metrics, and about the approaches allowing us to combine multiple metrics to obtain a single number.

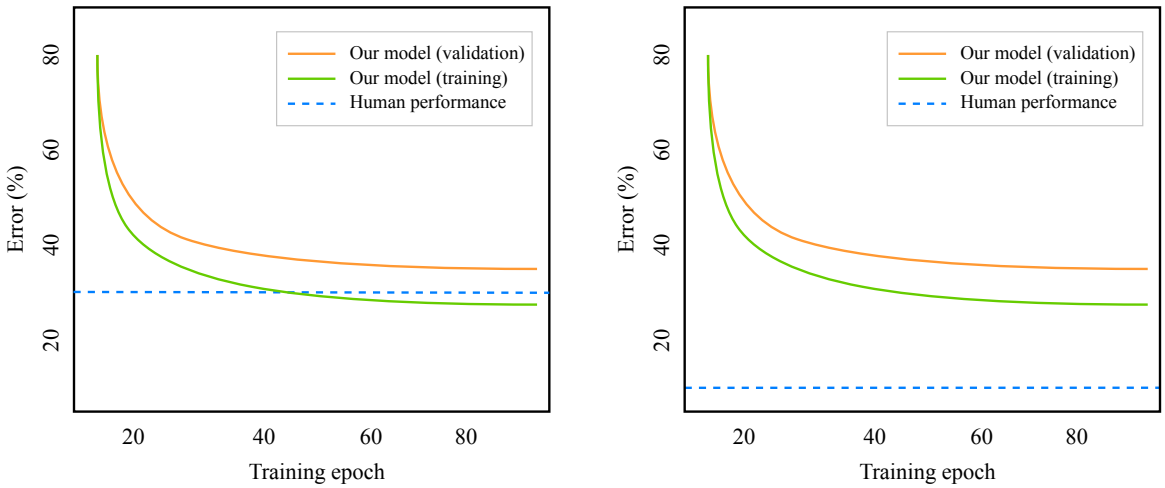

Figure 2: A model performance compared to a human-performance baseline: (a) the model looks good, so we can decide to regularize it or add more training examples; (b) the model isn't performing well, so we need to add more features, or increase the model complexity.

5.1.4 Choose the Right Baseline

Before you start working on a predictive model, it is important to establish baseline performance on your problem. A **baseline** is a model or an algorithm that provides a reference point for comparison.

Having a baseline gives an analyst confidence that the machine-learning-based solution works. If the value of the performance metric for the machine learning model is better than the value obtained using the baseline, then machine learning provides value.

Comparing your current model's performance to a baseline can orient the work in different directions. Let's say we know that human-level performance is achievable on our problem. We then take human performance as a baseline, as shown in Figure 2. In Figure 2a, the model looks good, so we can decide to regularize it or add more training examples. On the other hand, in Figure 2b, the model isn't performing well, so we should add more features, or increase the **model complexity**.

The baseline is a model or an algorithm that gets an input, and outputs a prediction. The baseline's prediction output must be of the same nature as the model's prediction. Otherwise, you cannot compare them.

A baseline doesn't have to be the result of any learning algorithm. It can be a rule-based or heuristic algorithm, a simple statistic applied to the training data, or something else.

The two most commonly used baseline algorithms are:

- random prediction, and
- zero rule.

The **random prediction algorithm** makes a prediction by randomly choosing a label from the collection of labels assigned to the training examples. In the classification problem, it corresponds to randomly picking one class from all classes in the problem. In the regression problem it means selecting from all unique target values in the training data.

The **zero rule algorithm** yields a tighter baseline than the random prediction algorithm. This means that it usually improves the value of the metric as compared to random prediction. To make predictions, the zero rule algorithm uses more information about the problem.

In classification, the zero rule algorithm strategy is to always predict the class most common in the training set, independently of the input value. It can look ineffective, but consider the following problem. Let the training data for your classification problem contain 800 examples of the positive class, and 200 examples of the negative class. The zero rule algorithm will predict the positive class all the time, and the **accuracy** (one of the popular performance metrics that we will consider in Section 5.5.2) of the baseline will be $800/1000 = 0.8$ or 80%, which is not bad for such a simple classifier. Now you know that your statistical model, independently of how close it is to the optimum, must have an accuracy of at least 80%.

Now, let's consider the zero rule algorithm for regression. According to the zero rule algorithm, the strategy for regression is to predict the sample average of the target values observed in the training data. This strategy will likely have a lower error rate than random prediction.

If you work on a standard, so-called classical, prediction problem, you can use a state-of-the-art algorithm found in a popular library such as Python's scikit-learn. For text classification, for example, represent the text as **bag-of-words**, and then train a **support vector machine** model with a **linear kernel**. Then try to beat that result with your own more advanced approach. This approach would also work well with image classification, machine translation, and other well-studies, so-called benchmark problems.

For a general numerical dataset, a linear model such as linear or logistic regression, or **k-nearest neighbors**, for $k = 5$, would be a decent baseline. For image classification, a simple **convolutional neural network** (CNN), with three convolutional layers ($32 - 64 - 32$ units per layer, each convolutional layer followed by a max pooling layer and a dropout layer) and two fully connected layers at the end (one with 128 units, and one with the number of units corresponding to the number of desired outputs) would be a good baseline.

You could also use an existing rule-based system, or build your own simple rule-based system. For example, if the problem is to build a model that predicts whether a given website visitor will like a recommended article, a simple rule-based system could work as follows. Take all articles liked by the user, find the top ten words in those articles according to their **TF-IDF** score, and then predict that the user will like an article if at least five of those ten can be found in the recommended article. Additionally, multiple specialized machine learning

libraries and APIs are available online. If they can be used directly, or repurposed to solve your problem, you should definitely consider them as a baseline.

Finding a good human baseline is not always simple. You might use Amazon **Mechanical Turk** service. Mechanical Turk (MT) is a web-platform where people solve simple tasks for a reward. MT provides an API that you can call to get human predictions. The quality of such predictions can vary from very low to relatively high, depending on the task and the reward. MT is relatively inexpensive, so you can get predictions fast and in large numbers.

To increase the quality of the predictions provided by turkers (this is how MT human workers are called), some analysts use an **ensemble of turkers**. You can ask three or five turkers to label the same example, and then pick the majority class among the labels (or average labels for regression). A more expensive alternative is to ask domain experts (or an ensemble, for even better quality) to label your data.

5.1.5 Split Data Into Three Sets

Recall that three sets are generally needed to build a solid model. The first, the **training set**, is used to train the model. It is the data the machine learning algorithm "sees." The second and third are the holdout sets. The **validation set** is not seen by the machine learning algorithm. The data analyst uses it to estimate the performance of different machine learning algorithms (or the same algorithm configured with different values of hyperparameters) or models when applied to new data. The remaining **test set**, which is also not seen by the learning algorithm, is used at the end of the project to evaluate and report the performance of the model the best performing on the validation data.

The process of splitting the entire dataset into three sets is described in Section **??** of Chapter 3. Here, I only reiterate the two most important properties of that process:

1. Validation and test sets must come from the same statistical distribution. That is, their properties have to be maximally similar, but the examples belonging to the two sets must be, obviously and ideally, different and obtained independently of one another.
2. Draw validation and test data from a distribution that looks much like the data you expect to observe once the model is deployed in production. It can be different from the distribution of the training data.

A couple of words about the latter point. Most of the time, the analyst simply shuffles the entire dataset, and then randomly fills the three sets from this shuffled data. In practice, however, it's common to have many examples that do not look like the production data. Sometimes, these examples are abundant and/or inexpensive. Using this data in the project may result in **distribution shift**, and the analyst may or may not be aware of it.

If you are aware of distribution shift, you will place all those easily available examples into your training set, but will avoid using them in the validation and test sets. This way, you evaluate the models against the data that is similar to that in your production setting. Doing

otherwise might result in achieving overly optimistic values of the performance metric during model testing, and selecting for production a suboptimal model.

The distribution shift can be a hard problem to tackle. Using a different data distribution for training could be a conscious choice because of the data availability. However, the analyst may be unaware that the statistical properties of the training and development data are different. This often happens when the model is frequently updated after production deployment, and new examples are added to the training set. The properties of the data used to train the model, and that of the data used to validate and test it, can diverge over time. Section **??** in the next chapter provides guidance on how to handle that problem.

5.1.6 Preconditions for Supervised Learning

Before you start working on your model, make sure the following conditions are satisfied:

1. You have a labeled dataset.
2. You have split the dataset into three subsets: training, validation, and test.
3. Examples in the validation and test sets are statistically similar.
4. You engineered features and filled missed values using only the training data.
5. You converted all examples into numerical feature vectors.[1]
6. You have selected a performance metric that returns a single number (see Section 5.5).
7. You have a baseline.

5.2 Representing Labels for Machine Learning

In the classical formulation of classification, labels look like values of a categorical feature. For example, in image classification, the labels could be "cat," "dog," "car," "building," and so on.

