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.

5 Supervised Model Building

Model building (or modeling) is one of the most overrated activities in machine learning. While it's clear that without it, no model will be built, on average a machine learning engineer spends on this activity 5-10% of their time on the project, if at all.

As I already mentioned, successful data collection and preparation, as well as feature engineering, are more important for the success of a machine learning project than modeling. In the vast majority of practical cases, 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 the chapters you skipped, they are really important.

5.1 Before You Start Working on the Model

Before you start working on the model, you have to make several checks and decisions.

5.1.1 Validate Schema Conformity

First of all, you have to make sure that the data you will work with is conform to the schema (as defined by the **schema file**). Even if it was you who prepared the data for machine learning, it's still probable that the data you prepared and the data you use to build the model is not exactly the same. This difference can be explained by various factors, but the most probable are the following three:

- the method you used to persist the data on the hard drive or to the database contains an error;
- the method you use to read the data from where it was persisted to contains an error;
- someone else could change the data or the schema without informing you.

If the data does not conform to the schema, the situation has to be considered as an error, the same way as an error in the programming code is detected. The reason for the difference between the actual data and its description in the schema has to be identified, the code that modified the data has to be fixed and, if needed, the entire data collection and preparation pipeline has to be run from scratch (as we discussed at the end of Chapter 3).

5.1.2 Define an Achievable Level of Performance

Define an achievable level of performance. Here are some guidelines:

- if a human can label examples without too much thinking (without using math or complex logic derivations that require a pen and a paper), then you can hope to achieve human-level performance with your model;

- if the information that is 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 very 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 already have a computer program that solves the same problem, you can expect to get at least as good performance; often the performance of a machine learning model can be become much better, as more labeled data comes in;
- if you are capable of observing a similar, but not exactly the same system, you can expect to get a similar, but not exactly the same level of performance of your machine learning model.

5.1.3 Choose a Performance Metric

We will talk about assessing the model performance later, but for now, just know that there can be several ways to estimate the level of performance of a model. There's no one best way you can always use, independently of the data and the problem you want to solve. On the other hand, it is recommended to choose one and only one **performance metric** before you start working on the model and then compare different models to one another and track the overall progress of the machine learning project by using this one metric.

Later in this chapter, you will read about the most popular and handy model performance metrics as well as the approaches allowing combining multiple metrics to obtain one number.

5.1.4 Split Data Into Three Set

As we already discussed, three sets are typically needed to build a solid model. The first set, training set, is used to train the model; it is the data the machine learning algorithm "sees". The second set, the validation set, is not seen by the machine learning algorithm; it is used by the data analyst to compare the performance of different machine learning algorithms (or of the same algorithm configured with different values of hyperparameters) on new data. The remaining test set is also not seen by the learning algorithm and is used at the end of the project to evaluate and report the performance of the model the best performing on the validation set.

The process of splitting the entire dataset into three sets is described in Chapter 3. Here, I only reiterate several important pieces of that process:

- the validation and test set have to 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;
- choose the validation and test sets from a distribution that looks much like the data you expect to observe in the production environment in the future once the model is deployed; this may not be the same as your training data's distribution.

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. However, in practice, it's not uncommon to have lots of training examples that look not entirely exactly the same as the data you will deal with in production; however, those examples may be available in abundance and are inexpensive to get. This is known to be the problem of **distribution shift** and the analyst may or may not know about it. If you are aware of the problem of distribution shift, you would rather put all those easily available but dissimilar examples into your training set, but you would avoid using them in the validation and test set because you would prefer to be sure that when you evaluate the model, you evaluate it against the data that is similar to your production setting data. Doing otherwise would result in a suboptimal model being chosen and overly optimistic values of the performance metric during model testing.

The distribution shift (that is the difference of statistical properties of the training and holdout sets) can be a hard problem to tackle. As I mentioned above, using a different data distribution for training could be a conscious choice of the data analyst because of data availability. However, it's also possible that the analyst is unaware that the statistical properties of the training and development sets are different. This situation is especially frequent when the model is frequently updated after being deployed in production: the properties of the data used to train the model and those of the additional data used to update and test the model can diverge over time. Later in this chapter, I will give some guidance on how to handle the distribution shift.

5.2 Preconditions for Supervised Learning

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

1. You have a labeled dataset.
2. You have split the dataset into three subsets: training, validation, and testing.
3. To avoid data leakage, you engineered features using only the training data.
4. You converted all examples into numerical feature vectors.

5.3 Representing Labels for Machine Learning

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

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

If an implementation of the learning algorithm requires numerical labels, which is often the case when you work with neural networks, you have to transform your lables to numbers. In

the case of **multiclass classification** (that is when the model has to predict only one class given an input feature vector), one-hot encoding is typically used to convert labels to binary vectors. For example, let's say you have a multiclass classification problem, your classes are $\{\text{dog}, \text{cat}, \text{other}\}$ 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:

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

Then your data, after you convert categorical labels into binary vectors, will become:

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]

In a **multi-label classification** problem, for each input, the model has to predict several labels 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 look like follows:

Image	Label
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 using bag of words, your data will become:

Image	Label
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 in which the learning algorithm expects the labels.

5.4 Selecting the Learning Algorithm

Choosing a machine learning algorithm can be a difficult task. If you have a lot of time, you can try all of them. However, usually the time you have to solve a problem is limited. 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.

Explainability

Does your model have to be explainable to a non-technical audience? Most very accurate learning algorithms are so-called "black boxes." They are capable of producing models that make very few errors, but why a model made a specific prediction could be very hard to understand and even harder to explain. Examples of such models are **deep neural networks** and **ensemble models**.

On the other hand, **kNN**, **linear regression**, and **decision tree learning** algorithms produce models that are not always the most accurate, however, the way they make their prediction is very straightforward.

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 adding more data gradually.

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 **gradient boosting**, can handle a huge number of examples and millions of features. Others, like **SVM**, can be very modest in their capacity.

Nonlinearity of the data

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 algorithms might work better.

Training speed
How much time is a learning algorithm allowed to use to build a model? Neural networks are known to be 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 forests, benefit from the availability of multiple CPU cores, so their model building time can be significantly reduced on a machine with dozens of cores.

Prediction speed
How fast does the model have to 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, and (some types of) neural networks, are extremely fast at the 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 it on the **validation set** like a hyperparameter. We talk about hyperparameter tuning later in this chapter. Alternatively, if you use scikit-learn, you could try their algorithm selection diagram shown in Figure 1.

5.5 Shallow Model Building Strategy

Shallow models make predictions based directly on the values in the input feature vector. Most popular machine learning algorithms produce shallow models. The only known deep models are deep neural networks.

The model-building strategies for shallow and deep models are slightly different. For shallow learning algorithms, the model building strategy 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 using T.
6. Use the training set and build the model M using A parametrized with H.
7. Use the validation set and calculate the value of the matric P for the model M.
8. Decide:
    1. If there are still untested hyperparameter values, pick another combination H of hyperparameter values using T and go to step 6.
    2. Otherwise, pick a different learning algorithm A and go to step 5, or go to step 9 if there are no more learning algorithms to try.

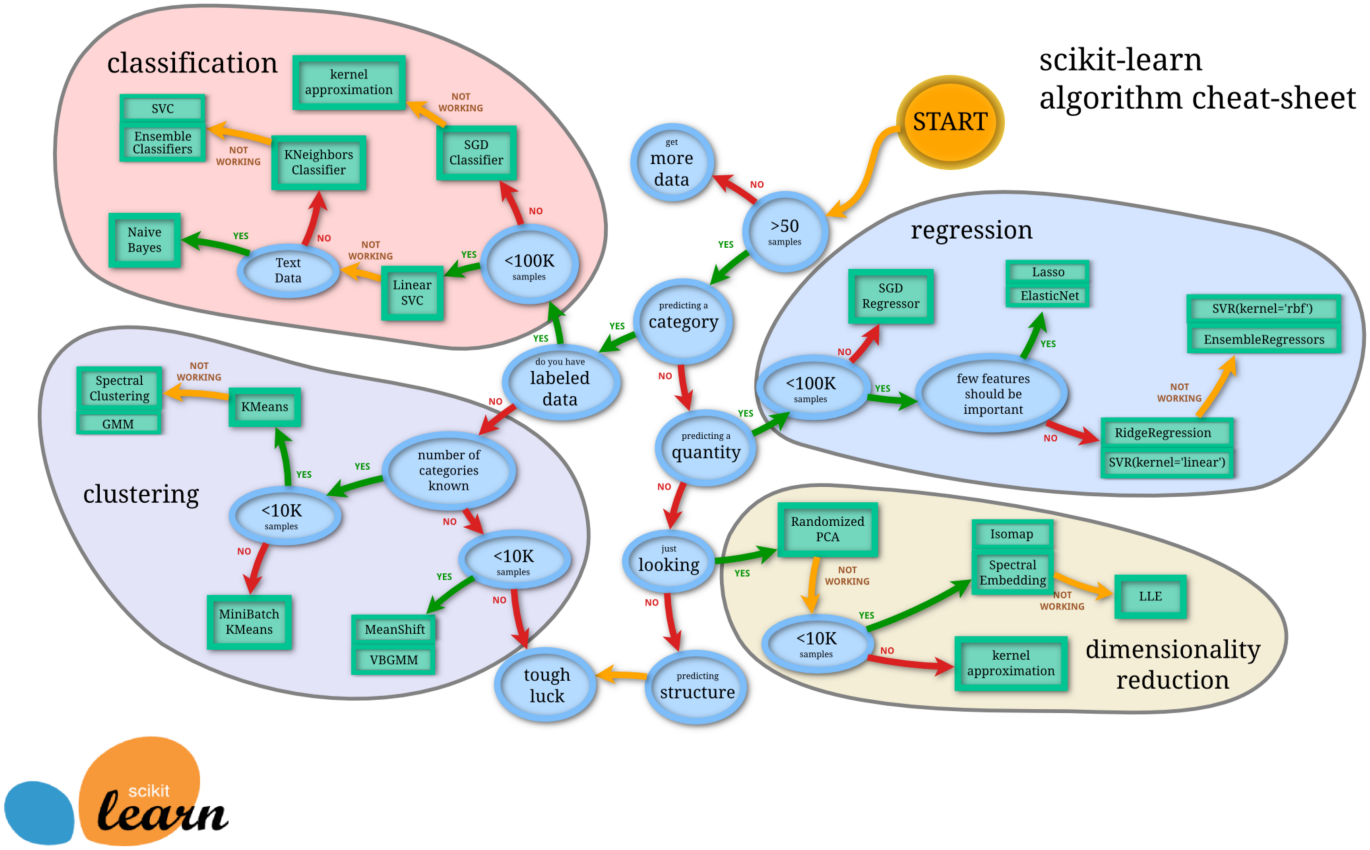


Figure 1: Machine learning algorithm selection diagram for scikit-learn.

9. Return the model for which the value of metric P is maximized.

In the above strategy, on step 1 you define the performance metric for your problem. The **performance metric** is a mathematical function or a subroutine that takes the model and the validation data as input and produces a numerical value. Ideally, the value produced by the metric is higher if the model performs better on the validation data. What "better" means depends on the problem you want to solve. We will consider performance metrics later in this chapter.

On step 2, you choose candidate algorithms and then shortlist two or three. To choose the candidate algorithms, you can use the selection criteria considered in the previous section. If you have more than three candidate algorithms then apply each of them to the training data by using the default values of hyperparameters and select two or three algorithms that have produced models with the highest values of the metric.

On step 3, you choose a **hyperparameter tuning** strategy. A hyperparameter tuning strategy is a sequence of actions that allows generating the combinations of values of hyperparameters to test. The values of hyperparameters can significantly affect the properties of the model that will be produced by the learning algorithm. We consider several hyperparameter tuning strategies later in this chapter.

5.6 Multiclass Classification

Although many classification problems can be defined using two classes, some are defined with more than two classes, which requires adaptations of machine learning algorithms.

In **multiclass classification**, the label can be one of C classes: $y \in \{1, \ldots, C\}$. Many machine learning algorithms are binary; SVM is an example. Some algorithms or models, like logistic regression, decision trees (and, as a consequence, the ensemble methods based on decision trees), k Nearest Neighbors, and neural networks, can naturally be extended to handle multiclass problems.

SVM cannot be naturally extended to multiclass problems. Other algorithms can be implemented more efficiently in the binary case. One common strategy for converting a binary classification learning algorithm to a multiclass problem is called **one versus rest**. The idea is to transform a multiclass problem into C binary classification problems and build C binary classifiers. For example, if we have three classes, $y \in \{1, 2, 3\}$, we create copies of the original datasets and modify them. In the first copy, we replace all labels not equal to 1 by 0. In the second copy, we replace all labels not equal to 2 by 0. In the third copy, we replace all labels not equal to 3 by 0. Now we have three binary classification problems where we have to learn to distinguish between labels 1 and 0, 2 and 0, and 3 and 0.

Once we have the three models, to classify the new input feature vector $\mathbf{x}$, we apply the three models to the input, and we get three predictions. We then pick the prediction of a non-zero class which is *the most certain*. Remember that in logistic regression, the model returns not a label but a score (between 0 and 1) that can be interpreted as (and truly is) the probability

that the label is positive. We can also interpret this score as the certainty of prediction. In SVM, the analog of certainty is the distance d from the input $\mathbf{x}$ to the decision boundary.

The larger the distance, the more certain is the prediction. Most learning algorithms either can be naturally converted to a multiclass case, or they return a score we can use in the one versus rest strategy. Later in this chapter, I will show how, by using a technique called **model calibration**, we can transform any model capable of returning a score into a model returning a probability.

5.7 Assessing Model Performance

Once a learning algorithm parametrized with specific values of hyperparameters produces a model of the training data, we want to know how good the model is. The most common way to get that knowledge is to calculate the value of a **performance metric** on the holdout data.

The holdout data consists of the examples that the learning algorithm didn't see, so if our model performs well on predicting the labels of the examples from a holdout set, we say that our model **generalizes well** or, simply, that it's good.

However, what does it mean to say that a model performs well? If we want to select the best model, we have to be able to effectively compare models. For this purpose, machine learning engineers use performance metrics.

