Python Machine Learning By Example, 3rd

Build intelligent systems using Python,
TensorFlow 2, PyTorch, and scikit-learn

Stephen CUI[1]

23 May, 2023

[1]cuixuanstephen@gmail.com

Chapter 1

Getting Started with Machine Learning and Python

1.1 Digging into the core of machine learning

1.1.1 Overfitting, underfitting, and the bias-variance trade-off

Overfitting

Overfitting means a model fits the existing observations too well but fails to predict future new observations.

The phenomenon of memorization can cause overfitting. This can occur when we're over extracting too much information from the training sets and making our model just work well with them, which is called low bias in machine learning. In case you need a quick recap of bias, here it is: bias is the difference between the average prediction and the true value. It is computed as follows:

$$Bias[\hat{y}] = E[\hat{y} - y]$$

At the same time, however, overfitting won't help us to generalize to new data and derive true patterns from it. The model, as a result, will perform poorly on datasets that weren't seen before. We call this situation high variance in machine learning. Again, a quick recap of variance: variance measures the spread of the prediction, which is the variability of the prediction. It can be calculated as follows:

$$Variance = E[\hat{y}^2] - E[\hat{y}]^2$$

Overfitting occurs when we try to describe the learning rules based on too many parameters relative to the small number of observations, instead of the underlying relationship. Overfitting also takes place when we make the model excessively complex so that it fits every training sample, such as memorizing the answers for all questions, as mentioned previously.

Table 1.1: Setup for 5-fold cross-validation

Round	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
1	Testing	Training	Training	Training	Training
2	Training	Testing	Training	Training	Training
3	Training	Training	Testing	Training	Training
4	Training	Training	Training	Testing	Training
5	Training	Training	Training	Training	Testing

Underfitting

The opposite scenario is **underfitting**. When a model is underfit, it doesn't perform well on the training sets and won't do so on the testing sets, which means it fails to capture the underlying trend of the data. We call any of these situations a high bias in machine learning; although its variance is low as the performance in training and test sets is pretty consistent, in a bad way.

The bias-variance trade-off

Obviously, we want to avoid both overfitting and underfitting. Recall that bias is the error stemming from incorrect assumptions in the learning algorithm; high bias results in underfitting. Variance measures how sensitive the model prediction is to variations in the datasets.

1.1.2 Avoiding overfitting with cross-validation

> When the training size is very large, it's often sufficient to split it into training, validation, and testing (three subsets) and conduct a performance check on the latter two. Cross-validation is less preferable in this case since it's computationally costly to train a model for each single round. But if you can afford it, there's no reason not to use cross-validation. When the size isn't so large, cross-validation is definitely a good choice.

There are mainly two cross-validation schemes in use: exhaustive and non-exhaustive. In the exhaustive scheme, we leave out a fixed number of observations in each round as testing (or validation) samples and use the remaining observations as training samples.

A non-exhaustive scheme, on the other hand, as the name implies, doesn't try out all possible partitions. The most widely used type of this scheme is k-fold cross-validation. We first randomly split the original data into k equal-sized folds. In each trial, one of these folds becomes the testing set, and the rest of the data becomes the training set.

K-fold cross-validation often has a lower variance compared to LOOCV, since we're using a chunk of samples instead of a single one for validation.

We can also randomly split the data into training and testing sets numerous times. This is formally called the holdout method. The problem with this algorithm is that

some samples may never end up in the testing set, while some may be selected multiple times in the testing set.

1.1.3 Avoiding overfitting with regularization

Another way of preventing overfitting is regularization. Recall that the unnecessary complexity of the model is a source of overfitting. Regularization adds extra parameters to the error function we're trying to minimize, in order to penalize complex models.

Besides penalizing complexity, we can also stop a training procedure early as a form of regularization. If we limit the time a model spends learning or we set some internal stopping criteria, it's more likely to produce a simpler model. The model complexity will be controlled in this way and, hence, overfitting becomes less probable. This approach is called early stopping in machine learning.