Some machine learning algorithms, like those you find in scikit-learn, accept labels in their natural form: strings. The library take care of transforming strings to numbers that are accepted by a specific learning algorithm.

Some implementations, however, like those in neural networks, require the analyst to transform the labels to numbers.

5.2.1 Multiclass Classification

In the case of **multiclass classification** (that is, when the model predicts only one label given an input feature vector), **one-hot encoding** is typically used to convert labels to

[1] As mentioned in the previous chapter, most modern machine learning libraries and packages expect numerical feature vectors. However, some algorithms, like decision tree learning, can naturally work with categorical features.

binary vectors. For example, let your classes be $\{\text{dog}, \text{cat}, \text{other}\}$, and you have the following data:

Image	Label
image_1.jpg	dog
image_2.jpg	dog
image_3.jpg	cat
image_4.jpg	other
image_5.jpg	cat

One-hot encoding would generate the following binary vectors for your classes:

$$\begin{aligned} \text{dog} &= [1, 0, 0], \\ \text{cat} &= [0, 1, 0], \\ \text{other} &= [0, 0, 1]. \end{aligned}$$

After you convert categorical labels into binary vectors, your data becomes:

Image	Label
image_1.jpg	[1,0,0]
image_2.jpg	[1,0,0]
image_3.jpg	[0,1,0]
image_4.jpg	[0,0,1]
image_5.jpg	[0,1,0]

5.2.2 Multi-label Classification

In **multi-label classification**, the model may predict several labels for one input at the same time (for example, an image can contain both a dog and a cat). In this case, you can use **bag-of-words** to represent the labels assigned to each example. Let your data be as follows:

Image	Labels
image_1.jpg	dog, cat
image_2.jpg	dog
image_3.jpg	cat, other
image_4.jpg	other
image_5.jpg	cat, dog

After you convert labels into binary vectors, your data becomes:

Image	Labels
image_1.jpg	[1,1,0]
image_2.jpg	[1,0,0]
image_3.jpg	[0,1,1]
image_4.jpg	[0,0,1]
image_5.jpg	[1,1,0]

Read the documentation of the specific implementation of a learning algorithm to know the format of the input expected by the learning algorithm.

5.3 Selecting the Learning Algorithm

Choosing a machine learning algorithm can be a difficult task. If you had a lot of time, you could try all of them. However, usually, the time to solve a problem is limited. To make an informed choice, you can ask yourself several questions before starting to work on the problem. Depending on your answers, you can shortlist some algorithms and try them on your data.

5.3.1 Main Properties of a Learning Algorithm

Below are several questions and answers which may guide you in choosing a machine learning algorithm or model.

Explainability

Do the model predictions require explanation for a non-technical audience? The most accurate machine learning algorithms and models are so-called "black boxes." They make very few prediction errors, but it may be difficult to understand, and even harder to explain, why a model or an algorithm made a specific prediction. Examples of such models are **deep neural networks** and **ensemble models**.

In contrast, **kNN**, **linear regression**, and **decision tree learning** algorithms are not always the most accurate. However, their predictions are easy to interpret by a non-expert.

In-memory vs. out-of-memory

Can your dataset be fully loaded into the RAM of your laptop or server? If yes, then you can choose from a wide variety of algorithms. Otherwise, you would prefer **incremental learning algorithms** that can improve the model by reading data gradually. Examples of such algorithms are **Naïve Bayes** and the algorithms for training neural networks.

Number of features and examples
How many training examples do you have in your dataset? How many features does each example have? Some algorithms, including those used for training **neural networks** and **random forests**, can handle a huge number of examples and millions of features. Others, like the algorithms for training **support vector machines** (SVM), can be relatively modest in their capacity.

Nonlinearity of the data
Is your data linearly separable? Can it be modeled using a linear model? If yes, SVM with the linear kernel, linear and logistic regression can be good choices. Otherwise, deep neural networks or ensemble models might work better.

Training speed
How much time is a learning algorithm allowed to use to build a model, and how often you will need to retrain the model on updated data? If training takes two days, and you need to retrain your model every 4 hours, then your model will never be up to date. Neural networks are slow to train. Simple algorithms like linear and logistic regression, or decision trees, are much faster.

Specialized libraries contain very efficient implementations of some algorithms. You may prefer to do research online to find such libraries. Some algorithms, such as random forest learning, benefit from multiple CPU cores, so their training time can be significantly reduced on a machine with dozens of cores. Some machine learning libraries leverage GPU (graphics processing unit) to speed up training.

Prediction speed
How fast must the model be when generating predictions? Will your model be used in a production environment where very high throughput is required? Models like SVMs and linear and logistic regression models, and not-very-deep feedforward neural networks, are extremely fast at prediction time. Others, like kNN, ensemble algorithms, and very deep or recurrent neural networks, are slower.

If you don't want to guess the best algorithm for your data, a popular way to choose one is by testing several candidate algorithms on the **validation set** as a hyperparameter. We talk about hyperparameter tuning in Section 5.6.

5.3.2 Algorithm Spot-Checking

Shortlisting candidate learning algorithms for a given problem is sometimes called **algorithm spot-checking**. For the most effective spot-checking, it is recommended to:

- select algorithms based on different principles (sometimes called orthogonal), such as instance-based algorithms, kernel-based, shallow learning, deep learning, ensembles;
- try each algorithm with $3 - 5$ different values of the most sensitive hyperparameters (such as the number of neighbors k in k-nearest neighbors, penalty C in support vector machines, or decision threshold in logistic regression);

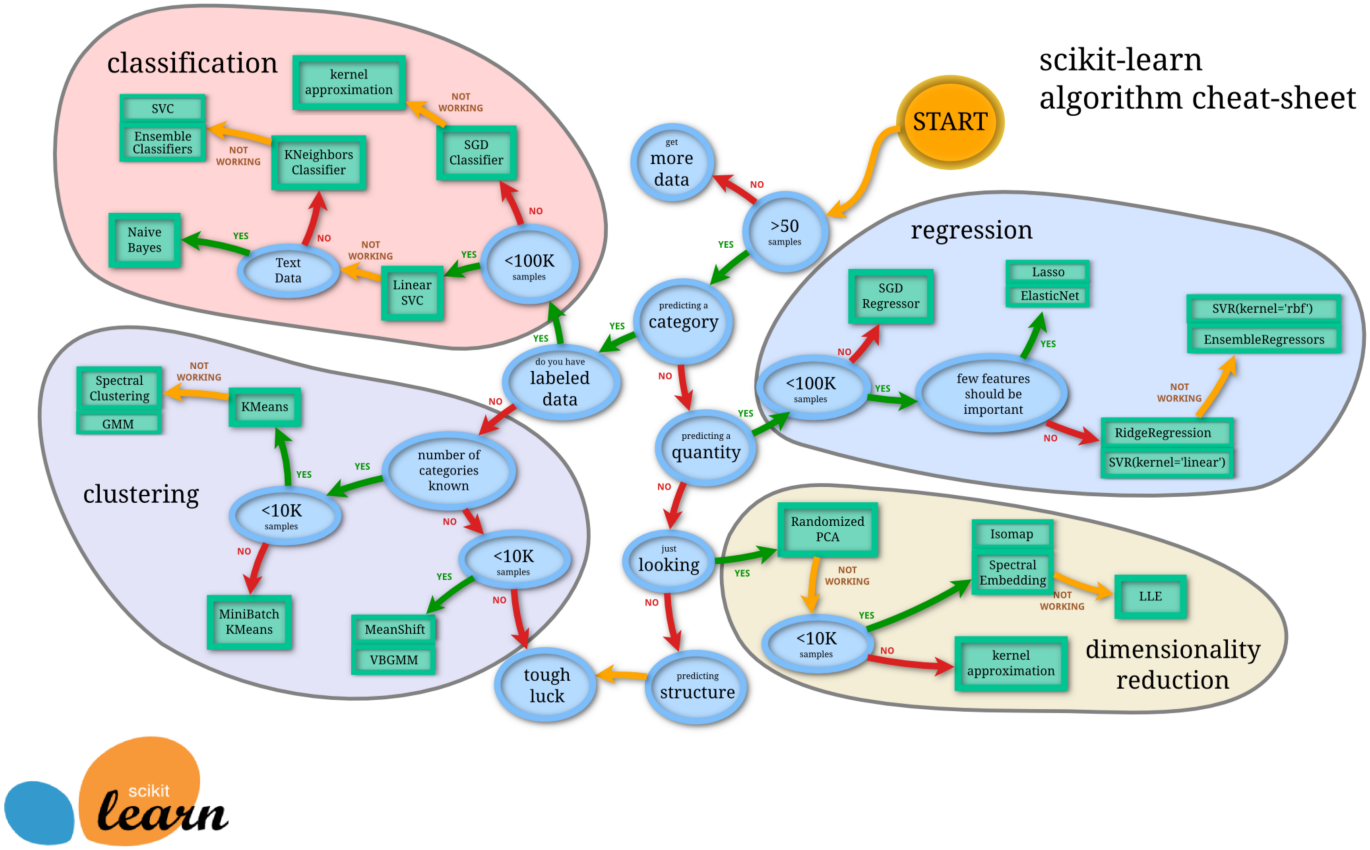

Figure 3: Machine learning algorithm selection diagram for scikit-learn. Source: scikit-learn.org

- use the same training/validation split for all experiments,
- if the learning algorithm is not deterministic (such as the learning algorithms for neural networks and random forests), run several experiments, and then average the results;
- once the project is over, note which algorithms performed the best, and use this information when working on a similar problem in the future.