5.7.1 Mean Squared Error

For regression, the assessment of the model is quite simple. The performance metric used to quantify the performance of the model is the same as the cost function: **mean squared error** (MSE), defined as,

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

where f is the model that takes a feature vector $\mathbf{x}$ as input and outputs a class y, and i ranging from 1 to N is an example from a dataset used to compute the value of the metric.

A well-fitting regression model results in predicted values close to the observed data values. The **mean model**, which always predicts the average of the labels in the training data, generally would be used if there were no informative features. The fit of a regression model being assessed should, therefore, be better than the fit of the mean model. In this case, the mean model acts as a **baseline**. If the MSE of the regression model is bigger than the MSE of the baseline, then we have a problem with our regression model. Besides **overfitting**, which we will consider later in this chapter, the reason of the below-the-baseline performance

of the model could be that the problem was defined with an error or the programming code contains a bug.

5.7.2 Median Absolute Error

If the data contains outliers, epsecially if those outliers are very far from the regression line, they can significantly affect the value of MSE, because the square of the error for such outlying examples will be high. In such a situation, it is useful to apply a different metric, the **median absolute error**, MdAE:

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

where $\left\{\frac{f(\mathbf{x}_i)-y_i}{y_i}\right\}_{i=1}^{N}$ denotes the set of absolute error values for all examples from $i = 1$ to N.

5.7.3 Almost Correct Predictions Error Rate

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

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

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

- accuracy,
- cost-sensitive accuracy,
- precision/recall, and
- area under the ROC curve.

To simplify the illustration, I use a binary classification problem. Where necessary, I show how to extend the approach to the multiclass case.

Before we can define model performance metrics, we need to understand the confusion matrix.

5.7.4 Confusion Matrix

The **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 label that the model predicted, and the other axis is the actual label. In a binary classification problem, there are two classes. Let's say, the model predicts two 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 confusion matrix shows that of the 24 examples that actually were spam, the model correctly classified 23 as spam. In this case, we say that we have 23 **true positives** or TP = 23. The model incorrectly classified 1 example as not_spam. In this case, we have 1 **false negative**, or FN = 1. Similarly, of 568 examples that actually were not spam, 556 were correctly classified (556 **true negatives** or TN = 556), and 12 were incorrectly classified (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 decide to add more labeled examples of these species to help the learning algorithm to "see" the difference between them. Alternatively, you might add additional features the learning algorithm can use to build a model that would better distinguish between these species.

Confusion matrix is used to calculate two other performance metrics: **precision** and **recall**.

5.7.5 Precision and Recall

The two most frequently used metrics to assess the model are **precision** and **recall**. Precision is the ratio of correct 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 correct positive predictions to the overall number of positive examples in the dataset:

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

To understand the meaning and importance of precision and recall for the model assessment it is often useful to think about the prediction problem as the problem of research of documents in the database using a query. Precision is the proportion of relevant documents in the list of all returned documents. Recall is the ratio of the relevant documents returned by the search engine to the total number of the relevant documents that could have been returned.

In the case of the spam detection problem, we want to have high precision (we want to avoid making mistakes by detecting that a legitimate message is a spam) and we are ready to tolerate lower recall (we tolerate some spam messages in our inbox).

Almost always, in practice, we have to choose between a high precision or a high recall. It's usually impossible to have both. We can achieve either of the two by various means:

- by assigning a higher weighting to the examples of a specific class (the SVM algorithm accepts weightings of classes as input);
- by tuning hyperparameters to maximize precision or recall on the validation set;
- by varying the decision threshold for algorithms that return probabilities of classes; for instance, if we use logistic regression or decision tree, to increase precision (at the cost of a lower recall), we can decide that the prediction will be positive only if the probability returned by the model is higher than 0.9.

Even if precision and recall are defined for the binary classification case, you can always use them to assess a multiclass classification model. To do that, 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. You would want to avoid situations where the first model has a higher precision while the second model has a higher recall. If it's the case, which one 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 the second metric. For example, you can say that you will accept any model whose recall is above 90% and then you will give preference to the model whose precision is the highest (assuming that its recall is a value above 90%). This technique is also known under the name of **optimizing and satisficing technique** of combining performance metrics.

Alternatively, some practitioners use a combination of precision and recall, such as **F-measure** (also known as **F-score**). The traditional F-measure known as 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 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 higher as precision, and 0.5, which, similarly, weighs recall lower than precision.

As an analyst, you must find a way to combine the two metrics that works best for your problem. Besides F-score, there are other ways to combine multiple metrics allowing to obtain one number:

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

5.7.6 Accuracy

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 equally important. In case of the spam/not spam, this may not be the case. For example, you would tolerate false positives less than false negatives. A false positive in spam detection is the situation in which your friend sends you an email, but the model labels it as spam and doesn't show you. On the other hand, the false negative is less of a problem: if your model doesn't detect a small percentage of spam messages, it's not a big deal.

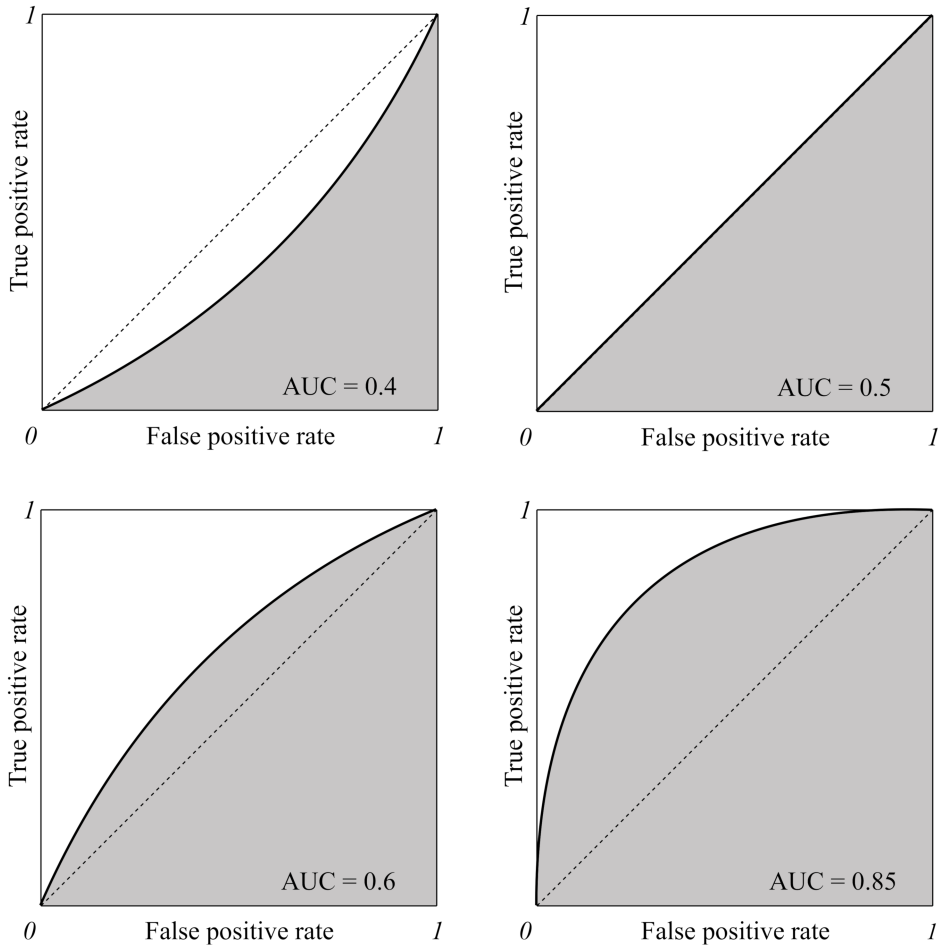


Figure 2: The area under the ROC curve (shown on gray).

5.7.7 Cost-Sensitive Accuracy

For dealing with the situation in which different classes have different importance, a useful metric is **cost-sensitive accuracy**. To compute a cost-sensitive accuracy, you first assign a cost (a positive number) to both types of mistakes: FP and FN. You then compute the counts TP, TN, FP, FN as usual and multiply the counts for FP and FN by the corresponding cost before calculating the accuracy using eq. 2.

5.7.8 Per-Class Accuracy

Accuracy measures the performance of the model for all classes at once. It's good because we preper to have a metric that resumes to one number. At the same time, accuracy is not a good choice of metric when the data is imbalanced. In an **imbalanced dataset**, examples belonging to some class or classes consistute the vast majority, while other classes have very few examples belonging to them. Imbalanced training data can also significantly affect the learning process. We will talk more about dealing with the imbalanced data later in this chapter.

For imbalanced validation or test data, a better accuracy measure is **per-class accuracy**. It works like this. First, we calculate the accuracy of prediction for each class $\{1, \ldots, C\}$ and then we 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 appropriate accuracy measure if, in a multiclass classifcation problem, there are many classes having 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.

5.7.9 Cohen's kappa statistic

Cohen's kappa statistic is a performance metric that apply to both multiclass and imbalanced learning problems. The advantage of this metric over the accuracy is that Cohen's kappa tells you how much better your classificatiom model is performing over the performance of a classifier that simply guesses at random class according to the frequency of each class.

Let's look once again at a confision matrix:

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

Cohen's kappa is defined as:

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

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

The value of p_o is obtained from the confusion matrix as,

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

The value of p_e is also obtained from the confusion matrix 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. There is no universally accepted way to interpret the values Cohen's kappa. It's usually considered that values between 0.61 and 0.80 indicate that the model is good, and values higher than 0.81 indicate that the model is very good.

5.7.10 Area under the ROC Curve (AUC)

The ROC curve (stands for "receiver operating characteristic;" the term comes from radar engineering) is a commonly used method to assess the performance of 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 some confidence 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 confidence score. If this range for a model is $[0, 1]$, then you can discretize it like this: $[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$. Then, you use each discrete value as the prediction threshold and predict the labels of examples in your dataset using the model and this threshold. For example, if you want to compute TPR and FPR for the threshold equal to 0.7, you apply the model to each example, get the score, and, if the score is higher than or equal to 0.7, you predict the positive class; otherwise, you predict the negative class.

Look at the illustration in Figure 2. It's easy to see that if the threshold is 0, all our predictions will be positive, so both TPR and FPR will be 1 (the upper right corner). On the other hand, if the threshold is 1, then no positive prediction will be made, both TPR and FPR will be 0 which corresponds to the lower-left corner.

The higher the **area under the ROC curve** (AUC), the better the classifier. A classifier with an AUC higher than 0.5 is better than a random classifier. If AUC is lower than 0.5, then something is wrong with your model. A perfect classifier would have an AUC of 1. Usually, if your model behaves well, 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 negatives into account) and allow visually and with low effort comparing the performance of different models.

5.7.11 Performance Metrics for Ranking

Precision and recall can be naturally applied to the ranking problem. Indeed, when I explained precition and recall I said that it's convenient to think of it as a measuring the quality of results of search of documents in the database using a query. Precision is the proportion of relevant documents in the list of all returned documents. Recall is the ratio of the relevant documents returned by the search engine to the total number of the relevant documents that could have been returned.

The drawback of precision-recall for measuring the success of ranking models is that they treat all retrieved documents equally; a relevant document in position k counts just as much as a relevant document in position 1. This is not usually how what we want. When a human looks at the results from a search engine, the few top-most results matter much 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.

DCG related measures is based on two assumptions:

1. Highly relevant documents are more useful when appearing earlier in a search engine result list.
2. Highly relevant documents are more useful than marginally relevant documents, which are in turn more useful than non-relevant documents.

To understand discounted cumulative gain, we first 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 has to be numeric, for example 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 if relevant. Notice, that CG_p is independent of the position each document is in the ranked list of results. It only characterizes the documents ranked up to position p as relevant or irrelevant to the query. The value of CG is unaffected by changes in the ordering of search results.

The metric called **discounted cumulative gain** (DCG) is based on a premise that highly relevant documents appearing lower in a search result list should be penalized as the graded relevance value is reduced logarithmically proportional to the position of the result.

For a give search result, DCG accumulated at a particular rank position p is traditionally 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, also commonly used in industry as well as in data science competitios such as Kaggle, places 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:

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

where IDCG is ideal discounted cumulative gain,

$$\mathrm{IDCG_p} \stackrel{\text{def}}{=} \sum_{i=1}^{|REL_p|} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

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 measure of the average performance of a search engine ranking algorithm or model.

Let us 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 the relevance to the query of each document returned by the search engine. The ranker has to assign o each document a score in the range from 0 to 3, where 0 means not relevant, 3 means highly relevant, and 1 and 2 mean "somewhere in between". Let, for the documents ordered by the ranking algorithm as,

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

our ranker provides the following relevance scores:

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

which means that, according to the ranker's perception, 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,

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

You can see that changing the order of any two documents does not affect the value of cumulative gain. Let's now calculate the discounted cumulative gain designed, by applying 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 the 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 due to the fact that 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 need to first 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 the the value of nDCG for a collection of test queries and the corresponding lists of search results, one per query, we average the values of nDCG_p obtained for each individual query. As you can see, 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. It is useful when the number p of relevance scores provided by the rankers for different queries is different.

5.8 Bias-Variance Tradeoff

As you already read above, working on a model is searching for an optimal algorithm and optimal values of that algorithm's hyperparameters. Tweaking the values of hyperparameters actually controls two tradeoffs. We already discussed the first one: the precision-recall tradeoff. The second one, no less important than the first one, is the **bias-variance tradeoff**.

The model is said to have a **low bias** if it predicts well the labels of the training data. If the model makes many mistakes on the training data, we say that the model has a **high bias** or that the model **underfits** the training data. So, underfitting is the inability of the model to predict well the labels of the data it was trained on. There could be several reasons for underfitting, the most important of which are:

- your model is too simple for the data (for example a linear model can often underfit);
- the features you engineered 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 **??** (left). The regression line doesn't repeat the bends of the line to which the data seemingly belongs; that is, 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, if possible;
- reducing regularization.

Overfitting is another problem a model can exhibit. The model that overfits predicts very well the labels of the training data but works poorly on the hold-out data.