While you don't know your problem well, try to solve it using as many orthogonal approaches as possible, rather than spending a lot of time on the most promising approach. It is generally a better idea to spend time experimenting with new algorithms and libraries, rather than trying to squeeze the maximum from the one with which you have the most experience.

If you don't have time to carefully spot-check algorithms, one simple "hack" is to find an efficient implementation of a learning algorithm or a model that most modern papers claim to beat, when applied to a problem similar to yours, and use it for solving your problem.

If you use scikit-learn, you could try their algorithm selection diagram shown in Figure 3.

5.4 Building a Pipeline

Many modern machine learning packages and frameworks support the notion of a **pipeline**. A pipeline is a sequence of transformations the training data goes through, before it becomes a model. An example of a pipeline used to train a document classification model out of a collection of labeled text documents is shown below:

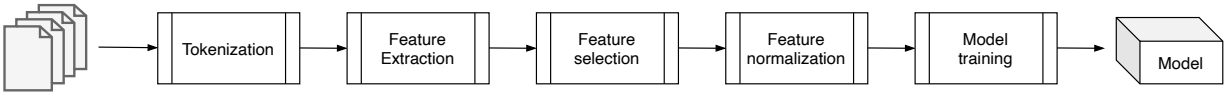

Figure 4: A pipeline used to produce a model starting with raw data.

Every stage of a pipeline receives the output of the previous stage, except for the first stage, whose input is the training dataset.

Below is a Python code fragment that constructs a simple **scikit-learn** pipeline. It consists of two steps: 1) dimensionality reduction using **Principal Component Analysis** (PCA), and 2) training a **support vector machine** (SVM) classifier:

```
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.decomposition import PCA

# Define a pipeline
pipe = Pipeline([('dim_reduction', PCA()), ('model_training', SVC())])

# Train parameters of both PCA and SVC
```

```
pipe.fit(X, y)

# Make a prediction
pipe.predict(new_example)
```

When the command `pipe.predict(new_example)` is executed, the input example is first transformed into a reduced dimensionality vector using the PCA model. That reduced dimensionality vector is used as input to the SVM model. PCA and SVM models were trained, one after the other, when the command `pipe.fit(X, y)` were executed.

Unfortunately, defining and training pipelines in R is not as straightforward as in Python, so we don't put the code in the book.

The pipeline can be saved to a file similar to saving a model. It will be deployed to production and used to generate predictions. In other words, during the **scoring**, the input example passes through the entire pipeline and "becomes" an output.

As you can see, the notion of a pipeline is a generalization of the notion of a model. From this point forward, unless stated otherwise, when I refer to model training, saving, deployment, serving, monitoring, or post-production maintenance, I mean the entire pipeline.

Before we consider the challenge of training a model, we need to decide how to measure the model quality. Often, we have a choice between several competing models, so-called model candidates, but only one will be deployed in production.

5.5 Assessing Model Performance

Remember, the **holdout data** consists of examples the learning algorithm didn't see during training. If our model performs well on a holdout set, we can say our model **generalizes well** and is of good quality or, simply, that it's good. The most common way to get a good model is to compare different models by calculating a **performance metric** on the holdout data.

5.5.1 Performance Metrics for Regression

Regression and classification models are assessed using different metrics. Let's first consider performance metrics for regression: mean squared error (MSE), median absolute error (MAE), and almost correct predictions error rate (ACPER).

The metric most often used to quantify the performance of a regression model is the same as the **cost function**: **mean squared error** (MSE), defined as,

$$\text{MSE}(f) \stackrel{\text{def}}{=} \frac{1}{N} \sum_{i=1...N} (f(\mathbf{x}_i) - y_i)^2, \tag{1}$$

where f is the model that takes a feature vector $\mathbf{x}$ as input and outputs a prediction, and i, ranging from 1 to N, denotes the index of an example from a dataset.

A **well-fitting** regression model predicts values close to the observed data values. The **mean model**, which always predicts the average of the training data labels, generally would be used if there were no informative features. Therefore, the regression model should fit better than that of the mean model. Thus, the mean model acts as a **baseline**. If the regression model MSE is greater than the baseline MSE, then we have a problem in our regression model. It may be **overfitting** or **underfitting** (we consider these in Section 5.8). It could also be that the problem was defined with an error, or the programming code contains a bug.

If the data contains outliers, the examples very far from the "true" regression line, they can significantly affect the value of MSE. By definition, the squared error for such outlying examples will be high. In such situations, it is better to apply a different metric, the **median absolute error**, MdAE:

$$\text{MdAE} \stackrel{\text{def}}{=} \text{median}\left(\{|f(\mathbf{x}_i) - y_i|\}_{i=1}^{N}\right),$$

where $\{|f(\mathbf{x}_i) - y_i|\}_{i=1}^{N}$ denotes the set of absolute error values for all examples, from $i = 1$ to N, on which the evaluation of the model is performed.

The **almost correct predictions error rate** (ACPER) is the percentage of predictions that is within p percentage of the true value. To calculate ACPER, proceed as follows:

1. Define a threshold percentage error that you consider acceptable (let's say 2%).
2. For each true value of the target y_i, the desired prediction should be between $y_i + 0.02y_i$ and $y_i - 0.02y_i$.
3. By using all examples $i = 1, \ldots, N$, calculate the percentage of predicted values fulfilling the above rule. This will give the value of the ACPER metric for your model.

5.5.2 Performance Metrics for Classification

For classification, things are a little more complicated. The most widely used metrics to assess a classification model are:

- precision-recall,
- accuracy,
- cost-sensitive accuracy, and
- area under the ROC curve (AUC).

To simplify, I will illustrate with a binary classification problem. Where necessary, I show how to extend the approach to the multiclass case.

First, we need to understand the confusion matrix.

A **confusion matrix** is a table that summarizes how successful the classification model is at predicting examples belonging to various classes. One axis of the confusion matrix is the class that the model predicted; the other axis is the actual label. Let's say, our model predicts classes "spam" and "not_spam":

	spam (predicted)	not_spam (predicted)
spam (actual)	23 (TP)	1 (FN)
not_spam (actual)	12 (FP)	556 (TN)

The above matrix shows that out of 24 actual spam examples, the model correctly classified 23. In this case, we say that we have 23 **true positives** or TP = 23. The model incorrectly classified 1 spam example as not_spam. In this case, we have 1 **false negative**, or FN = 1. Similarly, out of 568 actual not_spam examples, the model classified correctly 556 and incorrectly 12 examples (556 **true negatives**, TN = 556, and 12 **false positives**, FP = 12).

The confusion matrix for multiclass classification has as many rows and columns as there are different classes. It can help you to determine mistake patterns. For example, a confusion matrix could reveal that a model trained to recognize different species of animals tends to mistakenly predict "cat" instead of "panther," or "mouse" instead of "rat." In this case, you can add more labeled examples of these species to help the learning algorithm "see" the difference between those animals. Alternatively, you might add features that would help the learning algorithm do better at distinguishing between those pairs of species.

The confusion matrix is used to calculate three performance metrics: precision, recall, and accuracy. Precision and recall are most frequently used to assess a binary model.

Precision is the ratio of true positive predictions to the overall number of positive predictions:

$$\text{precision} \stackrel{\text{def}}{=} \frac{\text{TP}}{\text{TP} + \text{FP}}.$$

Recall is the ratio of true positive predictions to the overall number of positive examples:

$$\text{recall} \stackrel{\text{def}}{=} \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

To understand the meaning and the importance of precision and recall for model assessment, it's useful to think about the prediction problem as the problem of research of documents in a database using a query. The precision is the proportion of relevant documents actually found in the list of all returned documents. The recall is the ratio of the relevant documents returned by the search engine, compared to the total number of relevant documents that should have been returned.

In spam detection, we want to have high precision, to avoid wrongly placing a legitimate message in our spam folder. We are willing to tolerate lower recall, since we can deal with some spam messages in our inbox.

In practice, we choose between high precision or high recall. It's practically impossible to have both. This is called the **precision-recall tradeoff**. We can achieve either by various means:

- by assigning a higher weigh to the examples of a specific class. For example, SVM in scikit-learn accepts weights of classes as input;
- by tuning hyperparameters to maximize either precision or recall on the validation set;
- by varying the decision threshold for algorithms that return prediction scores. Let's say we have a logistic regression model or a decision tree. To increase precision (at the cost of a lower recall), we can decide that the prediction will be positive only if the score returned by the model is higher than 0.9 (instead of the default value of 0.5).

Even if precision and recall are defined for binary classification, you can also use them to assess a **multiclass classification** model. First select a class for which you want to assess these metrics. Then you consider all examples of the selected class as positives and all examples of the remaining classes as negatives.

In practice, to compare the performance of two models, you would prefer to have only one number that represents the performance of each model. For example, you would like to avoid situations where the first model has a higher precision, when the second model has a higher recall: if it's the case, which model is better?

One way to compare models based on one number is to threshold the minimum acceptable value for one metric, say recall, and then only compare models based on the value of another metric. For example, say you will accept any model whose recall is above 90%. Then you will give preference to the model whose precision is the highest (assuming that its recall is above 90%). This technique is known as **optimizing and satisficing technique**.

Some practitioners use a combination of precision and recall called **F-measure**, also known as **F-score**. The traditional F-measure, or F_1-score, is the harmonic mean of precision and recall:

$$F_1 = \left(\frac{2}{\text{recall}^{-1} + \text{precision}^{-1}} \right) = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

More generally, F-measure is parametrized with a positive real β, chosen such that recall is considered β times as important as precision:

$$F_\beta = (1 + \beta^2) \times \frac{\text{precision} \times \text{recall}}{(\beta^2 \times \text{precision}) + \text{recall}}$$

Two commonly used values for β are 2, which weighs recall twice as high as precision, and 0.5, which weighs recall twice as low as precision.

You should find a way to combine the two metrics that works best for your problem. Besides F-score, there are other ways to obtain a single number by combining multiple metrics:

- simple average, or weighted average of metrics;
- threshold $n-1$ metrics and optimize the n^{th} (a generalization of the above optimizing and satisficing technique);
- invent your own domain-specific "recipe."

Accuracy is given by the number of correctly classified examples, divided by the total number of classified examples. In terms of the confusion matrix, it is given by:

$$\text{accuracy} \stackrel{\text{def}}{=} \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{2}$$

Accuracy is a useful metric when errors in predicting all classes are judged to be equally important. It's the case, for example, for object recognition for a domestic robot: a chair is no more important than a table. In the case of the spam/not spam prediction, this probably would not be so. Likely, you would tolerate false negatives more than false positives. Remember, a false positive is when your friend sends you an email, but the model places it in the spam folder and you don't see it. A false negative, a situation in which a spam message gets to the inbox, is less of a problem.

For dealing with the situations in which different classes have different importance, a useful metric is **cost-sensitive accuracy**. First, assign a cost (a positive number) to both types of mistakes: FP and FN. Then compute the counts TP, TN, FP, FN as usual, and multiply the counts for FP and FN by their corresponding costs before calculating the accuracy using Equation 2, above.

Accuracy measures the performance of the model for all classes at once, and it conveniently returns a single number. However, accuracy is not a good performance metric when the data is imbalanced. In an **imbalanced dataset**, examples belonging to some class or a few classes constitute the vast majority, while other classes include very few examples. Imbalanced training data can significantly and adversely affect the model. We will talk more about dealing with the imbalanced data in Section **??** of Chapter 6.

For imbalanced data, a better metric is **per-class accuracy**. First, calculate the accuracy of prediction for each class $\{1, \ldots, C\}$, and then take an average of C individual accuracy measures. For the above confusion matrix of the spam detection problem, the accuracy for the class "spam" is $23/(23+1) = 0.96$, the accuracy for the class "not_spam" is $556/(12+556) = 0.98$. The per-class accuracy is then $(0.96+0.98)/2 = 0.97$.

Per-class accuracy will not be an appropriate model quality measure for a multiclass classification problem where many classes have very few examples (roughly, less than a dozen examples per class). In that case, the accuracy values obtained for the binary classification problems corresponding to these minority classes will not be statistically reliable.

Cohen's kappa statistic is a performance metric that applies to both multiclass and imbalanced learning problems. The advantage of this metric over accuracy is that Cohen's kappa tells you how much better your classification model is performing, compared to a classifier that randomly guesses a class according to the frequency of each class.

Cohen's kappa is defined as:

$$\kappa \stackrel{\text{def}}{=} \frac{p_o - p_e}{1 - p_e},$$

where p_o is called the observed agreement, and p_e is the expected agreement.

Let's look once again at a confusion matrix:

	class1 (predicted)	class2 (predicted)
class1 (actual)	a	b
class2 (actual)	c	d

The observed agreement p_o is obtained from the confusion matrix as,

$$p_o \stackrel{\text{def}}{=} \frac{a + d}{a + b + c + d}.$$

The expected agreement p_e, in turn, is obtained as $p_e \stackrel{\text{def}}{=} p_{\text{class1}} + p_{\text{class2}}$, where,

$$p_{\text{class1}} \stackrel{\text{def}}{=} \frac{a + b}{a + b + c + d} \times \frac{a + c}{a + b + c + d},$$

and

$$p_{\text{class2}} \stackrel{\text{def}}{=} \frac{c + d}{a + b + c + d} \times \frac{b + d}{a + b + c + d}$$

The value of Cohen's kappa is always less than or equal to 1. Values of 0 or less indicate that the model has a problem. While there is no universally accepted way to interpret the values of Cohen's kappa, it's usually considered that values between 0.61 and 0.80 indicate that the model is good, and values 0.81 or higher suggest that the model is very good.

The **ROC curve** (stands for "receiver operating characteristic;" the term comes from radar engineering) is a commonly-used method of assessing classification models. ROC curves use a combination of the **true positive rate** (defined exactly as **recall**) and **false positive rate** (the proportion of negative examples predicted incorrectly) to build up a summary picture of the classification performance.

The true positive rate (TPR) and the false positive rate (FPR) are respectively defined as,

$$\text{TPR} \stackrel{\text{def}}{=} \frac{\text{TP}}{\text{TP}+\text{FN}} \text{ and } \text{FPR} \stackrel{\text{def}}{=} \frac{\text{FP}}{\text{FP}+\text{TN}}$$

ROC curves can only be used to assess classifiers that return a score (or a probability) of prediction. For example, logistic regression, neural networks, and decision trees (and ensemble models based on decision trees) can be assessed using ROC curves.

To draw a ROC curve, you first discretize the range of the score. For instance, you can discretize the range $[0, 1]$ like this: $[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$. Then, use each discrete value as the prediction threshold for your model. For example, if you want to calculate TPR and FPR for the threshold equal to 0.7, you apply the model to each example and get the score. If the score is greater than or equal to 0.7, you predict the positive class. Otherwise, you predict the negative class.

Look at the illustration in Figure 5. It's easy to see that if the threshold equals 0, all our predictions will be positive, so both TPR and FPR will equal 1 (the upper right corner). On the other hand, if the threshold equals 1, then no positive prediction will be possible. Both TPR and FPR will equal 0, which corresponds to the lower-left corner.

The greater the **area under the ROC curve** (AUC), the better the classifier. A classifier with an AUC greater than 0.5 is better than a model that classifies at random. If AUC is lower than 0.5, then something is wrong, most likely a bug in the code or wrong labels in the data. A perfect classifier would have an AUC of 1. In practice, you obtain a good classifier by selecting the value of the threshold that gives TPR close to 1 while keeping FPR near 0.

ROC curves are popular because they are relatively simple to understand. They capture more than one aspect of the classification, by taking both false positives and false negatives into account. They allow the analyst to easily and visually compare different model performances.