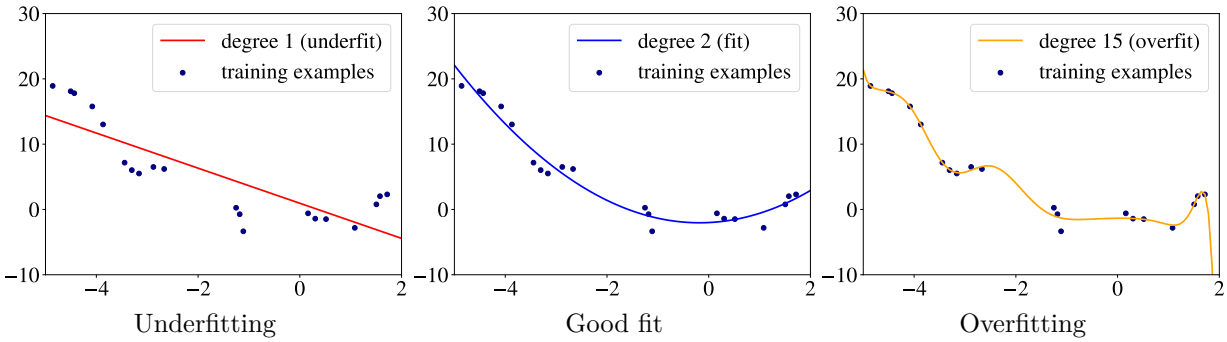


Figure 3: Examples of underfitting (linear model), good fit (quadratic model), and overfitting (polynomial of degree 15).

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

In the literature, you can find another name for the problem of overfitting: the problem of **high variance**. The variance is an error of the model due to its sensitivity to small fluctuations in the training set. It means that if your training data was sampled differently, the learning would result in a significantly different model. This is why the model that overfits performs poorly on the test data: test and training data are sampled from the dataset independently of one another, and, thus, the small fluctuations in the training and test data are likely to be very different.

Several reasons can lead to overfitting, the most important of which are:

- your model is too complex for the data (for example a very tall decision tree or a very deep or wide neural network often overfit);
- you have too many features but few training examples;
- you regularize not enough.

Figure 3 illustrates a one-dimensional dataset for which a regression model underfits, fits well and overfits the data. Several solutions to the problem of overfitting are possible:

- use a simpler model (linear instead of polynomial regression, or SVM with a linear kernel instead of RBF, a neural network with fewer layers/units);
- reduce the dimensionality of examples in the dataset;
- add more training data, if possible;
- regularize the model.

In practice, by trying to reduce variance you increase bias and the other way around. In other words, trying to minimize overfitting leads to underfitting. The inverse is also true: 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. This is why this phenomenon is

called the bias-variance tradeoff.

While many factors determine whether the model will perform 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, that is it will have a low bias. At the same time, a model relying on memorization will not be able to correctly predict labels of the previously unseen data; that is, it will have high variance.

A typical evolution of the average prediction error of a model applied to the training and holdout data, as the model complexity grows, is shown in Figure 4.

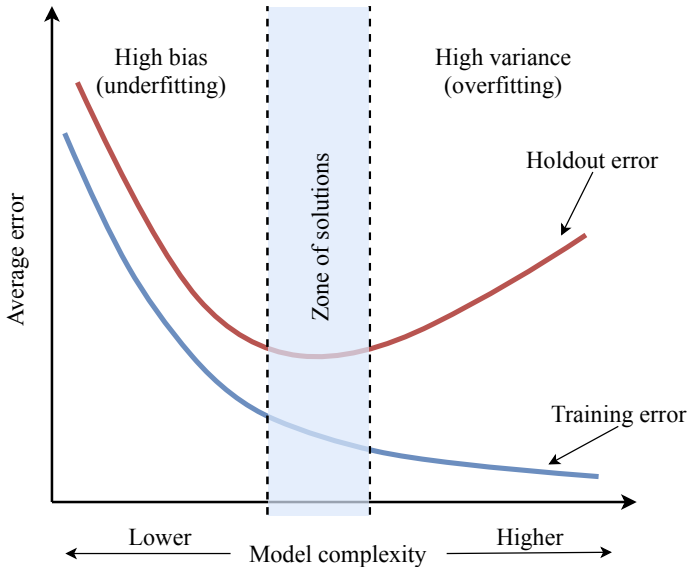


Figure 4: Bias-variance tradeoff.

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 you reach this zone by varying model complexity and other hyperparameters of the learning algorithm, 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 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 following section).

You might wonder how to increase the complexity of the model if you observe high bias in

your model. That depends on the model you work with. If you work with shallow models, like linear regression, then the complexity can be increased 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 reducing bias by combining many high-bias "weak" models; by increasing the number of weak models you can reduce the bias of your ensemble model.

If you work with neural networks, you increase the complexity of the model by increasing its size: the number of units per layer or the number of layers. Training the neural network model longer also often results in lower bias. The advantage of 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 4, you now 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 that encompasses 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.

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 by adding a penalizing term whose value is higher when the model is more complex.

For simplicity, let's illustrate regularization using the example of linear regression. The same principle can be applied to a wide variety of models.

For the sake of illustration and without loss of generality, let's assume that our feature vector $\mathbf{x}$ is a two-dimensional 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. In our illustrative two-dimensional case, the equation of the linear regression line f will have the form $f = w^{(1)}x^{(1)} + w^{(2)}x^{(2)} + b$, where $w^{(1)}$, $w^{(2)}$ and b are the parameters whose values have to be deduced by the learning algorithm from the training data by minimizing the objective. A model is considered less complex, if some of the parameters $w^{(\cdot)}$ have values close or equal to zero.

For clarity, let's stay in the two-dimensional input setting. An L1-regularized objective 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, and 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. In the next section, I will show how to do that.

An L2-regularized objective in the two-dimesional 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**, a model that has most of its parameters (in case of linear models, most of $w^{(\cdot)}$) equal to zero, provided the hyperparameter C is large enough. So, as we already discussed in the previous chapter, L1 implicitly performs **feature selection** by deciding which features are essential for prediction and which are not. That can be useful in case you want to increase model explainability. However, if your only goal is to maximize the performance of the model on the holdout data, then L2 usually gives better results.

L1 and L2 regularization methods can also be combined in what is called **elastic net regularization** with L1 and L2 regularizations being special cases. You can find in the literature the name **ridge regularization** for L2 and **lasso** for L1.

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

Neural networks 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 talk about these techniques later in the section on training neural networks.

5.10 Hyperparameter Tuning

As you remember, step 3 of the shallow model building strategy prescribes to choose a **hyperparameter tuning** strategy. Hyperparameters play an important role in the process

of creation of the model by a machine learning algorithm. Some hyperparameters influence the speed of training, but the most important hyperparameters control the two tradeoffs: bias-variance and precision-recall.

As you probably know, hyperparameters aren't optimized by the learning algorithm itself. The data analyst has to "tune" hyperparameters by experimentally finding the best combination of values, one per hyperparameter. Each machine learning model and the learning algorithm has a unique set of hyperparameters. Furthermore, your entire machine learning pipeline that includes such steps as data pre-processing, feature extraction, model building, and making predictions, can have its own hyperparameters. For example, in data pre-processing the hyperparameters could specify whether to use data-augmentation or not or which technique to use to fill missing values; in feature engineering, a hyperparameter could define which feature selection technique to apply; when making predictions using a model that returns a probability for each class, a hyperparameter could specify the decision threshold for each class.

One typical way to tune hyperparameters, when you have enough data to have a decent validation set (in which each class is represented by at least a couple of dozen examples) and the number of hyperparameters and their range is not too large is to use **grid search**.

5.10.1 Grid Search

Grid search is the most simple hyperparameter tuning technique. I will explain this technique for the problem of tuning two numerical hyperparameters. The technique consists of discretizing each of the two hyperparameters and then testing each pair of discrete hyperparameter values, as shown in Figure 5. Each test consists of applying to the training data the learning algorithm configured using a pair of hyperparameter values, building a model and computing the value of 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 building the final model.

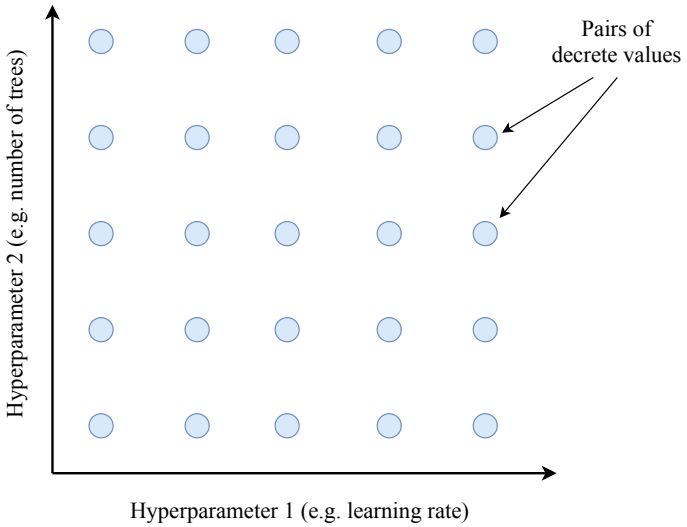


Figure 5: Grid search for two hypeparameters: each blue cercle represents a pair of hyperparameter values.

As you could notice, trying all combinations of hyperparameters, especially if there are more than a couple of them, could be time-consuming, especially for large datasets. There are more efficient techniques, such as random search, coarse-to-fine search, and Bayesian hyperparameter optimization.

5.10.2 Random Search

Random search differs from grid search in that you no longer 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 and set the total number of combinations you want to test as shown in Figure 6.

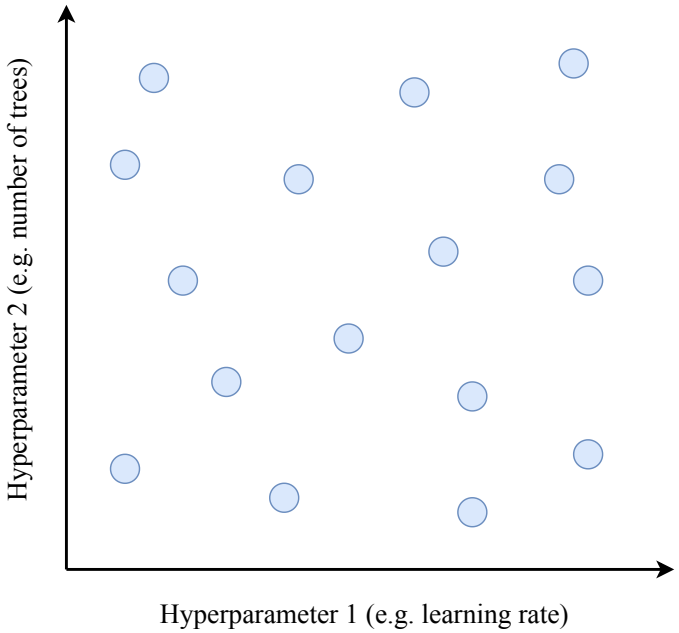


Figure 6: Random search for two hypeparameters and 16 pairs to test.

In practice, analysts most frequently use a combination of grid search and random search called **coarse-to-fine search**.

5.10.3 Coarse-to-Fine Search

The technique of coarse-to-fine search of hyperparameter values consists of first using a coarse random search to find the regions of high potential and then using a fine grid search to find the best values for hyperparameters in the regions of high potential as shown in Figure 7.

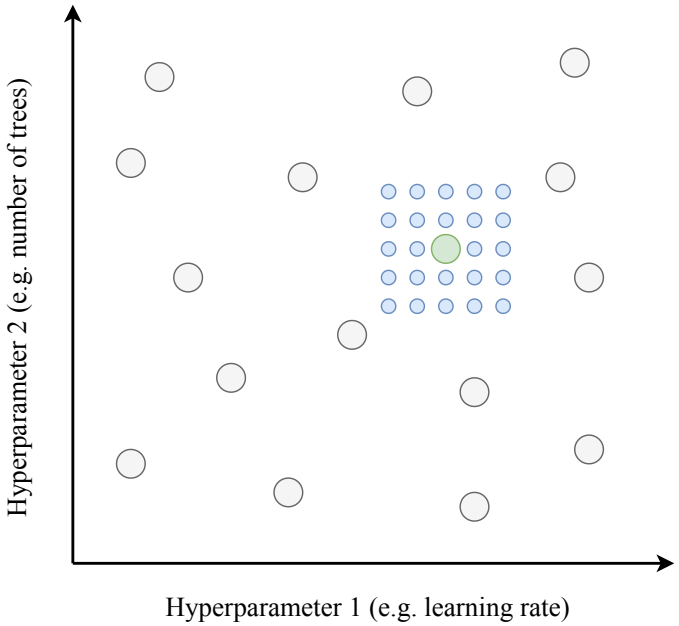


Figure 7: Coarse-to-fine search for two hypeparameters, 16 coarse random search pairs to test and one grid search in the region of the highest value found using the random search.

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

5.10.4 Other Techniques

Bayesian techniques differ from random or grid search in that they use past evaluation results to choose the next values to evaluate. Bayesian hyperparameter optimization techniques can find better hyperparameters in less time because they reason about the best set of hyperparameters to evaluate based on past trials.

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

5.10.5 Cross-Validation

When you don't have a decent validation set to tune your hyperparameters on, the common technique that can help you is called **cross-validation**. When you have few training examples, it could be prohibitive to have both validation and test set. You would prefer to use more data to train the model. In such a case, you only split your data into a training and a test set. Then you 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 you want 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 in practice. With five-fold cross-validation, 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 as follows. 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 as the validation set. You continue building models iteratively like this and compute the value of the metric of interest on each validation set, from F_1 to F_5. Then you average the five values of the metric to get the final value.

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

5.11 Dealing With Distribution Shift

It can sometimes happen that the holdout data, the one that must resemble the data you can observe in production, is not available in sufficiently large quantities. At the same time, you can have access to the labeled data that looks quite like the data from the production environment, but not exactly the same. For example, you might have lots of labeled images from some collection such as Web crawl, but you want to build the classifier of Instagram photos. You might not have enough labeled Instagram photos to use them for training, so you hope to build the model by using the labeled photos from the Web crawl and then use that model to classify the photos from Instagram.

A situation in which the distribution of the training and test data is not the same is called **distribution shift**. Dealing with the distribution shift is currently considered an open research area. Researchers distinguish three types of distribution shift:

- **covariate shift** — shift in the values of features;
- **prior probability shift** — shift in the values of the target;
- **concept shift** — shift in the relationship between the features and the target.

You might know that your data is affected by a distribution shift but you don't usually know what type of shift it is.

If the number of examples in the test set is relatively high compared to the size of the training set, you could randomly pick a certain fraction of test examples and put some of them to the training set and use some test examples for validation. Then you would train the model as usual. However, more often you have a very high number of training examples and relatively few test examples. In the latter case, an effective way to proceed sometimes referred to as **adversarial validation** is as follows.

We assume that the feature vectors in the training and test example contain the same number of features and those features represent the same information. Split your original training set into two subsets: training set 1 and training set 2. Transform the examples in training set 1 in the following way. In each example from training set 1, add the original label as an additional feature to the feature vector. Add the new label "Training" to each example in the modified training set 1.

Transform the examples in the test set in the following way. In each example from the test set, add the original label as an additional feature to the feature vector. Add the new label "Test" to each example in the modified test set. Merge the modified training set 1 and the modified test set to obtain a new training set for a binary classification problem of distinguishing "Training" and "Test" examples. By using that synthetic dataset, build a binary classifier that returns a prediction score.

Now we have a binary classifier that can predict, for a given original example, whether it's a training or a test example. Apply that binary classifier to the examples from training set 2 and find the examples predicted "Test", which the binary model was most certain about. Use those examples as validation data for your original problem. Remove from training set 1 the examples for which the model predicted "Training" with the highest certainty. Use the remaining examples in training set 1 as the training data for your original problem.

You have to find out experimentally what is the ideal way to split the original training set into training set 1 and training set 2. You also have to find out how many examples from training set 1 to use for training and how many of them to use for validation.

5.12 Model Calibration

Sometimes it is important that the classification model returns not just the predicted class, but also the probability that the predicted class is correct. Some models return a score along with the predicted class. That score, however, is not always the probability. We say that the model is well-calibrated when it returns the classification score for input example $\mathbf{x}$ and predicted label $\hat{y}$ that can be used as the probability for $\mathbf{x}$ to belong to class $\hat{y}$. When a well-calibrated model returns the score s (such that $0 \leq s \leq 1$) that the input $\mathbf{x}$ belongs to some class c, we indeed observe that the prediction is true for $s \times 100\%$ examples. For instance, a well-calibrated binary classifier should classify the samples such that among the

examples to which it gave a score value close to 0.8, approximately 80% actually belong to the positive class.

Most machine learning algorithms build models that are not well-calibrated as shown by the **calibration plots** in Figure 8.

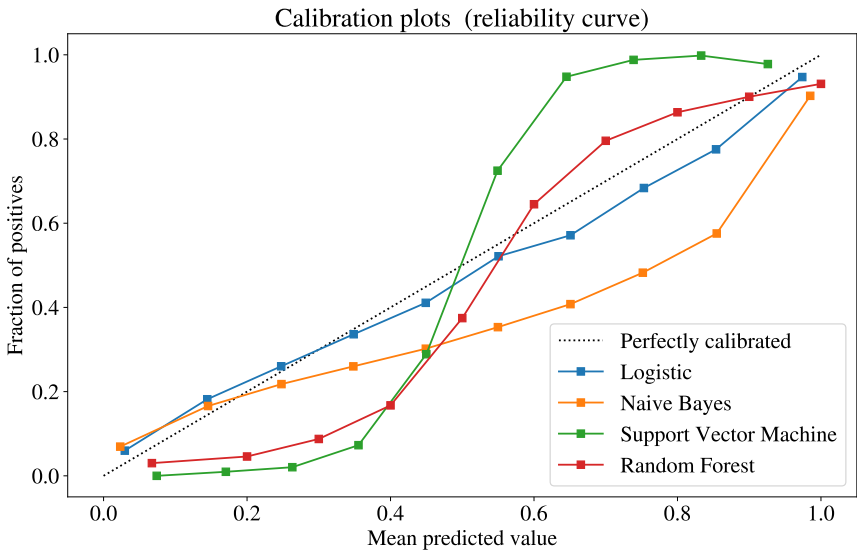


Figure 8: Calibration plots for models built by several machine learning algorithms applied to a random binary dataset.

A calibration plot for a binary model allows seeing how well the model is calibrated. On the X-axis, there are bins that group examples by the predicted score. For example, if we have 10 bins, the left-most bin groups all examples for which the predicted score is in the range $[0, 0.1)$ while the right-most bin groups all examples for which the predicted score is in the range $(0.9, 1.0]$. On the Y-axis there are the fractions of positive examples in each bean. (For multiclass classification, we would have one calibration plot per class in a **one versus rest** way.)

When the model is well-calibrated, the calibration plot oscillates around the diagonal (shown as a dotted line in Figure 8). The closer the calibration plot is to the diagonal, the better the model is calibrated. The calibration plot for logistic regression is close to the diagonal, as the logistic regression model indeed returns the real probabilities of the positive class.

When the model is not well-calibrated, the calibration plot usually has a sigmoid-shape as you can see in Figure 8 for Support Vector Machine and Random Forest.

There are two techniques that are often used to calibrate a binary model: **Platt scaling** and **isotonic regression**. They are based on similar principles. Let us have a model f that we

want to calibrate. First of all, we need a holdout dataset specifically for calibration: to avoid overfitting, we can use for calibration neither training nor validation data. Let this dataset be of size M. Then, we apply the model f to each example $i = 1, \ldots, M$ and obtain, each example i, the prediction f_i. Then we build a new dataset $\mathcal{Z}$, where each example is a tuple (f_i, y_i), where y_i is the true label of example i (labels have the values in the set $\{0, 1\}$).

The only difference between Platt scaling and isotonic regression is that the former builds a logistic regression model by using the dataset $\mathcal{Z}$, while the latter builds the isotonic regression of $\mathcal{Z}$. Once we have the calibration model z, obtained either using Platt scaling or isotonic regression, we can predict the calibrated probability for an input $\mathbf{x}$ as $z(f(\mathbf{x}))$.

Notice that a calibrated model may or may not result in better quality prediction for your problem. That depends on the chosen model performance metric.

According to experiemnts[1]: Platt scaling is most effective when the distortion in the predicted probabilities is sigmoid-shaped. Isotonic regression is a more powerful calibration method that can correct a wider range of distortions. Unfortunately, this extra power comes at a price. Analysis has shown that isotonic regression is more prone to overfitting, and thus performs worse than Platt scaling when data is scarce. Experiments with eight classification problems also suggested that random forests, neural networks, and bagged decision trees are the best learning methods for predicting well-calibrated probabilities prior to calibration, but after calibration, the best methods are boosted trees, random forest, and SVM.

5.13 Deep Model Building Strategy

For deep neural networks, the model building strategy has more moving parts. At the same time, it's also more principled and, as a consequence, amenable to automation.

Model building starts with shortlisting several network architectures (also known as network topologies). If you work with image data and you want to build your model from scratch, then a convolutional neural network (CNN) with at least one convolutional layer, followed by a max-pooling layer, and one fully connected layer may be your default choice of topology.

If you work with text or other sequence data such as time series, you have a choice between a CNN, a gated recurrent neural network (such as LSTM or GRU) or Transformer.

Instead of building your model from scratch, you can also decide to start with a pre-trained model. Companies like Google and Microsoft have trained very deep neural networks with architectures optimized for image or natural language processing tasks.

Among the most used pre-trained models for image processing tasks are **VGG16** and **VGG19** (based on the **VGG** architecture), **InceptionV3** based on the **GoogLeNet** architecture, and **ResNet50** based on the **residual network** architecture.

[1] Alexandru Niculescu-Mizil and Rich Caruana, "Predicting Good Probabilities With Supervised Learning", appearing in Proceedings of the 22nd International Conference on Machine Learning, Bonn, Germany, 2005.

For natural language text processing, such pre-trained models as **BERT** (based on the **Transformer** architecture) and **ELMo** (based on the **bi-directional LSTM** architecture) often improves the quality of the model compared with training a model from scratch.

An advantage of using pre-trained models is that the latter are usually pre-trained on a huge quantity of data available to its creators but unavailable to you. Even if your own dataset is smaller and is not exactly similar to the one that was used to pre-train the model, the parameters learned by the pre-trained models on datasets different to yours can still be useful for your task.

You can use a pre-trained model in two ways:

1) use its learned parameters to initialize your own model, or
2) use the pre-trained model as a feature extractor for your model.

If you use the pre-trained model the former way, it gives you more flexibility but usually, you have to train a very deep neural network. That requires a significant amount of computational resources. If you use it the latter way, you "freeze" the parameters of the pre-trained model and only train the parameters of one or several additional layers that you add on top of the pre-trained model.

5.13.1 Neural Network Building Strategy

Using an existing model to create a new model is called **transfer learning**. We will return to this topic later in this chapter. For the moment, assume that you build from scratch a model based on an architecture of your choice. A common strategy to build a neural network looks as follows:

1. Define a performance metric P.
2. Define the cost function C.
3. Pick a parameter-initialization strategy W.
4. Pick a cost-function optimization algorithm A.
5. Choose a hyperparameter tuning strategy T.
6. Pick a combination H of hyperparameter values using T.
7. Use the training set and build the model M using A parametrized with H to optimize C.
8. If there are still untested hyperparameter values, pick another combination H of hyperparameter values using T and go to step 7.
9. Return the model for which the value of metric P was maximized.

Now let's discuss some of the steps of the above strategy in detail. Step 1 is similar to step 1 of the shallow model building strategy: you must define a metric that would allow comparing the performance of two models on the holdout data and selecting the best of two. An example of a performance metric is **F-score** or **MSE**.

In step 2, you must define what your learning algorithm will optimize in order to build a

model. In shallow learning algorithms, the optimization objective is an integral part of the algorithm (like in linear and logistic regression and support vector machines[2]) or there's no objective and the optimization is implicit (like in decision trees and k-Nearest Neighbors). If your neural network is a regression model, then, in most cases, the cost function is the **mean squared error** (MSE) defined in eq. 1.

For classification, a typical choice of the cost function is **categorical cross-entropy** (for multiclass classification) and **binary cross-entropy** (for binary or multi-label classification).

I already mentioned that when you train a neural network for multiclass classification, you should represent labels using the one-hot encoding. Let's denote as $y_{i,j}$ the value of the one-hot vector representing example i, where i spans from 1 to N, in position j, where j spans from 1 to C and C is the number of classes in our classification problem. The categorical cross-entropy loss for classification of example i is defined as,

$$\text{CCE}_i \stackrel{\text{def}}{=} -\sum_{j=1}^{C} \left[y_{i,j} \times \log_2(\hat{y}_{i,j}) \right],$$

where $\hat{y}$ is the C-dimensional vector of prediction issued by the neural network for the input feature vector $\mathbf{x}_i$. The cost function is typically defined as the sum of losses of individual examples:

$$\text{CCE} \stackrel{\text{def}}{=} \sum_{i=1}^{N} \text{CCE}_i.$$

In binary classification, the output of the neural network for the input feature vector $\mathbf{x}_i$ is a single value $\hat{y}_i$, while the label of the example is a single value y_i, just like in logistic regression. The binary cross-entropy loss for classification of example i is defined as,

$$\text{BCE}_i \stackrel{\text{def}}{=} -y_i \times \log_2(\hat{y}_i) - (1 - y_i) \times \log_2(1 - \hat{y}_i).$$

Similarly, the cost function for classification of the training set is typically defined as the sum of losses of individual examples:

$$\text{BCE} \stackrel{\text{def}}{=} \sum_{i=1}^{N} \text{BCE}_i.$$

Binary cross-entropy is also used in multiclass classification. The labels are now C-dimensional bag-of-words vectors y_i, while the predictions are C-dimensional vectors $\hat{y}_i$, whose values $\hat{y}_{i,j}$

[2]Indeed, there are several formulations of the SVM models (each coming with an associated learning algorithm) and they have different cost functions. In neural networks, you select the architecture of the network, the cost function, and the learning algorithm independently of one another.

in each dimension j range between 0 and 1. The loss for the prediction of one label $\hat{y}_i$ is defined as,

$$\text{BCEM}_i \stackrel{\text{def}}{=} \sum_{j=1}^{C} \left[-y_{i,j} \times \log_2(\hat{y}_{i,j}) - (1 - y_{i,j}) \times \log_2(1 - \hat{y}_{i,j}) \right].$$

The cost function for the classification of the entire training set is typically defined as the sum of losses of individual examples,

$$\text{BCEM} \stackrel{\text{def}}{=} \sum_{i=1}^{N} \text{BCEM}_i.$$

Note that output layers in multiclass and multi-label classification are different. In multiclass classification, one **softmax** unit is used. It generates a C-dimensional vector whose values are bounded by the range $(0, 1)$ and sum to 1. On the other hand, in multi-label classification, the output layer contains C logistic units whose values also lie in the range $(0, 1)$ but the sum of their values lies in the range $(0, C)$.

For completeness, let's also see what the output of the neural network looks like so that you see why the choice of specific loss functions was made.

In regression, the output layer of the neural network contains only one unit. If the output values can range from minus infinity to infinity, then the output unit doesn't contain non-linearity: the output of the neural network can be any number. If the neural network has to predict a positive number, then the ReLU non-linearity is used. Let the output of the output unit before non-linearity be denoted as z. The output after applying the ReLU non-linearity (ReLU for "rectified linear unit") will be $\max\{0, z\}$.

In binary classification, the output layer contains only one logistic unit. Let the output for the input example i of the output unit before non-linearity be denoted as z_i. The output $\hat{y}_i$ after applying the logistic nonlinearity will be as follows,

$$\hat{y}_i \stackrel{\text{def}}{=} \frac{1}{1 + e^{-z_i}},$$

where e is the base of the natural logarithm also known as Euler's number.

The only difference between binary and multi-label classification is that the output layer of the neural network for multi-label classification contains C logistic units, one per label.