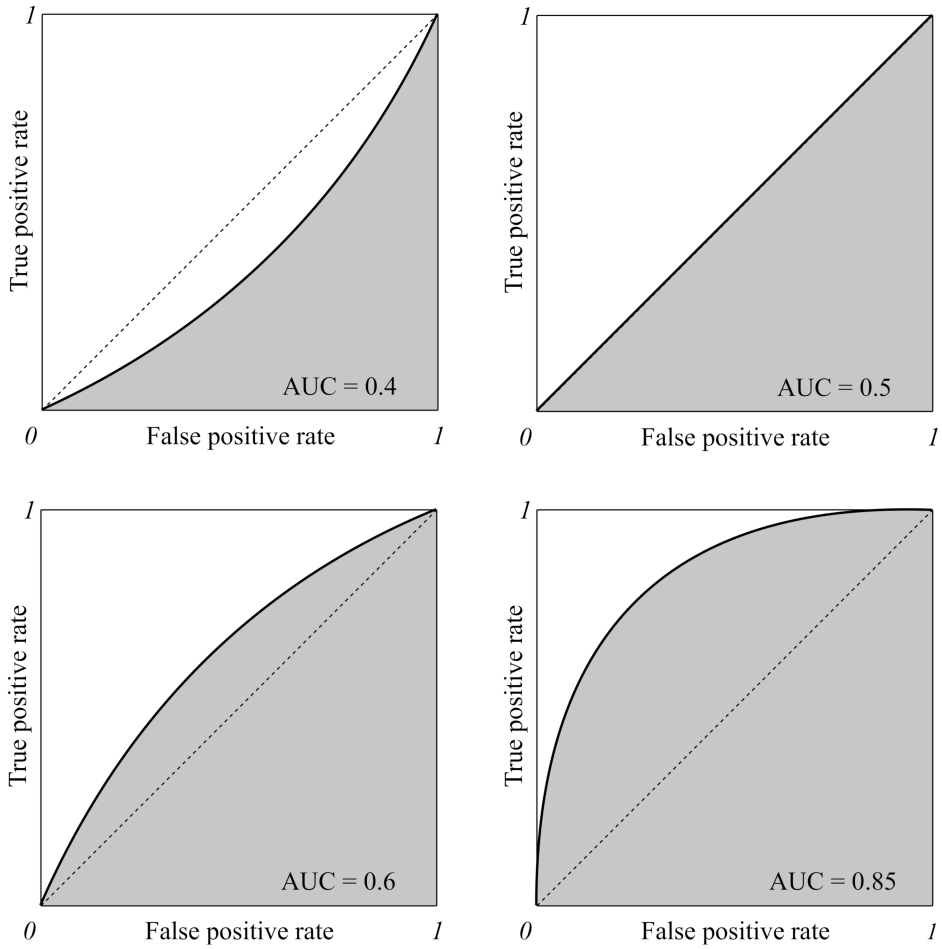

Figure 5: The area under the ROC curve (shown in grey).

5.5.3 Performance Metrics for Ranking

Precision and recall can be naturally applied to the ranking problem. Recall that it's convenient to think of these two metrics as measuring the quality of document search results. Precision is the proportion of relevant documents actually found in the list of all returned documents. The recall is the ratio of the relevant documents returned by the search engine, compared to the total number of the relevant documents that should have been returned.

The drawback of measuring the quality of ranking models with precision and recall is that these metrics treat all retrieved documents equally. A relevant document listed at position k is worth just as much as a relevant document at the top of the list. This is usually not what

we want in document retrieval. When a human looks at search results, the few top-most results matter more than the results shown at the bottom of the list.

Discounted cumulative gain (DCG) is a popular measure of ranking quality in search engines. DCG measures the usefulness, or gain, of a document based on its position in the result list. The gain is accumulated from the top of the result list to the bottom, with the gain of each result discounted at lower positions.

To understand discounted cumulative gain, we introduce a measure called cumulative gain.

Cumulative gain (CG) is the sum of the graded relevance values of all results in a search result list. The CG at a particular rank position p is defined as:

$$\mathrm{CG}_p \stackrel{\text{def}}{=} \sum_{i=1}^{p} rel_i,$$

where rel_i is the graded relevance of the result at position i. Generally, graded relevance reflects the relevance of a document to a query on a scale using numbers, letters, or descriptions (such as "not relevant," "somewhat relevant," "relevant," or "very relevant"). To use it in the above formula, rel_i must be numeric, for example, ranging from 0 (the document at position i is entirely irrelevant to the query) to 1 (the document at position i is maximally relevant to the query). Alternatively, rel_i can be binary: 0 when the document is not relevant to the query, and 1 when relevant. Notice that CG_p is independent of the position each document holds in the ranked result list. It only characterizes the documents ranked up to position p as relevant or irrelevant to the query.

Discounted cumulative gain is based on two assumptions:

1. Highly relevant documents are more useful when appearing earlier in the result list.
2. Highly relevant documents are more useful than marginally relevant documents, while the latter, in turn, are more useful than non-relevant documents.

For a given search result, DCG accumulated at a particular rank position p is often defined as:

$$\mathrm{DCG}_p \stackrel{\text{def}}{=} \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)} = rel_1 + \sum_{i=2}^{p} \frac{rel_i}{\log_2(i+1)}.$$

An alternative formulation of DCG, commonly used in industry and data science competitions such as Kaggle, places a stronger emphasis on retrieving relevant documents:

$$\mathrm{DCG}_p \stackrel{\text{def}}{=} \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{\log_2(i+1)}.$$

For a query, the **normalized discounted cumulative gain** (nDCG), is defined as:

$$\text{nDCG}_p \stackrel{\text{def}}{=} \frac{\text{DCG}_p}{\text{IDCG}_p},$$

where IDCG is the ideal discounted cumulative gain,

$$\text{IDCG}_\text{p} \stackrel{\text{def}}{=} \sum_{i=1}^{|\text{REL}_p|} \frac{2^{rel_i} - 1}{\log_2(i+1)},$$

and REL_p represents the list of the documents relevant to the query in the corpus up to position p (ordered by their relevance). So, REL_p is the ideal ranking, up to position p, that the search engine ranking algorithm (or model) should have returned for the query. The nDCG values for all queries are usually averaged to obtain a performance measure for a search engine ranking algorithm or model.

Let's consider the following example. Let a search engine return a list of documents in response to a search query. We ask a ranker (a human) to judge each document's relevance. The ranker must assign a score from 0 to 3, where 0 means not relevant, 3 means highly relevant, while 1 and 2 mean "somewhere in between." Say the documents appeared in this order:

$$D_1, D_2, D_3, D_4, D_5.$$

Our ranker provides the following relevance scores:

$$3, 1, 0, 3, 2.$$

This means that document D_1 has a relevance of 3, D_2 has a relevance of 1, D_3 has a relevance of 0, and so on. The cumulative gain of this search result, up to position $p = 5$, is,

$$\text{CG}_5 = \sum_{i=1}^{5} rel_i = 3 + 1 + 0 + 3 + 2 = 9.$$

You can see that changing the order of any documents will not affect the value of cumulative gain. Now we will calculate the discounted cumulative gain designed, with the presence of the logarithmic discounting, to have a higher value if highly relevant documents appear early in the result list. To calculate DCG_5, let's calculate the value of the expression $\frac{rel_i}{\log_2(i+1)}$ for each i:

i	rel_i	$\log_2(i+1)$	$\frac{rel_i}{\log_2(i+1)}$
1	3	1.00	3.00
2	1	1.58	0.63
3	0	2.00	0.00
4	3	2.32	1.29
5	2	2.58	0.77

So DCG_5 of this ranking is given by $3.00 + 0.63 + 0.00 + 1.29 + 0.77 = 5.70$.

Now, if we switch the positions of D_1 and D_2, the value of DCG_5 will become lower. This is because a less relevant document is now placed higher in the ranking, while a more relevant document is discounted more by being placed in a lower position.

To calculate the normalized discounted cumulative gain, nDCG_5, we first need to find the value of the discounted cumulative gain of the ideal ordering, IDCG_5. The ideal ordering, according to the relevance scores, is $3, 3, 2, 1, 0$. The value of IDCG_5 is then equal to $3.00 + 1.89 + 1.00 + 0.43 + 0.0 = 6.32$. Finally, nDCG_5 is given by,

$$\text{nDCG}_5 = \frac{\text{DCG}_5}{\text{IDCG}_5} = \frac{5.70}{6.32} = 0.90.$$

To obtain nDCG for a collection of test queries and the corresponding lists of search results, we average the values of nDCG_p obtained for each individual query. The advantage of using the normalized discounted cumulative gain over other measures is that the values of nDCG_p obtained for different values of p are comparable. This property is useful when the number p of relevance scores, provided by the rankers, is different for different queries.

Now that we have a performance metric, we can use it to compare models in the process known as hyperparameter tuning.

5.6 Hyperparameter Tuning

Hyperparameters play an important role in the model training process. Some hyperparameters influence the speed of training, but the most important hyperparameters control the two tradeoffs: bias-variance and precision-recall.

Hyperparameters aren't optimized by the learning algorithm itself. The data analyst "tunes" hyperparameters by experimenting with combinations of values, one per hyperparameter. Each machine learning model and each learning algorithm have a unique set of hyperparameters. Furthermore, every step in your entire machine learning pipeline, data pre-processing, feature extraction, model training, and making predictions, can have its own hyperparameters.