In the multiclass classification, the output layer also produces C outputs. However, the output of each unit of the output layer, in this case, is controlled by the softmax function. Let the output, for the input example i, of the output unit j before nonlinearity be $z_{i,j}$. The output $\hat{y}_{i,j}$ after nonlinearity is given by,

$$\hat{y}_{i,j} \stackrel{\text{def}}{=} \frac{e^{z_{i,j}}}{\sum_{k=1}^{C} e^{z_{i,k}}}.$$

In step 3, you must select a parameter-initialization strategy. At the beginning of the training of a neural network, the values of the parameters in all units are unknown, but you have to initialize them with some values. Training algorithms for neural networks, such as gradient descent and its stochastic variants, are iterative in nature and thus require the analyst to specify some initial point from which to begin the iterations. It turns out that the initialization might affect the properties of the model you will obtain as the result of the training.

Typical unit parameter initialization strategies are:

- **ones** — all parameters are initialized to 1;
- **zeros** — all parameters are initialized to 0;
- **random normal** — parameters are initialized to values sampled from the normal distribution, typically with mean of 0 and standard deviation of 0.05;
- **random uniform** — parameters are initialized to values sampled from the uniform distribution with the range $[-0.05, 0.05]$;
- **Xavier normal** — parameters are initialized to values sampled from the truncated normal distribution centered on 0 with standard deviation equal to $\sqrt{2/(\text{in} + \text{out})}$ where in is the number of units in the preceding layer the current unit (the one whose weights you initialize) is connected to and out is the number of units on the subsequent layer the current unit is connected to;
- **Xavier uniform** — parameters are initialized to values sampled from a uniform distribution within $[-\text{limit}, \text{limit}]$ where limit is $\sqrt{6/(\text{in} + \text{out})}$ where in is the number of units in the preceding layer the current unit is connected to and out is the number of units on the subsequent layer the current unit is connected to.

Other initialization strategies exist. If you work with a neural network training module such as TensorFlow, Keras, or PyTorch, they usually provide some parameter initializers and also recommend default choices.

The bias term in all units is usually initialized with a zero.

While we know that the parameter initialization does affect the properties of the model, we cannot say in advance which strategy will provide the best result for your problem. Random and Xavier initializers are the most common though so it's recommended to start your experiments with one of those two strategies.

In step 4, you must select a cost-function optimization algorithm. When the cost function is differentiable (and it's the case for the cost functions we considered above) **gradient descent** and **stochastic gradient descent** are two most frequently used optimization algorithms.

Gradient descent is an iterative optimization algorithm for finding a local minimum of any differentiable function. The difference between a local and a global minimum of a function is

shown in Figure 9.

Functions and optimization

In this block, for the curious reader, I explain the basics of mathematical function and function optimization. You can safely skip it if you only want to know the mechanics of training neural networks.

A function is a relation that associates each element x of a set $\mathcal{X}$, the **domain** of the function, to a single element y of another set $\mathcal{Y}$, the **codomain** of the function. A function usually has a name. If the function is called f, this relation is denoted $y = f(x)$ (read f of x), the element x is the argument or input of the function, and y is the value of the function or the output. The symbol that is used for representing the input is the variable of the function (we often say that f is a function of the variable x).

We say that $f(x)$ has a **local minimum** at $x = c$ if $f(x) \geq f(c)$ for every x in some open interval around $x = c$. An **interval** is a set of real numbers with the property that any number that lies between two numbers in the set is also included in the set. An **open interval** does not include its endpoints and is denoted using parentheses. For example, $(0, 1)$ means "all numbers greater than 0 and less than 1". The minimal value among all the local minima is called the **global minimum**.

A **derivative** f' of a function f is a function or a value that describes how fast f grows (or decreases). If the derivative is a constant value, like 5 or -3, then the function grows (or decreases) constantly at any point x of its domain. If the derivative f' is itself a function, then the function f can grow at a different pace in different regions of its domain. If the derivative f' is positive at some point x, then the function f grows at this point. If the derivative of f is negative at some x, then the function decreases at this point. The derivative of zero at x means that the function's slope at x is horizontal.

The process of finding a derivative is called **differentiation**.

Derivatives for basic functions are known. For example if $f(x) = x^2$, then $f'(x) = 2x$; if $f(x) = 2x$ then $f'(x) = 2$; if $f(x) = 2$ then $f'(x) = 0$ (the derivative of any function $f(x) = c$, where c is a constant value, is zero).

If the function we want to differentiate is not basic, we can find its derivative using the **chain rule**. For instance if $F(x) = f(g(x))$, where f and g are some functions, then $F'(x) = f'(g(x))g'(x)$. For example if $F(x) = (5x + 1)^2$ then $g(x) = 5x + 1$ and $f(g(x)) = (g(x))^2$. By applying the chain rule, we find $F'(x) = 2(5x + 1)g'(x) = 2(5x + 1)5 = 50x + 10$.

Gradient is the generalization of derivative for functions that take several inputs (or one input in the form of a vector or some other complex structure). A gradient of a function is a vector of **partial derivatives**. You can look at finding a partial derivative of a function as the process of finding the derivative by focusing on one of the function's inputs and by considering all other inputs as constant values.

For example, if our function is defined as $f([x^{(1)}, x^{(2)}]) = ax^{(1)} + bx^{(2)} + c$, then the

partial derivative of function f *with respect to* $x^{(1)}$, denoted as $\frac{\partial f}{\partial x^{(1)}}$, is given by,

$$\frac{\partial f}{\partial x^{(1)}} = a + 0 + 0 = a,$$

where a is the derivative of the function $ax^{(1)}$; the two zeroes are respectively derivatives of $bx^{(2)}$ and c, because $x^{(2)}$ is considered constant when we calculate the derivative with respect to $x^{(1)}$, and the derivative of any constant is zero.

Similarly, the partial derivative of function f with respect to $x^{(2)}$, $\frac{\partial f}{\partial x^{(2)}}$, is given by,

$$\frac{\partial f}{\partial x^{(2)}} = 0 + b + 0 = b.$$

The gradient of function f, denoted as ∇f is given by the vector $[\frac{\partial f}{\partial x^{(1)}}, \frac{\partial f}{\partial x^{(2)}}]$.

The chain rule works with partial derivatives too.

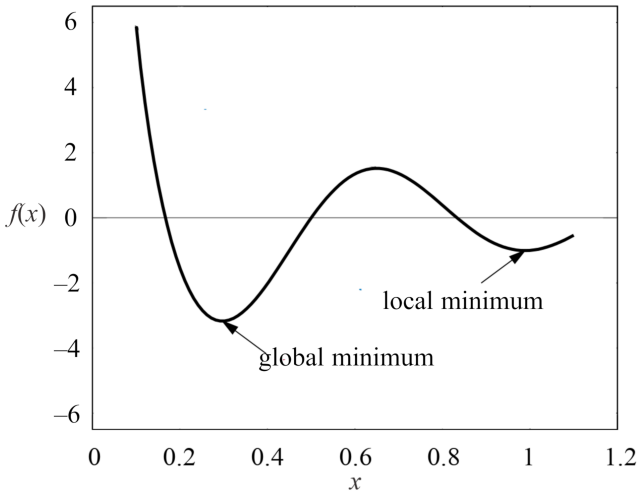


Figure 9: A local and a global minima of a function.

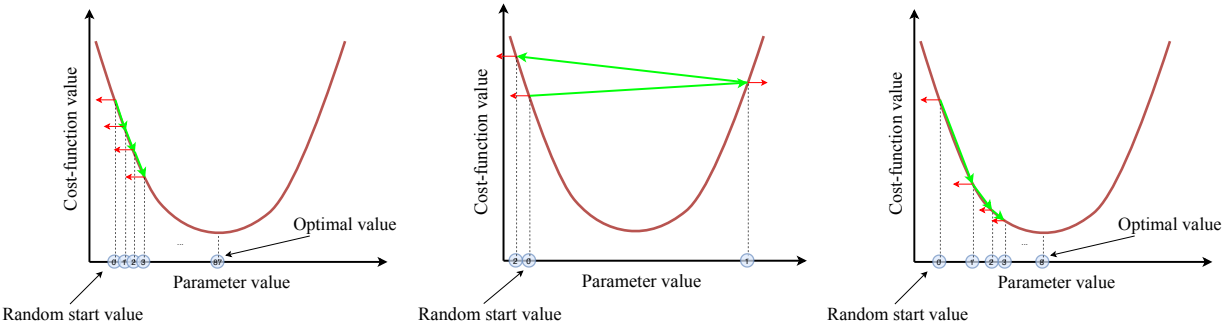
To find a local minimum of a function using gradient descent, one starts at some random point in the domain of the function and takes steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point.

Gradient descent in machine learning proceeds in **epochs**. An epoch consists of using the training set entirely to update each parameter. In the beginning, the first epoch, we initialize the parameters of our neural network using one of the parameter-initialization strategies discussed above. The **backpropagation** algorithm computes the partial derivatives of each

parameter using the chain rule for derivatives of complex functions[3]. At each epoch, gradient descent updates all parameters one by one using partial derivatives. The **learning rate** α controls the significance of an update. The process continues until **convergence**, the state when the values of parameters don't change much after each epoch; then the algorithm stops.

Gradient descent is sensitive to the choice of the learning rate α; picking the right value of the learning rate for your problem is not easy. If you select a value that is too high, you might not reach convergence at all. On the other hand, too small values of α can slow down the learning to the point of no observable progress. In Figure 10, you can see an illustration of gradient descent for one parameter of a neural network and three values of the learning rate. The values of the parameter at each iteration is shown as a blue circle. The number inside the circle indicates the epoch. The red arrows indicate the direction of the gradient along the horizontal axis — the direction away from the minimum. The green arrows show the change in the value of the cost function after each epoch.

Therefore, at each epoch, gradient descent moves the parameter value towards the minimum. If the learning rate is too small, the movement towards the minimum will be very slow (Figure 10a). If the learning rate is too large, the value of the parameter will oscillate away from the minimum (Figure 10b).



(a) too small learning rate (a) too large learning rate (b) good learning rate

Figure 10: The influence of the learning rate on the convergence: (a) too small, the convergence will be slow; (b) too large, no convergence; (c) the right value of learning rate.

Gradient descent is also slow for large datasets because it uses the entire dataset to compute the gradient of each parameter at each epoch. Fortunately, several significant improvements to this algorithm have been proposed.

Minibatch stochastic gradient descent (minibatch SGD) is a variant of the gradient descent algorithm that speeds up the computation by approximating the gradient using smaller batches (subsets) of the training data. The size of the minibatch is a hyperparameter

[3]The explanation of backpropagation is beyond the scope of this book. You should only know that every modern software library for training neural networks contains an implementation of this algorithm.

and you can tune it; a power of two, between 32 and a few hundred, are recommended values: 32, 64, 128, 256, and so on. However, the problem of choosing a value for α is still present in the "vanilla" minibatch SGD. Though, that hyperparameter can, of course, also be tuned.

Even if you choose an appropriate value of the learning rate, the learning can still stagnate at later epochs because, instead of reaching a local minimum, the gradient descent keeps oscillating around it due to too large updates. There are many learning rate decay schedules that allow updating the size of the learning rate as the learning progresses by reducing it depending on the epoch count. The benefits of using a learning decay schedule include faster gradient descent convergence (faster learning) and higher model accuracy. We will talk about learning rate decay schedules later in this chapter.

5.13.2 Regularization

In neural networks, besides L1 and L2 regularization, you can use neural network-specific regularizers: **dropout**, **early stopping**, and **batch-normalization**. The latter is technically not a regularization technique, but it often has a regularization effect on the model.

The concept of dropout is very simple. Each time you run a training example through the network, you temporarily exclude at random some units from the computation. The higher the percentage of units excluded the higher the regularization effect. Neural network libraries allow you to add a dropout layer between two successive layers, or you can specify the dropout parameter for the layer. The dropout parameter varies in the range $[0, 1]$ and it has to be found experimentally.

Early stopping is the way to train a neural network by saving the preliminary model after every epoch and assessing the performance of the preliminary model on the validation set. During gradient descent, as the number of epochs increases, the cost decreases. The decreased cost means that the model fits the training data well. However, after some epoch e, the model can start overfitting: the cost keeps decreasing, but the performance of the model on the validation data deteriorates. If you keep, in a file, the version of the model after each epoch, you can stop the training once you start observing a decreased performance on the validation set. Alternatively, you can keep running the training process for a fixed number of epochs and then, in the end, you pick the best model. Models saved after each epoch are called **checkpoints**. Some machine learning practitioners rely on this technique very often; others try to properly regularize the model to avoid such undesirable behavior.

Batch normalization (which rather has to be called batch standardization) is a technique that consists of **standardizing** the outputs of each layer before the units of the subsequent layer receive them as input. In practice, batch normalization results in faster and more stable training, as well as some regularization effect. So, it's always a good idea to try to use batch normalization. In neural network libraries, you can often insert a batch normalization layer between two layers.

Another regularization technique that can be applied not just to neural networks, but to

virtually any learning algorithm, is **data augmentation**. This technique is often used to regularize models that work with images. In practice, applying data augmentation often results in increased performance of the model.

5.13.3 Learning Rate Decay Schedules

As I already mentioned, learning rate decay is useful for convergence of gradient descent: it consists of gradually reducing the value of the learning rate α as the epochs progress. As a consequence, the updates of the parameters become finer. There are several techniques, known as schedules, to control α.

Time-based learning rate decay schedules alter the learning rate depending on the learning rate of the previous epoch. The mathematical formula for the learning rate update according to a popular time-based learning rate decay schedule is:

$$\alpha_n \leftarrow \frac{\alpha_{n-1}}{1 + d \times n},$$

where α_n is the new value of the learning rate, α_{n-1} is the value of the learning at epoch $n - 1$, and d is the **decay rate**, a hyperparameter. For example, if the initial value of the learning rate $\alpha_0 = 0.3$, then the values of the learning rate at the first five epochs are shown below:

learning rate	epoch
0.15	1
0.10	2
0.08	3
0.06	4
0.05	5

Step-based learning rate decay schedules change the learning rate according to some pre-defined steps. The mathematical formula for the learning rate update according to a popular step-based learning rate decay schedule is:

$$\alpha_n \leftarrow \alpha_0 d^{\text{floor}\left(\frac{1+n}{r}\right)},$$

where α_n is the learning rate at epoch n, α_0 is the initial value of the learning rate, d is the decay rate that reflects how much the learning rate should change at each drop step (0.5 corresponds to halving) and r is the so-called drop rate defining the length of drop steps (10 corresponds to a drop every 10 epochs). The floor operator here returns 0 if the value of its argument is smaller than 1.

Exponential learning rate decay schedules are similar to step-based but instead of drop steps, a decreasing exponential function is used. The mathematical formula for the learning rate update according to a popular exponential learning rate decay schedule is:

$$\alpha_n \leftarrow \alpha_0 e^{-d \times n}$$

where d is the decay rate.

There are several popular upgrades to minibatch SGD, such as Momentum, RMSProp, and Adam. These algorithms update the learning rate automatically based on the performance of the learning process so that you don't have to worry about choosing the initial value of the learning rate, the decay schedule and rate and the values of other related hyperparameters. These algorithms have demonstrated good performance in practice and practitioners often use them instead of trying to tune the learning rate by themselves.

Momentum is a method that helps accelerate SGD by orienting the gradient descent in the relevant direction and reducing oscillations. Instead of using only the gradient of the current epoch to guide the search, momentum accumulates the gradient of the past steps to determine the direction to go. Momentum takes away the need to adjust the learning rate; it does that automatically.

More recent advancements in neural network cost function optimization algorithms include **RMSProp** and **Adam**, the latter being the most recent and versatile. It's recommended to start building the model with Adam and then, if the performance of the model doesn't reach the acceptable level despite all efforts, to try a different cost function optimization algorithm.

Step 5 of the deep model building strategy is similar to that in the shallow model building strategy — choose a hyperparameter tuning strategy T. At step 6, you pick a combination of hyperparameter values using T. Typical parameters include the size of the minibatch, the value of the learning rate (if you use the vanilla minibatch SGD) or one of the algorithms that update the learning rate automatically, such as Adam. The analyst also decides on the initial number of layers and units per layer in the neural network. There's no way to know how many layers and units per layer to start with, so start with something reasonable, that will allow you to build your first model fast enough. For example, two hidden layers and 128 units per layer might be a good starting point.

Step 7 reads as "Use the training set and build the model M by applying the optimization algorithm A parametrized with the values H of hyperparameters to optimize the cost function C" and this is where the difference with shallow learning becomes substantial.

When you work with a shallow model, you can only tweak some hyperparameters to improve the model. You don't have much control over the model architecture and complexity. With neural networks, you have all the control and building a model is actually a process and not a single action. To build a deep model, as I already mentioned above, you start with a reasonably sized model and then you follow the flowchart shown in Figure 11.

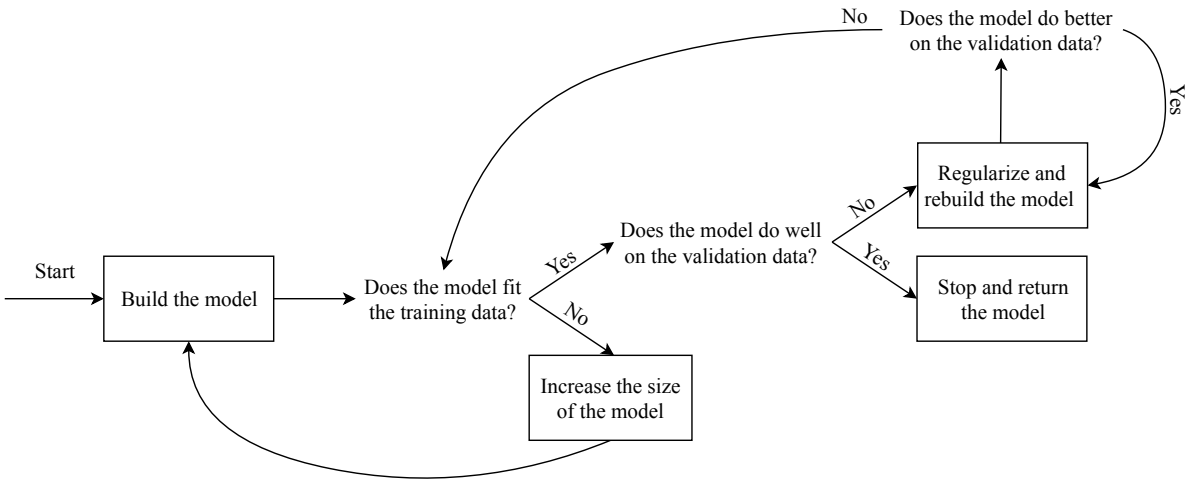


Figure 11: The neural network model building flowchart.

As you can see in Figure 11, you start with some model and then you increase its size until it fits the training data well. Once you reach that condition, you test the model on the validation data. If it performs well on the validation data according to the performance metric, then you stop and return the model. Otherwise, you regularize and rebuild the model. As we have seen, regularization in neural networks usually achieved in several ways. The most effective is **dropout** when you randomly remove some connections from the network by making it simpler and "dumber". A simpler model would work better on the validation data, and this is your goal. Another way to regularize is to add **L1** or **L2** regularization.

If after several loops of regularization and model rebuild you don't see any improvement in the model performance on the validation data, you check if it still fits the training data. If it doesn't, then you increase the size of the model (by increasing the size of individual layers or by adding another layer) and continue doing so until the model fits the training data once again. Then you test it again on the validation data and the process continues until the increase of the size of the model doesn't result, in the end, in increased performance on the validation data, no matter how hard you regularize and how wide or deep your neural network becomes. Then you stop.

If you are not satisfied with the performance of your best model on the validation data, you can, on step 8, pick a different combination of hyperparameters and build a different model. You will continue to test different values of hyperparameters until there are no more values to test. Then you keep the best model among those you trained in the process.

If the performance of the best model is still not satisfactory for you, you can try a different network architecture, add more labeled data, or try transfer learning.

5.13.4 Hyperparameter Tuning

The properties of a trained neural network depend a lot on the choice of the values of hyperparameters. But before you choose specific values of hyperparameters, build a model and validate its properties on the validation data, you must decide which hyperparameters are important enough for you to spend the time on trying their values.

Obviously, if you had infinite time and computing resources, you would tune all hyperparameters. However, in practice, you have finite time and often relatively modest resources. Which hyperparameters to tune?

While there is no definitive answer to that question, there are several observations that might help you in choosing the hyperparameters to tune when you work on a specific model:

- your model is more sensitive to some hyperparameters than to others;
- the choice is often between using the default value of a hyperparameter or changing it.

The libraries for training neural networks often come with default values for most hyperparameters: the version of the stochastic gradient descent (often Adam), the parameter initialization strategy (often random normal or random uniform), minibatch size (often 32) and so on. Those choices of default values were made based on observations from practical experience. Open-source libraries and modules are often the fruit of the collaboration of many scientists and engineers; these talented and experienced people established "good" defaults for many hyperparameters based on experience gained when working with various datasets and practical problems.

If you decide to tune a hyperparameter as opposed to using the default value, it makes the most sense to tune the hyperparameters to which the model is the most sensitive. The table below[4] shows several hyperparameters and approximate sensitivity of a model to them:

Hyperparameter	**Sensitivity**
Learning rate	High
Learning rate schedule	High
Loss function	High
Units per layer	High
Parameter initialization strategy	Medium
Number of layers	Medium
Layer properties	Medium
Degree of regularization	Medium
Choice of optimizer	Low
Optimizer properties	Low
Size of minibatch	Low
Choice of non-linearity	Low

[4]Taken from the talk "Troubleshooting Deep Neural Networks" by Josh Tobin et al., January 2019.

5.13.5 Handling Multiple Inputs

In practice, machine learning engineers often work with multimodal data. For example, the input could be an image and a text and the binary output could indicate whether the text describes the given image.

It's hard to adapt **shallow learning** algorithms to work with multimodal data. For example, you can try to vectorize each input (by applying the corresponding feature engineering method) and then concatenate two feature vectors together to form one wider feature vector. For example, if your image has features $[i^{(1)}, i^{(2)}, i^{(3)}]$ and your text has features $[t^{(1)}, t^{(2)}, t^{(3)}, t^{(4)}]$ your concatenated feature vector will be $[i^{(1)}, i^{(2)}, i^{(3)}, t^{(1)}, t^{(2)}, t^{(3)}, t^{(4)}]$.

With neural networks, you have substantially more flexibility. You can build two subnetworks, one for each type of input. For example, a CNN subnetwork would read the image while an RNN subnetwork would read the text. Both subnetworks have as their last layer an embedding: CNN has an embedding of the image, while RNN has an embedding of the text. You can now concatenate two embeddings and then add a classification layer, such as softmax or sigmoid, on top of the concatenated embeddings. Neural network libraries provide simple-to-use tools that allow concatenating or averaging of layers from several subnetworks.

5.13.6 Handling Multiple Outputs

In some problems, you would like to predict multiple outputs for one input. Some problems with multiple outputs can be effectively converted into a multi-label classification problem. Especially those that have labels of the same nature (like tags) or fake labels can be created as a full enumeration of combinations of original labels.

However, in some cases the outputs are multimodal, and their combinations cannot be effectively enumerated. Consider the following example: you want to build a model that detects an object on an image and returns its coordinates. In addition, the model has to return a tag describing the object, such as "person," "cat," or "hamster." Your training example will be a feature vector that represents an image. The label will be represented as a vector of coordinates of the object and another vector with a one-hot encoded tag.

To handle a situation like that, you can create one subnetwork that would work as an encoder. It will read the input image using, for example, one or several convolution layers. The encoder's last layer would be the embedding of the image. Then you add two other subnetworks on top of the embedding layer: one that takes the embedding vector as input and predicts the coordinates of an object. This first subnetwork can have a ReLU as the last layer, which is a good choice for predicting positive real numbers, such as coordinates; this subnetwork could use the mean squared error cost C_1. The second subnetwork will take the same embedding vector as input and predict the probabilities for each tag. This second subnetwork can have a softmax as the last layer, which is appropriate for the probabilistic output, and use the averaged negative log-likelihood cost C_2 (also called **cross-entropy** cost).

Obviously, you are interested in both accurately predicted coordinates and the tags. However, it is impossible to optimize the two cost functions at the same time. By trying to optimize one, you risk hurting the second one and the other way around. What you can do is add another hyperparameter γ in the range $(0, 1)$ and define the combined cost function as $\gamma C_1 + (1-\gamma)C_2$. Then you tune the value for γ on the validation data just like any other hyperparameter.

5.13.7 Transfer Learning

As I already mentioned, **transfer learning** consists of using a pre-trained model to build a new model. Pre-trained models are usually created using a huge quantity of data available to its creators, usually large corporations, but not necessarily available to you. The parameters learned by the pre-trained models can be useful for your task.

A pre-trained model can be used in two ways:

1) its learned parameters can be used to initialize your own model, or
2) it can be used as a feature extractor for your model.

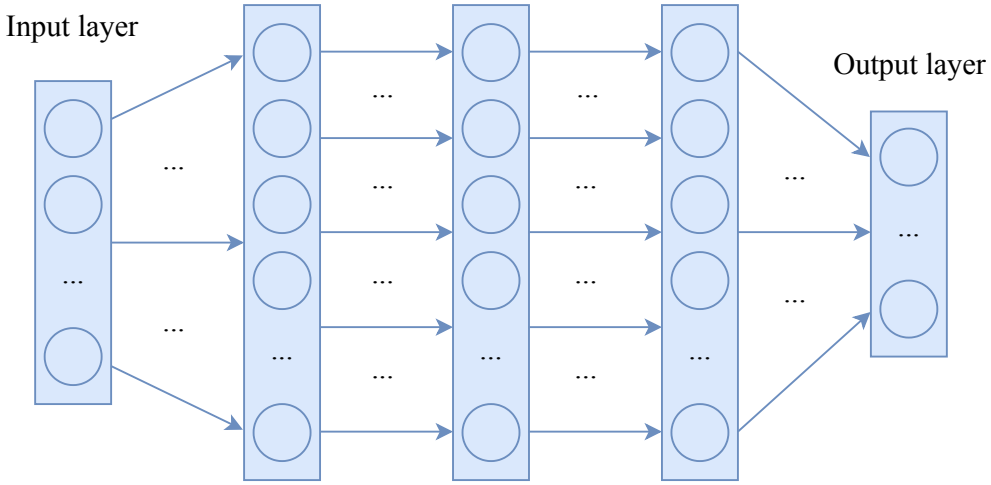
As we already discussed, the choice of parameter initialization strategy affects the properties of the learned model. Pre-trained models available on the Internet, or those built by you, usually perform well for solving the respective problem they were built for. If your current problem is similar to the one that was solved by a pre-trained model, there are high chances that the parameters optimal for your problem will not be too different from the parameters of the pre-trained model, especially in the earliest (closest to the input) neural network layers. If you initialize your model with the values of the parameters of pre-trained models, the learning might go faster because gradient descent will search for the optimal parameter values for your problem in a smaller area of potentially good values. If the pre-trained model was built using a training set much bigger than yours, searching in a smaller area of potentially good values might also lead to a better generalization. Indeed, if some behavior of the model you want to build is not reflected by your training examples, this behavior could still be "inherited" by your model from the pre-trained model.

If you use the pre-trained model the former way, it gives you more flexibility: the gradient descent will modify the parameters in all layers and potentially reach a better performance for your problem. The downside of doing that is that you will often end up training a very deep neural network. Some pre-trained models contain hundreds of layers and millions of units. Training a large network like that can be challenging: it will definitely require a significant amount of computational resources; in addition, in very deep neural networks the problem if the vanishing gradient is more severe than in a network with a couple of hidden layers.

If you have a limited amount of computational resources, you might prefer using some layers of the pre-trained model as feature extractors for your model. In practice, it means that you keep only several earlier layers of the pre-trained model (the layers closest to the input layer, including the latter) and you keep their parameters "frozen", that is unchanged and unchangeable. Then you add new layers on top of the frozen layers, including the output

layer appropriate for your task; only the parameters of the new layers will then be updated by gradient descent during training.