For example, in data pre-processing, the hyperparameters could specify whether to use data-augmentation or using which technique to fill missing values. In feature engineering,

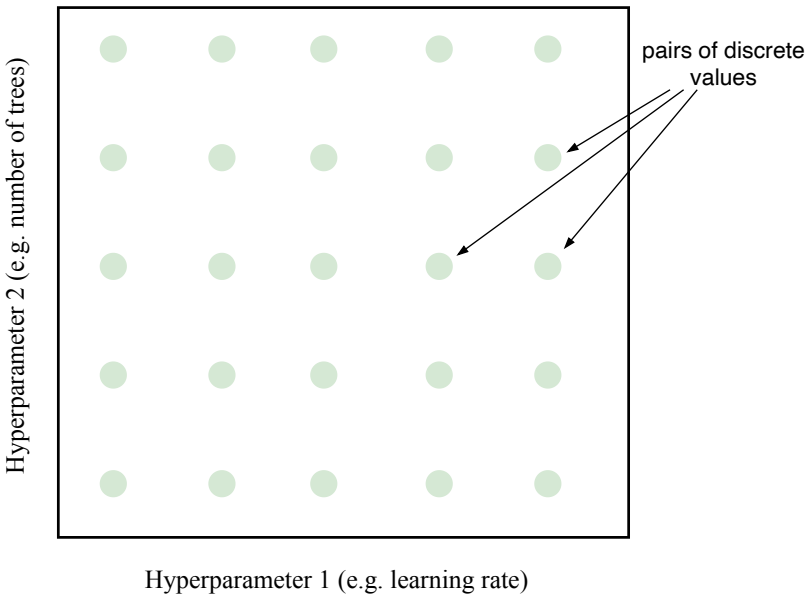

Figure 6: Grid search for two hyperparameters: each green circle represents a pair of hyperparameter values.

a hyperparameter could define which feature selection technique to apply. When making predictions with a model that returns a score, a hyperparameter could specify the decision threshold for each class.

Below, we consider several popular **hyperparameter tuning techniques**.

5.6.1 Grid Search

Grid search is the simplest hyperparameter tuning technique. It's used when the number of hyperparameters and their range is not too large.

We explain it for the problem of tuning two numerical hyperparameters. The technique consists of discretizing each of the two hyperparameters, and then evaluating each pair of discrete values, as shown in Figure 6.

Each evaluation consists of:

1) configuring a pipeline with a pair of hyperparameter values,
2) applying the pipeline to the training data and training a model, and
3) computing the performance metric for the model on the validation data.

The pair of hyperparameter values that results in the best performing model is then selected

for training the final model.

The Python code below uses grid search with cross-validation.[2] It shows how to optimize the hyperparameters of the simple two-stage scikit-learn pipeline considered above:

```
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.model_selection import GridSearchCV

# Define a pipeline
pipe = Pipeline([('dim_reduction', PCA()), ('model_training', SVC())])

# Define hyperparamer values to try
param_grid = dict(dim_reduction__n_components=[2, 5, 10], \
model_training__C=[0.1, 10, 100])

grid_search = GridSearchCV(pipe, param_grid=param_grid)

# Make a prediction
pipe.predict(new_example)
```

In the above example, we use grid search to try the values $[2, 5, 10]$ of the hyperparameter `n_components` of PCA, and the values $[0.1, 10, 100]$ of the hyperparameter `C` of SVM.

Trying multiple combinations of hyperparameters could be time-consuming for large datasets. There are more efficient techniques, such as random search, coarse-to-fine search, and Bayesian hyperparameter optimization.

5.6.2 Random Search

Random search differs from grid search in that you do not provide a discrete set of values to explore for each hyperparameter. Instead, you provide a statistical distribution for each hyperparameter from which values are randomly sampled. Then set the total number of combinations you want to evaluate, as shown in Figure 7.

[2] We talk about cross-validation in Subsection 5.6.5.

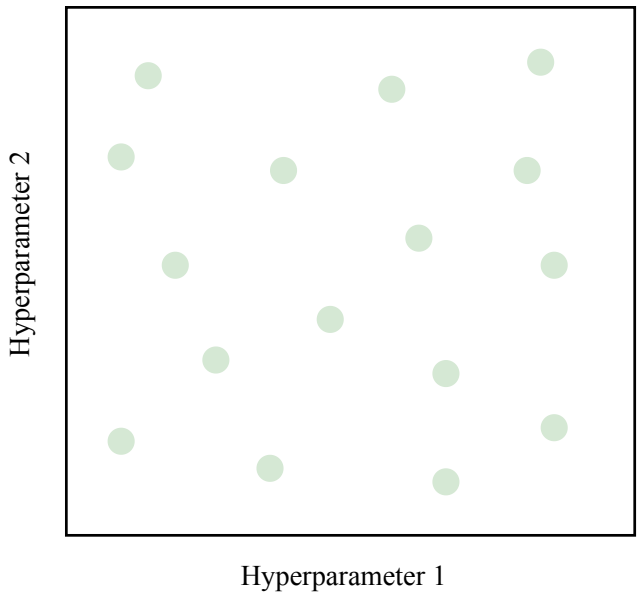

Figure 7: Random search for two hyperparameters and 16 pairs to test.

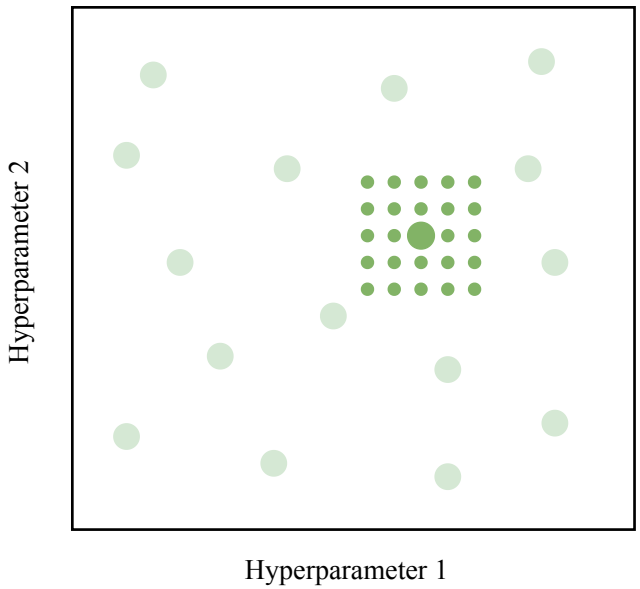

Figure 8: Coarse-to-fine search for two hyperparameters: 16 coarse random search pairs to test and one grid search in the region of the highest value found using the random search.

5.6.3 Coarse-to-Fine Search

In practice, analysts often use a combination of grid search and random search called **coarse-to-fine search**. This technique uses a coarse random search to first find the regions of high potential. Then, using a fine grid search in these regions, one finds the best values for hyperparameters, as shown in Figure 8.

You can decide to only explore one high-potential region or several such regions, depending on the available time and computational resources.

5.6.4 Other Techniques

Bayesian techniques differ from random and grid searches in that they use past evaluation results to choose the next values to evaluate. In practice, this allows Bayesian hyperparameter optimization techniques to find better values of hyperparameters in less time.

There are also gradient-based techniques, evolutionary optimization techniques, and other algorithmic hyperparameter tuning methods. Most modern machine learning libraries implement one or more such techniques. There are also hyperparameter tuning libraries that can be used to tune hyperparameters of virtually any learning algorithm, including the algorithms you programmed yourself.

5.6.5 Cross-Validation

Grid search and other techniques of hyperparameter tuning discussed above are used when you have a good-sized validation set.[3] When you don't, a common technique of model evaluation is **cross-validation**. Indeed, when you have few training examples, it could be prohibitive to have both validation and test sets. You would prefer to use more data to train the model. In such a case, you should only split your data in two: a training and a test set. Then use cross-validation on the training set to simulate a validation set.

Cross-validation works as follows. First, you fix the values of the hyperparameters to evaluate. Then you split your training set into several subsets of the same size. Each subset is called a fold. Typically, **five-fold cross-validation** is used, and you randomly split your training data into five folds: $\{F_1, F_2, \ldots, F_5\}$. Each F_k, $k = 1, \ldots, 5$, contains 20% of your training data. Then you train five models in a specific manner. To train the first model, f_1, you use all examples from folds F_2, F_3, F_4, and F_5 as the training set, and the examples from F_1 as the validation set. To train the second model, f_2, you use the examples from folds F_1, F_3, F_4, and F_5 to train, and the examples from F_2 to validate. You continue training models f_k iteratively[4] for all remaining folds, and compute the value of the metric of interest on each

[3] A decent validation set contains at least a hundred examples, and each class in the set is represented by at least a couple of dozen examples.

[4] The process of cross-validation is easier to illustrate as an iterative process; though, one can, of course, build all five models F_1 to F_5 in parallel.

validation set, from F_1 to F_5. Then you average the five values of the metric to get the final value. More generally, in n-fold cross-validation, you train model f_n on all folds, except for the n-th fold F_n.

You can use grid search, random search, or any other such technique with cross-validation to find the best values of hyperparameters. Once you have found those values, you typically use the entire training set to train the final model by using the best values of hyperparameters found via cross-validation. Finally, you assess the final model using the test set.

While finding the best values of hyperparameters is tempting, it might be unrealistic to try all of them. Remember that time is precious, and perfect is often an enemy of good. Deploy a "good enough" model to production, then continue to run the search of the ideal values for hyperparameters (for weeks if it is what it takes).