An illustration of the process is shown in Figure 12. The blue neural network is a pre-trained model. Some of the blue layers are reused in the new model; the green layers are added by the analyst and tailored for the problem in hand. As discussed earlier, the analyst may decide to freeze the parameters of the blue part of the new network and only train the parameters of the green part. Alternatively, several right-most blue layers could be set trainable. How many layers of the pre-trained model to use to build the new model, as well as how many layers to freeze, is up to the analyst: it's part of the decisions about the architecture that would work best for your problem.



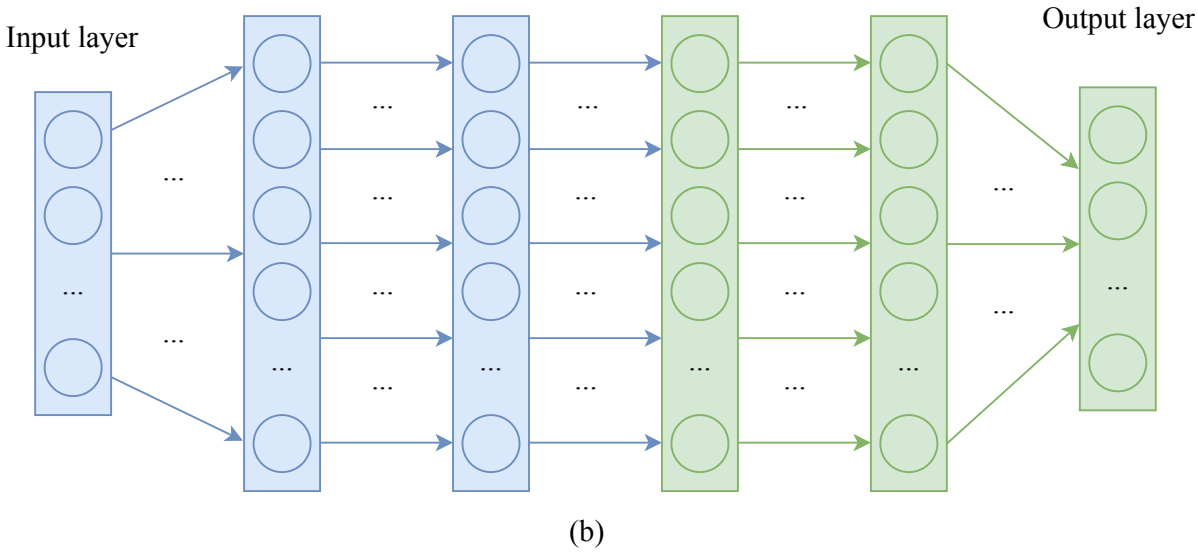


Figure 12: An illustration of transfer learning: (a) the pre-trained model and (b) your model where you used the left part of the pre-trained model and added new layers including a different output layer tailored for your problem.

5.14 Handling Imbalanced Datasets

Often in practice, examples of some class will be underrepresented in your training data. 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. If you use SVM with a soft margin, you can define the loss for misclassified examples. Because noise is always present in the training data, there are high chances that many examples of genuine transactions would end up on the wrong side of the decision boundary by contributing to the cost.

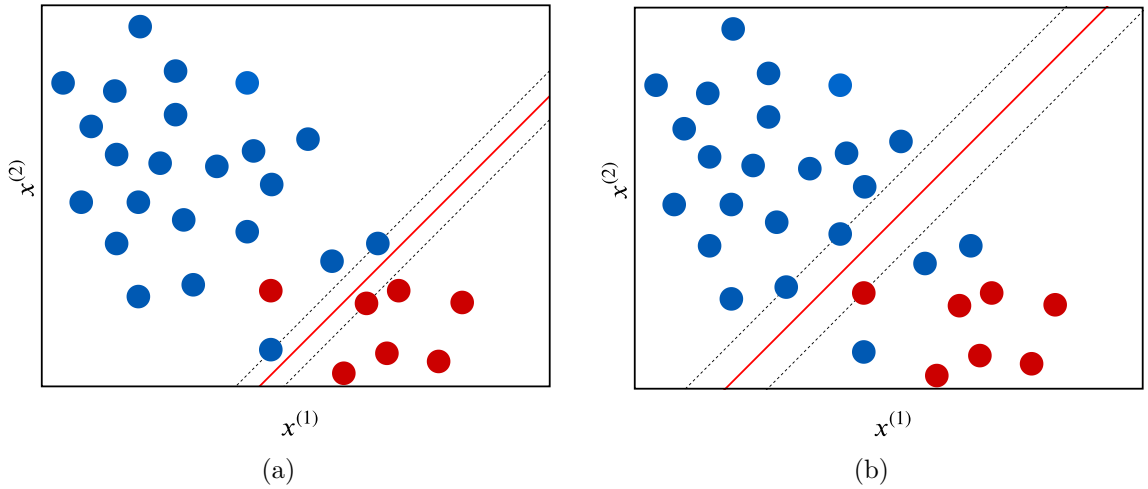


Figure 13: An illustration of an imbalanced problem. (a) Both classes have the same weight; (b) examples of the minority class have a higher weight.

The SVM algorithm tries to move the hyperplane to avoid misclassified examples as much as possible. The "fraudulent" examples, which are in the minority, risk being misclassified in order to classify more numerous examples of the majority class correctly. This situation is illustrated in Figure 13a. This problem is observed for most learning algorithms applied to **imbalanced datasets**.

If you set the loss of misclassification of examples of the minority class higher, then the model will try harder to avoid misclassifying those examples, but this will incur the loss of misclassification of some examples of the majority class, as illustrated in Figure 13b.

Some SVM implementations allow providing weights for every class. The learning algorithm takes this information into account when looking for the best hyperplane.

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.

An opposite approach, **undersampling**, is to randomly remove from the training set some examples of the majority class.

You might also try to create synthetic examples by randomly sampling feature values of several examples of the minority class and combining them to obtain a new example of that class. There are two popular algorithms that oversample the minority class by creating synthetic examples: the *synthetic minority oversampling technique* (**SMOTE**) and the *adaptive synthetic sampling method* (**ADASYN**).

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 as $\mathcal{S}_k$) 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 λ is an arbitrary number in the range $[0, 1]$. (See an illustration for $\lambda = 0.5$ in Figure 14.)

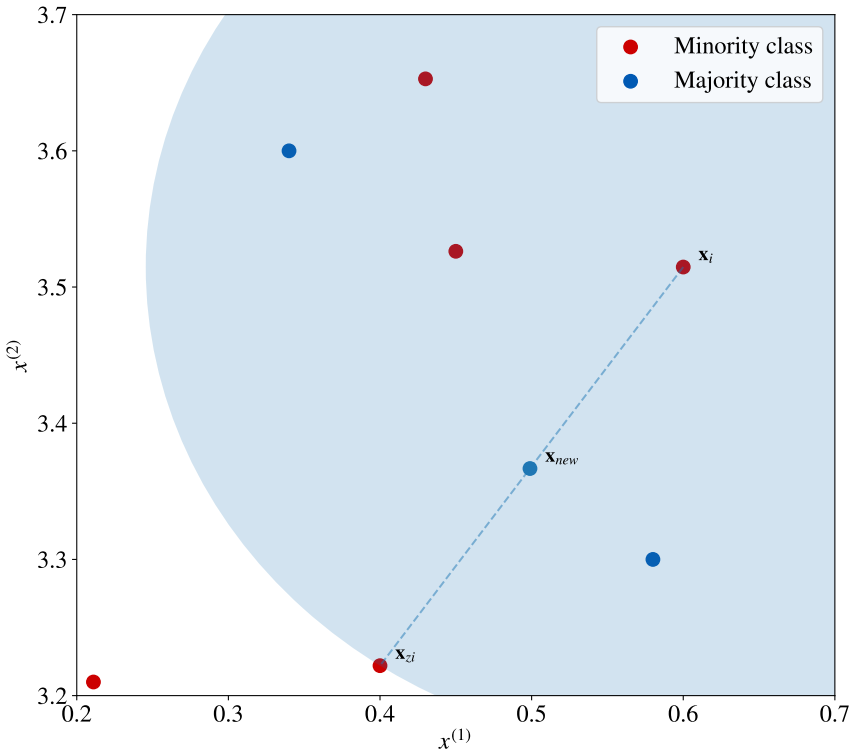


Figure 14: An illustration of a synthetic example generation for SMOTE and ADASYN. (Built using a script adapted from Guillaume Lemaitre.)

Both SMOTE and ADASYN randomly pick among 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.

Some algorithms are less sensitive to the problem of an imbalanced dataset. Decision trees, as well as random forest and gradient boosting, often perform well on imbalanced datasets.

5.15 Model Building as an Iterative Process

Model building is usually an iterative process where an analyst builds a model, observes its behavior and makes adjustments based on observations.

5.15.1 Reasons for Poor Model Behavior

If your model does poorly on the training data (the model underfits the training data), the common reasons for such a phenomenon usually are:

- the model architecture or the learning algorithm you have chosen is not expressive enough (try more advanced learning algorithm, an ensemble method or a deeper neural network);
- you regularize too much (reduce regularization);
- you have chosen suboptimal values for hyperparameters (tune hyperparameters);
- the features you engineered don't have enough predictive power (add more informative features);
- you don't have enough data to generalize (try to get more data, use **data augmentation** or **transfer learning**).

If your model does well on the training data but poorly on the holdout data (the model overfits the training data), the most common reasons for that are:

- you don't have enough data to generalize (add more data or use data augmentation);
- your model is underregularized (add regularization or, for neural networks, both regularization and batch normalization);
- your training data distribution is different from the holdout data distribution (reduce the **distribution shift**);
- you have chosen suboptimal values for hyperparameters (tune hyperparameters);
- your features don't have enough predictive power (add features with high predictive power).

5.15.2 Iterative Model Refinement

If you have access to new labeled data (for example, you can label examples yourself or easily request the help of a labeler) then, once you have identified a model that works reasonably well according to the metric, you can refine the model using a simple iterative process:

- build the model using the best values of hyperparameters identified so far,
- test the model using the metric by applying the model to a small subset of the validation set (100-300 examples),
- find the most frequent error patterns on that small validation set; remove those examples from the validation set as your model will now overfit to them;
- generate new features or add more data to fix the error patterns observed on the small subset of validation examples;
- repeat until no frequent error patterns are observed (all errors look random and dissimilar).

5.15.3 Error Analysis

Whether you are satisfied by the performance of your model on the holdout data or dissatisfied, you can always improve its quality by analyzing the individual errors. I have shown above that the best way to proceed is to work iteratively, by considering 100—300 examples at a time. This process is called **error analysis**.

By considering a small number of examples at a time, you can iterate fast (by rebuilding the model after each iteration) but still consider enough examples to spot obvious patterns.

However, how do you decide whether an error pattern you spotted is worth spending time to fix it? A frequent way to do that is based on the error pattern frequencies. Let's see how it works.

Let your model have an accuracy of 80% which corresponds to the error rate of 20%. This means that if you fix all error patterns, you can hope to improve the performance of your model by at most 20 percentage points. Now, let your small batch of examples you base your error analysis on is of size 300 and your model made $0.2 \times 300 = 60$ errors.

Observe the errors one by one and try to get an idea of what kind of particularities in the input example could lead to a misclassification of each of the 60 examples. To be even more concrete, let our classification problem be to detect pedestrians on the images of the street. Let among those 60 images your model didn't detect a pedestrian but should have on 50 images. Let you see two patterns when that error happens: 1) the image is blurry in 40 examples and the picture is taken during the nighttime in 5 examples. Now, should you spend time to address both problems? If you address the blurry image problem (for example, by adding more labeled blurry images to your training data) you can hope to decrease your error by $(40/60) \times 20 = 13$ percentage point. So, in the best case, after you solve the blurry

image misclassification problem, you can hope that your error becomes $20 - 13 = 7$ percent, a significant decrease from the initial 20% error.

On the other hand, if you decide to solve the nighttime image problem, you can hope to decrease your error by $5/60 \times 20 = 1.7$ percentage point. So, that in the best-case scenario, your model will make $20 - 1.7 = 18.3$ percent errors, which might be significant for some problem, or insignificant for others. You, as the data analyst, should decide whether addressing a specific error pattern is worth the effort.

To fix an error pattern, an analyst can use one or a combination of techniques, such as:

- preprocessing the input (e.g. background removal for images, spelling correction for texts);
- data augmentation (e.g., blurring and darkening of images);
- adding more training examples;
- engineering new features that would allow the learning algorithm to distinguish between "hard" cases.

5.15.4 Error Analysis in Complex Systems

Let's say you work on a complex document classification system that consists of three chained models as shown in Figure 15.

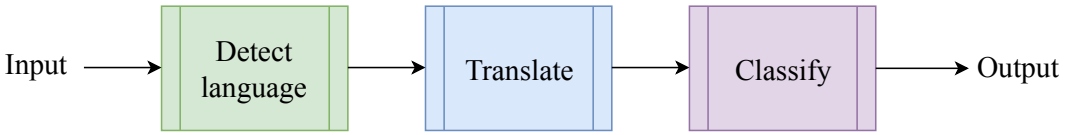


Figure 15: A complex document classification system.

Let the document classification accuracy of the entire system be 80%. It could be high accuracy or low accuracy, depending on the business requirement. If the classification is binary, the accuracy of 80% doesn't seem high. On the other hand, if the classification model (the rightmost block in Figure 15) supports thousands of classes, then the accuracy of 80% doesn't seem low. However, for some business cases, the user of the system might expect human-like or even superhuman performance.

Imagine that you are in a position where the business expects a higher than 80% performance from the document classification system you have built. To get the most out of your additional effort, you have to decide which part of the system needs improvement in the first place.

When the decision about something is made on several chained levels, like in the problem shown in Figure 15, and if those decisions are independent of one another, the accuracy multiplies. If, for example, the accuracy of the language predictor was 95%, the accuracy of

the machine translation model[5] was 90%, and the accuracy of the classifier was 85% then, in the case of independence of the three models, the overall accuracy of the entire three-stage system would be $0.95 \times 0.90 \times 0.85 = 0.73$ or 73 percent. At first glance, it seems obvious that the most gain in the accuracy of the entire system would come from maximizing the accuracy of the third model — the classifier. However, in practice, some errors made by a given model might not significantly affect the overall performance of the system. For example, if the language predictor often confuses Spanish and Portuguese, the machine translation model could still be capable of generating the translation good enough for the third-stage classification model.

Additionally, while you worked on the model of the classifier at the third-stage, you might have come to the conclusion that you reached the maximum of the performance of this model, so it doesn't make sense to try to improve it. Now, which one of the previous two models, that is the language detector and the machine translator, you have to improve to increase the quality of the entire three-stage system?

One way to get the idea of the upper bound on the possible improvement of the entire system thanks to one improved model is to perform the **error analysis by parts**. The latter consists of replacing the predictions of that one model with the perfect labels (such as human-provided labels) and see how the performance of the entire system improves. For example, instead of using the machine translation system at stage two in Figure 15, you can use the language prediction provided by the first model and then ask a professional human translator to translate the text from the predicted language (if the prediction of the language was correct) or keep the original text (if the prediction of the language was wrong).

Let's say you asked a human for a hundred translations. Now you can measure how the fact of having perfect translations affects the overall system performance. Let the accuracy of classification become 74%. So, the potential gain in overall system performance from reaching the human-level translation performance is only one percentage point. Reaching the human-level performance of a machine translation model can turn out to be a daunting task not worth the effort, especially when what we can achieve in the end is one percentage point gain for the entire system. So, you would prefer spending more time on building a better language predictor if the potential gain in overall system performance from increasing the language prediction quality is higher.

5.15.5 Fixing Wrong Labels

When the dataset is labeled by humans, it's often the case that the labels assigned to your training examples are wrong. This can be a reason for the poor performance of the model on both training and holdout data. Indeed, if similar examples have conflicting labels — some correct and some incorrect — the learning algorithm can learn to predict the wrong label.

[5]Measuring the error of the machine translation system in practice is tricky as the translation is rarely entirely accurate or inaccurate.

One simple way to identify the examples that have wrong labels is to apply the model you have built to the training data it was built from and analyze the examples for which the model made the wrong prediction (according to the current labeling). If the model performs reasonably well according to a metric but makes errors on certain training examples, it's often the case that those training examples have wrong labels. Similarly, you can correct labels on the holdout data.

If the fact of having wrong labels in the training data is a serious matter for your task, you can avoid wrong labeling by asking several individuals to provide labels for the same training example, and only accept an example for training if all individuals assigned the same label to that example. In less demanding situations, you can accept the label is the majority of individuals assigned that label to the training example.

5.16 Stacking Models

Ensemble learning is building an ensemble model, which is a combination of several base models, each individually performing worse than the ensemble model.

There are ensemble learning algorithms, such as random forest and gradient boosting, that build an ensemble of several hundred to thousands of **weak models** and obtain a **strong model** that has a significantly better performance than the performance of each weak model. We will not talk about these algorithms here, as I assume that the reader knows the basics of popular machine learning algorithms[6].

The reason that combining multiple models can bring better performance is that when several uncorrelated models agree they are more likely to agree on the correct outcome. The keyword here is "uncorrelated." Ideally, base models should be obtained using different features or using algorithms or models of a different nature — for example, SVM and random forest. Combining different versions of the decision tree learning algorithm, or several SVMs with different hyperparameters, may not result in a significant performance boost.

Here, I will specifically talk about stacking models, which is an ensemble learning method that consists of building a strong model that takes outputs of other strong models as inputs. The base strong models used for stacking are usually weakly correlated models and the goal of the ensemble learning is to learn to combine the strengths of each base model and achieve even better performance than the performance of each individual base model.

More generally, there are three typical ways to combine weakly correlated models: 1) averaging, 2) majority vote and 3) stacking.

Averaging works for regression as well as those classification models that return classification scores. It consists of applying all your base models to the input $\mathbf{x}$ and then averaging the predictions. To see if the averaged model works better than each individual algorithm, you can test it on the validation set using a metric of your choice.

[6] You can read about ensemble learning algorithms in my The Hundred-Page Machine Learning Book.

Majority vote works for classification models. It consists of applying all your base models to the input $\mathbf{x}$ and then returning the majority class among all predictions. In the case of a tie, you can either randomly pick one of the classes, or, return an error message (if the fact of misclassifying would incur a significant loss for the business).

Stacking, being the most effective of the three ensembling methods, consists of building a meta-model that takes the output of base models as input. Let's say you want to combine classifiers f_1, f_2, and f_3, all predicting the same set of classes. To create a training example $(\hat{\mathbf{x}}_i, \hat{y}_i)$ for the stacked model from the original training example $(\mathbf{x}_i, y_i)$, set $\hat{\mathbf{x}}_i = [f_1(\mathbf{x}), f_2(\mathbf{x}), f_3(\mathbf{x})]$ and $\hat{y}_i = y_i$. This is illustrated in Figure 16.

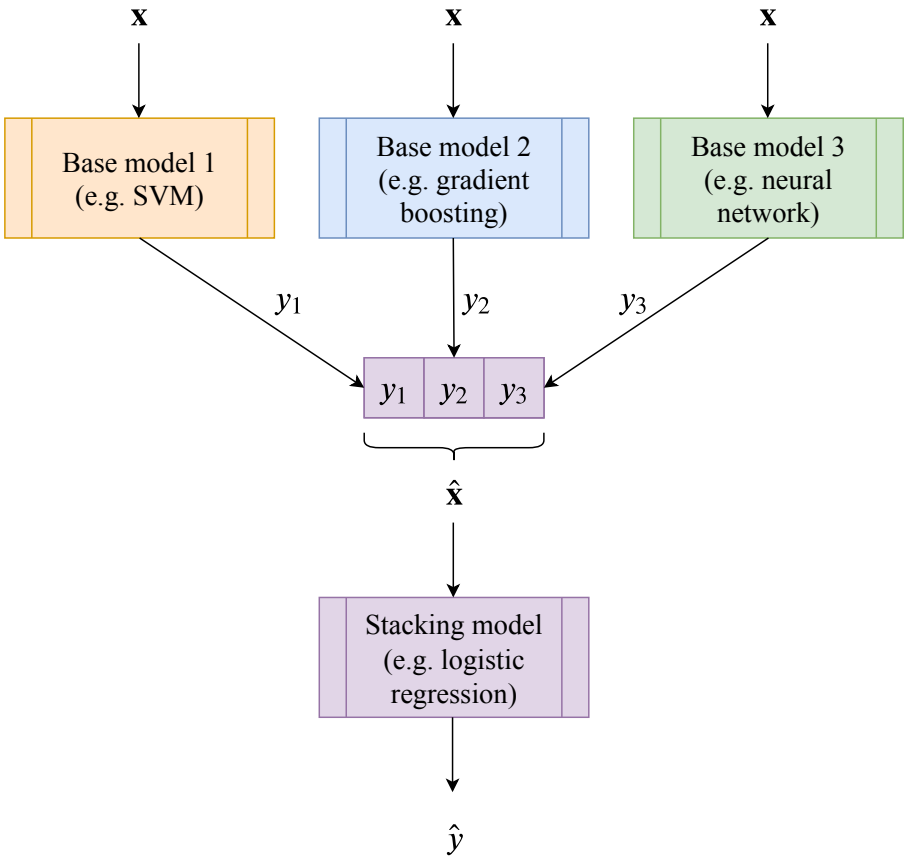


Figure 16: A stacking of three weakly correlated strong models.

If some of your base models return not just a class but also a score for each class, you can use the values of the score as additional input features for the stacked model.

To train the stacked model, it is recommended to use examples from the training set and tune the hyperparameters of the stacked model using cross-validation.

Obviously, you have to make sure that your stacked model performs better on the validation set than each of the base models you stacked.

Be careful to avoid **data leakage** when training the stacking model. To create the training dataset for the stacked model, follow a process similar to cross-validation. First, split all training data into ten or more blocks (the more blocks the better, but the process of building the model will also be slower). Temporarily exclude one block from the training data and train the base models on the remaining blocks. Then apply the base models to the examples in the excluded block, obtain the predictions and build the training examples for the stacked model by using the predictions from the base models. Repeat the same process for each of the remaining blocks and you will end up with the training set for the stacking model. That new training set will be of the same size as the size of the original training set.

5.17 Model Building Best Practices

5.17.1 Deliver a Good Model

What is a good model? A good model has two properties:

- it has the desired quality according to the performance metric;
- it is safe to serve in the production environment.

For a model to be safe to serve means to satisfy the following requirements:

- not crash or cause errors in the serving system when being loaded, or when sent bad or unexpected inputs;
- not use too many resources (such as CPU, GPU or RAM).

We will consider the above two requirements and discuss how to satisfy them in one of the later chapters.

5.17.2 Trust Popular Open Source Implementations

If you don't use an exotic or very recent programming language, it is considered unreasonable, in most practical situations, to implement machine learning algorithms from scratch. Modern open-source libraries and modules for machine learning for the popular modern programming languages and platforms, such as Python, Java, and .NET have permissive licenses and contain efficient industry-standard implementations of popular machine learning algorithms. Additionally, open-source libraries and modules exist specifically for training neural networks.

You would prefer to program your own implementation of a machine learning algorithm only if the model is intended to be executed in a very resource-constrained environment or you need to run your model with the speed no existing library can provide.

5.17.3 Optimize the Performance Metric

While reducing the error on the training data is usually what a learning algorithm tries to achieve, the error on the test data is what you, as a data analyst, want to minimize. Moreover, in many practical situations, you want to optimize a business-problem-specific model **performance metric** computed on the test data.

The model that minimizes the error is not always the same model that optimizes the performance metric, so do strive for error minimization only during the process of iterative model building. Once you reached the minimum error on the validation data or came very close to it, search for the values of hyperparameters that optimize the performance metric, even if, as a consequence, the error increases.

5.17.4 Analyze the Least Certain Predictions

If your model returns a prediction score along with the predicted class, then the least certain predictions according to the score are the ones you should pay the closest attention to. For example, if you have a binary model and you interpret all predictions with the score above 0.7 as positive, then examine the predictions with the scores from 0.65 to 0.75. These are examples for which the model is uncertain and, therefore, either the training set contains contradictive labeling for similar feature vectors (which you would prefer to identify and fix) or the training set doesn't contain enough labeled examples similar to these.

5.17.5 Upgrade from Scratch

It is often a case that the model, once deployed to production, must be periodically updated with new data (for example, to adapt to a particular user's needs). If it's the case for your model, the data used to build the model has to be collected automatically using scripts (as we discussed in Chapter 3 when we talked about **reproducibility**).

Every time the data is updated, the hyperparameters of the model have to be tuned from scratch because the change in the data can lead to suboptimal performance of the model trained with the old values of hyperparameters.

Avoid the practice of iteratively upgrading the existing model by adding only new data and running additional training iterations, even if the model (such as a neural network) allows to be iteratively upgraded. Notice that upgrading the model is not the same as **transfer learning**. Analysts use transfer learning either because the data used to build the pre-trained model or the adequate computing resources are not available. You would without a doubt obtain a better model if you had access to more data, more computational resources, and decided to build the model from scratch.

5.17.6 Write Efficient Code, Compile and Parallelize

By writing fast and efficient code, you can speed up the training by an order of magnitude as compared to an inefficient quick-and-dirty script you implemented during experimentation just "to make it work". Modern datasets are large, so you might end up waiting for hours, or even days for data preprocessing and then the same or even longer time for training.

Always write the code with efficiency in mind, even if it seems to be a function, a method, or a script that you will not run frequently. It often happens in practice that some code that was supposed to run once, ends up being called in a loop millions of times.

Avoid loops in your code as much as possible. For example, if you need to compute a dot product of two vectors, or multiply a matrix by a vector, use fast implementations of dot product or matrix multiplication methods implemented in scientific libraries and modules such as numpy and scipy libraries in Python. Those libraries and modules were created by talented and skilled software engineers and scientists in low-level programming languages such as C and use hardware acceleration, so they work blazingly fast.

Where possible, compile the code before executing it. Such libraries as PyPy and Numba for Python or pqR for R would compile the code into the OS native binary code which can significantly increase the speed of data processing and model training.

Another important aspect is parallelization. If you work with modern machine learning libraries and modules you can find implementations of learning algorithms that exploit multicore CPUs; some libraries allow using GPUs to speed up the building of many machine learning models, not only neural networks. Some algorithms, such as SVM, cannot be effectively parallelized. In such cases, you can still exploit a multicore CPU by running multiple experiments (one experiment for each combination of hyperparameter values) in parallel. Furthermore, each fold of cross-validation can be computed in parallel with other folds.

Where possible, use a solid-state drive (SSD) to store the data, use distributed computing (some learning algorithms are implemented for running in a distributed environment such as Spark). Try to put all the needed data into the RAM of your laptop or server. It's not uncommon for machine learning engineers to work on a server with 512 gigabytes or even several terabytes of RAM.

By reducing to the minimum the time needed to build a model by a machine, you, as an analyst, can spend more time tweaking your model and trying various hypotheses about data pre-processing, feature engineering, neural network architectures, and others. Human touch and intuition is where the biggest benefit for the machine learning project lies. The more you, as a human, can work instead of waiting the higher the chances that the machine learning project will be a success.

5.18 Data Leakage in Model Building

While the **data leakage** problems that can arise at this stage don't have such a significant impact on the model as the leakage happening at the previous stages of the machine learning project life cycle, they are still worth noting and avoiding. We have already seen one example of leakage that can happen when you build a stacked ensemble model.

Another kind of leakage can happen if you use the same cross-validation split to select the best machine learning algorithm. It might happen that your best choice of algorithm will be the result of that specific learning algorithm working better on that specific cross-validation split. To avoid that kind of leakage, you might prefer to generate m different splits, then run m cross-validations for each algorithm, and then average the performance of each learning algorithm on m splits. As you can imagine, it increases a lot the amount of computation, so do that only if you have a small dataset or a lot of computational resources.

5.19 Contributors

I'm grateful to the following people for their valuable contributions to the quality of this chapter: *vacant*.