Now, let's consider the challenge of training a shallow model.

5.7 Shallow Model Training

Shallow models make predictions based directly on the values in the input feature vector. Most popular machine learning algorithms produce shallow models. The only kind of deep models commonly used are deep neural networks. We consider a strategy to train them in Section **??** of the next chapter.

5.7.1 Shallow Model Training Strategy

A typical model training strategy for shallow learning algorithms looks as follows:

1. Define a performance metric P.
2. Shortlist learning algorithms.
3. Choose a hyperparameter tuning strategy T.
4. Pick a learning algorithm A.
5. Pick a combination H of hyperparameter values for algorithm A using strategy T.
6. Use the training set and train a model M using algorithm A parametrized with hyperparameter values H.
7. Use the validation set and calculate the value of metric P for model M.
8. Decide:
    a. If there are still untested hyperparameter values, pick another combination H of hyperparameter values using strategy T and go back to step 6.
    b. Otherwise, pick a different learning algorithm A and go back to step 5, or proceed to step 9 if there are no more learning algorithms to try.
9. Return the model for which the value of metric P is maximized.

In the above strategy, step 1, you define the performance metric for your problem. As we have seen in Section 5.5, it is a mathematical function or a subroutine that takes a model and a dataset as input, and produces a numerical value that reflects how well the model works.

In step 2, you choose candidate algorithms and then shortlist some of them (usually, two or three). To do that, you can use the selection criteria considered in Section 5.3.

In step 3, you choose a hyperparameter tuning strategy. It is a sequence of actions that generates the combinations of hyperparameter values to test. We have considered several hyperparameter-tuning strategies in Section 5.6.

5.7.2 Saving and Restoring the Model

Once you trained a model or a pipeline, you must save it to a file so that it can be deployed to production and then used for scoring. Both model and pipeline can be serialized. In Python, **Pickle** is typically used for serialization (saving) and deserialization (restoring) of objects. In R, it's RDS.

Here's how model serialization/deserialization is done in Python:

```
import pickle
from sklearn.svm import SVC
from sklearn import datasets

# Prepare data
X, y = datasets.load_iris(return_X_y=True)

# Instantiate the model
model = SVC()

# Train the model
model.fit(X, y)

# Save the model to file
pickle.dump(model, open("model_file.pkl", "wb"))

# Restore the model from file
restored_model = pickle.load(open("model_file.pkl", "rb"))

# Make a prediction
prediction = restored_model.predict(new_example)
```

A similar code in R would look as follows:

```
library("e1071")

# Prepare data
attach(iris)
X <- subset(iris, select=-Species)
```

```
y <- Species

# Train the model
model <- svm(X,y)

# Save the model to file
saveRDS(model, "./model_file.rds")

# Restore the model from file
restored_model <- readRDS("./model_file.rds")

# Make a prediction
prediction <- predict(restored_model, new_example)
```

Now, let's talk about the particularities of the model training process that analysts must take care of in practice to produce an optimal model.

5.8 Bias-Variance Tradeoff

Developing a model includes both searching for an optimal algorithm, as well as finding the best performing hyperparameters. Tweaking the hyperparameters actually controls two tradeoffs. We already discussed the first one: the precision-recall tradeoff. The second one, equally important, is the **bias-variance tradeoff**.

5.8.1 Underfitting

The model is said to have a **low bias** if it ablely predicts the training data labels. If the model makes too many mistakes on the training data, we say that it has a **high bias**, or that the model **underfits** the training data. There could be several reasons for underfitting:

- the model is too simple for the data (for example linear models often underfit);
- the features are not informative enough;
- you regularize too much (we talk about regularization in the next section).

An example of underfitting in regression is shown in Figure 9 (left). The regression line doesn't repeat the bends of the line to which the data seemingly belongs. The model oversimplifies the data. The possible solutions to the problem of underfitting include:

- trying a more complex model,
- engineering features with higher **predictive power**,
- adding more training data, when possible, and
- reducing regularization.

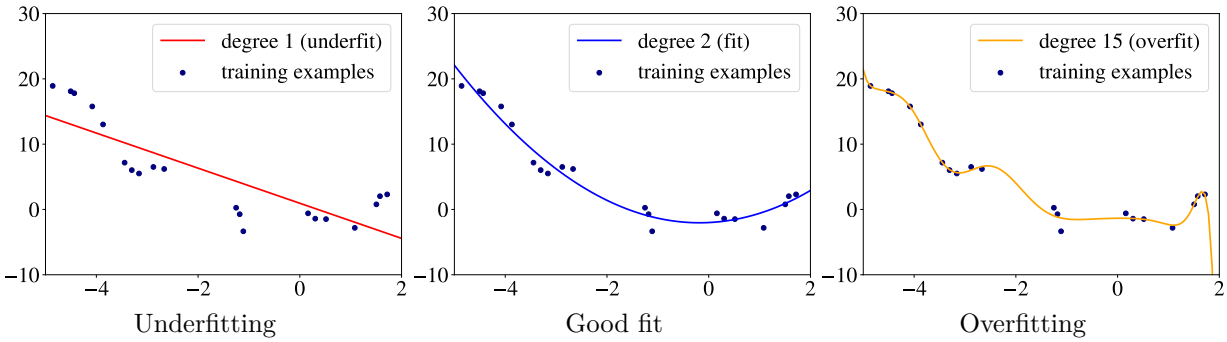

Figure 9: Examples of underfitting (linear model), good fit (quadratic model), and overfitting (polynomial of degree 15).

5.8.2 Overfitting

Overfitting is another problem a model can exhibit. The model that overfits usually predicts the training data labels very well, but works poorly on the holdout data.

An example of overfitting in regression is shown in Figure 9 (right). The regression line predicts almost perfectly the targets for almost all training examples, but will likely make significant errors on new data if you decide to use it for predictions.

You will find another name for overfitting in the literature: **high variance**. The model is unduly sensitive to small fluctuations in the training set. If you sampled the training data differently, the result would be a significantly different model. These overfitting models perform poorly on the holdout data, since holdout and training data are sampled from the dataset independently of one another. So, the small fluctuations in the training and holdout data are likely to be different.

Several reasons can lead to overfitting:

- the model is too complex for the data. Very tall decision trees or a very deep neural network often overfit;
- there are too many features and few training examples; and
- you don't regularize enough.

Several solutions to overfitting are possible:

- use a simpler model. Try linear instead of polynomial regression, or SVM with a linear kernel instead of **radial basis function** (RBF), or a neural network with fewer layers/units;[5]

[5]While reducing the number of model parameters is generally recommended to reduce overfitting and improve the generalization of the model, the phenomenon of **deep double descent** sometimes proves otherwise. The phenomenon was observed in various architectures, including **CNN** and **transformers**: validation performance first improves, then gets worse, and then improves again with increasing model size.

- reduce the dimensionality of examples in the dataset;
- add more training data, if possible; and,
- regularize the model.

5.8.3 The Tradeoff

In practice, by trying to reduce variance, you increase bias, and vice versa. In other words, reducing overfitting leads to underfitting, and the other way around. This is called the **bias-variance tradeoff**: by trying too hard to build a model that performs perfectly on the training data, you end up with a model that performs poorly on the holdout data.

While many factors determine whether the model performs well on the training data, the most important factor is the complexity of the model. A sufficiently complex model will learn to memorize all training examples and their labels and, thus, will not make prediction errors when applied to the training data. It will have low bias. However, a model relying on memorization will not be able to correctly predict labels of previously unseen data. It will have high variance.

As the model complexity grows, the typical evolution of the average prediction error of a model when applied to the training and holdout data is shown in Figure 10.

The zone you would like to be in is the "zone of solutions," the light-blue rectangle where both bias and variance are low. Once in this zone, you can fine-tune the hyperparameters to reach the needed precision-recall ratio, or optimize another model performance metric appropriate for your problem.

To reach the zone of solutions, you can either,

- move to the right by increasing the complexity of the model, and, by so doing, reducing its bias, or
- move to the left by regularizing the model to reduce variance by making the model simpler (we talk about regularization in the next section).

If you work with shallow models, like linear regression, you can increase the complexity by switching to higher-order polynomial regression. Similarly, you can increase the depth of the decision tree, or use polynomial or RBF kernels, in support vector machine (SVM) instead of the linear kernel. Ensemble learning algorithms, based on the idea of boosting, allow bias reduction by combining several (usually, hundreds of) high-bias "weak" models.

As of July 2020, we don't yet fully understand why it happens.

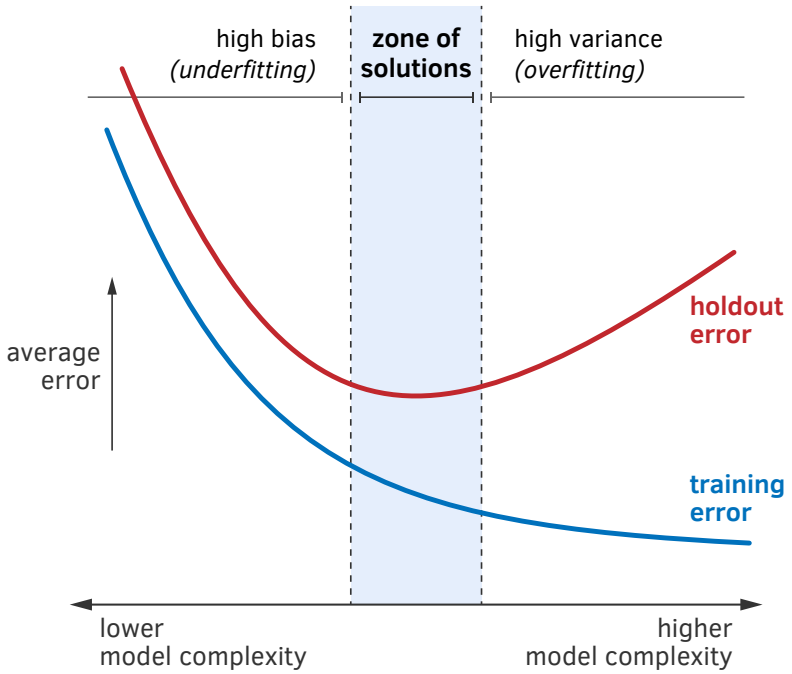

Figure 10: Bias-variance tradeoff.

If you work with neural networks, you can increase the model's complexity by increasing its size: the number of units per layer, and the number of layers. Training a neural network model longer (i.e., for more epochs) also usually results in lower bias. The advantage of using neural networks, with respect to the bias-variance tradeoff, is that you can slightly increase the size of the network and observe a slight decrease in bias. Most popular shallow models and the associated learning algorithms cannot provide you such flexibility.

If, by increasing the complexity of your model, you find yourself in the right-hand side of the graph in Figure 10, you have to reduce the variance of the model. The most common way to do that is to apply regularization.

5.9 Regularization

Regularization is an umbrella term for methods that force a learning algorithm to train a less complex model. In practice, it leads to higher bias, but significantly reduces the variance.

The two most widely used types of regularization are **L1** and **L2 regularization**. The idea is quite simple. To create a regularized model, we modify the objective function. This is the expression optimized by the learning algorithm when training the model. Regularization adds a penalizing term whose value is higher when the model is more complex.

For simplicity, we will illustrate regularization using linear regression, but same principle can be applied to a wide variety of models.

Let $\mathbf{x}$ be a two-dimensional feature vector $\left[x^{(1)}, x^{(2)}\right]$. Recall the linear regression objective:

$$\min_{w^{(1)},w^{(2)},b} \left[\frac{1}{N} \times \sum_{i=1}^{N} (f_i - y_i)^2 \right], \tag{3}$$

In the above equation, $f_i \stackrel{\text{def}}{=} f(\mathbf{x}_i)$, and f is the equation of the regression line. The equation of the linear regression line f will have the form $f = w^{(1)}x^{(1)} + w^{(2)}x^{(2)} + b$. The learning algorithm will deduce the values of parameters $w^{(1)}$, $w^{(2)}$, and b from the training data by minimizing the objective. A model is considered less complex if some of the parameters $w^{(\cdot)}$ are close to or equal to zero.

5.9.1 L1 and L2 Regularization

An L1-regularized objective in Equation 3 looks like this:

$$\min_{w^{(1)},w^{(2)},b} \left[C \times \left(|w^{(1)}| + |w^{(2)}| \right) + \frac{1}{N} \times \sum_{i=1}^{N} (f_i - y_i)^2 \right], \tag{4}$$

where C is a **hyperparameter** that controls the importance of regularization. If we set C to zero, the model becomes a standard non-regularized linear regression model. On the other hand, if we set C to a high value, the learning algorithm will try to set most $w^{(\cdot)}$ to a value close or equal to zero to minimize the objective. The model will become very simple, which can lead to underfitting. The role of the data analyst is to find such a value of the hyperparameter C that doesn't increase the bias too much, but reduces the variance to a level reasonable for the problem at hand.

An L2-regularized objective in our two-dimensional setting looks like this:

$$\min_{w^{(1)},w^{(2)},b} \left[C \times \left((w^{(1)})^2 + (w^{(2)})^2 \right) + \frac{1}{N} \times \sum_{i=1}^{N} (f_i - y_i)^2 \right], \tag{5}$$

In practice, L1 regularization produces a **sparse model**, assuming the value of hyperparameter C is great enough. This is a model where most of its parameters equal exactly zero. So, as discussed in the previous chapter, L1 implicitly performs **feature selection** by

deciding which features are essential for prediction, and which are not. This property of L1 regularization is useful when we want to increase model **explainability**. However, if our goal is to maximize the model performance on the holdout data, then L2 usually gives better results.

In the literature, you will also see the names **lasso** for L1 and **ridge regularization** for L2.

5.9.2 Other Forms of Regularization

L1 and L2 regularization methods can be combined in what's called **elastic net regularization**.

In addition to being widely used with linear models, L1 and L2 are often used with neural networks and many other types of models that directly minimize an objective function.

Neural networks can also benefit from two other regularization techniques: **dropout** and **batch-normalization**. There are also non-mathematical methods that have a regularization effect: **data augmentation** and **early stopping**. We will talk about these techniques in more detail in the next chapter, when we consider training neural networks.

5.10 Summary

Before starting to work on the model, you should make several checks and decisions. First, make sure that the data conforms to the schema, as defined by the schema file. Then, define an achievable level of performance, and choose a performance metric. Ideally, it should represent the model performance as a single number. Furthermore, it is important to establish a baseline that provides a reference point to compare your machine learning models. Finally, split your data into three sets: train, validation, and test.

Most modern implementations of classification learning algorithms require that the training examples have numerical labels, so you typically must transform your labels into numerical vectors. Two popular ways to do that are one-hot encoding (for binary and multiclass problems) and bag-of-words (for multi-label problems).

To choose a machine learning algorithm that would work best for your problem, ask yourself the following questions:

- Do the model's predictions have to be explainable to a non-technical audience? If yes, you would prefer using less accurate, but more explainable algorithms, such as kNN, linear regression, and decision tree learning.
- Can your dataset be fully loaded into the RAM of your laptop or server? If not, you would prefer incremental learning algorithms.
- How many training examples do you have in your dataset, and how many features does each example have? Some algorithms, including those used for training neural networks and random forests, can handle a huge number of examples and millions of features. Others are relatively modest in their capacity.

- Is your data linearly separable, or can it be modeled using a linear model? If yes, SVM with the linear kernel, linear and logistic regression, can be good choices. Otherwise, deep neural networks or ensemble models might work better.
- How much time is a learning algorithm allowed to use to train a model? Neural networks are known to be slow to train. Simple algorithms like linear and logistic regression, or decision trees are much faster.
- How fast must the scoring perform in production? Models like SVM, linear and logistic regression, as well as not very deep feedforward neural networks, are extremely fast at the prediction time. The scoring using deep and recurrent neural networks, as well as gradient boosting models, is slower.

If you don't want to guess the best algorithm for your problem, a recommended approach is to spot-check several algorithms, and then test them on the validation set as a hyperparameter.

A typical way to know how good is the model, is to calculate the value of a performance metric on the holdout data. There are performance metrics defined for classification and regressions models, as well as for ranking models.

Tweaking the values of hyperparameters controls two tradeoffs: precision-recall and bias-variance. By varying the complexity of the model, we can reach the so-called "zone of solutions," a situation in which both bias and variance of the model are relatively low. The solution that optimizes the performance metric is usually found inside that zone.

Regularization is an umbrella term for methods that force the learning algorithm to build a less complex model. In practice, that often leads to slightly higher bias, but significantly reduces the variance. Two popular techniques of regularization are L1 and L2. In addition, neural networks benefit from two other regularization techniques: dropout and batch normalization.

Most modern machine learning packages and frameworks support the notion of a pipeline. A pipeline is a sequence of transformations the training data undergoes before it becomes a model. In a pipeline, each stage applies some transformation to the input it receives. Every stage receives the output of the previous stage, except for the first stage. The first stage receives the training dataset as input. The pipeline can be saved to a file similar to saving a model. It can be deployed to production and used to generate predictions.

Hyperparameters aren't optimized by the learning algorithm itself. A data analyst must "tune" hyperparameters by experimenting with different combinations of values. Grid search is the simplest and the most widely used hyperparameter tuning technique. It consists of discretizing the values of hyperparameters, and trying all combinations of values by 1) training a model for each combination of hyperparameters, and 2) computing the performance metric by applying each trained model to the validation set.

A decent validation set contains at least a hundred examples, and each class in the set is represented by at least a couple of dozen examples. When you don't have a decent validation set to tune your hyperparameters, you can use cross-validation.