Last but not least, it's worth noting that regularization should be kept at a moderate level or, to be more precise, fine-tuned to an optimal level. Too small a regularization doesn't make any impact; too large a regularization will result in underfitting, as it moves the model away from the ground truth.

1.1.4 Avoiding overfitting with feature selection and dimensionality reduction

The number of features corresponds to the dimensionality of the data. Our machine learning approach depends on the number of dimensions versus the number of examples.

Not all of the features are useful and they may only add randomness to our results. It's therefore often important to do good feature selection. **Feature selection** is the process of picking a subset of significant features for use in better model construction. In practice, not every feature in a dataset carries information useful for discriminating samples; some features are either redundant or irrelevant, and hence can be discarded with little loss.

Another common approach of reducing dimensionality is to transform high-dimensional data into lower-dimensional space. This is known as **dimensionality reduction** or **feature projection**.

1.2 Data preprocessing and feature engineering

One of the methodologies popular in the data mining community is called the **Cross-Industry Standard Process for Data Mining (CRISP-DM)**

CRISP-DM consists of the following phases, which aren't mutually exclusive and can occur in parallel:

- **Business understanding**: This phase is often taken care of by specialized domain experts. Usually, we have a businessperson formulate a business problem, such as selling more units of a certain product.

- **Data understanding**: This is also a phase that may require input from domain experts; however, often a technical specialist needs to get involved more than in the business understanding phase. The domain expert may be proficient with spreadsheet programs but have trouble with complicated data. In this machine learning book, it's usually termed the exploration phase.
- **Data preparation**: This is also a phase where a domain expert with only Microsoft Excel knowledge may not be able to help you. This is the phase where we create our training and test datasets. In this book, it's usually termed the preprocessing phase.
- **Modeling**: This is the phase most people associate with machine learning. In this phase, we formulate a model and fit our data.
- **Evaluation**: In this phase, we evaluate how well the model fits the data to check whether we were able to solve our business problem.
- **Deployment**: This phase usually involves setting up the system in a production environment (it's considered good practice to have a separate production system). Typically, this is done by a specialized team.

Dealing with missing values

The simplest answer is to just ignore them. The second solution is to substitute missing values with a fixed value—this is called **imputing**. We can impute the arithmetic mean, median, or mode of the valid values of a certain feature. Ideally, we will have some prior knowledge of a variable that is somewhat reliable. For instance, we may know the seasonal averages of temperature for a certain location and be able to impute guesses for missing temperature values given a date.

Power transforms

A very common transformation for values that vary by orders of magnitude is to take the logarithm. Another useful power transform is the Box-Cox transformation, named after its creators, two statisticians called George Box and Sir David Roxbee Cox. The Box-Cox transformation attempts to find the best power needed to transform the original data into data that's closer to the normal distribution. In case you are interested, the transform is defined as follows:

$$y_i^{(\lambda)} = \begin{cases} \frac{y_i^{\lambda} - 1}{\lambda}, \lambda \neq 0 \\ \ln(y_i), \lambda = 0 \end{cases} \tag{1.1}$$

1.2.1 Combining models

Voting and averaging

It just means the final output will be the majority or average of prediction output values from multiple models. Nonetheless, combining the results of models that are highly

correlated to each other doesn't guarantee a spectacular improvement. It is better to somehow diversify the models by using different features or different algorithms. If you find two models are strongly correlated, you may, for example, decide to remove one of them from the ensemble and increase proportionally the weight of the other model.

Bagging(Bootstrap aggregating)

Bootstrapping is a statistical procedure that creates multiple datasets from the existing one by sampling data **with replacement**.

Boosting

It makes sense to take into account the strength of each individual learner using weights. This general idea is called boosting. In boosting, all models are trained in sequence, instead of in parallel as in bagging. Each model is trained on the same dataset, but each data sample is under a different weight factoring in the previous model's success. The weights are reassigned after a model is trained, which will be used for the next training round. In general, weights for mispredicted samples are increased to stress their prediction difficulty.

Stacking

Stacking takes the output values of machine learning models and then uses them as input values for another algorithm. You can, of course, feed the output of the higher-level algorithm to another predictor. It's possible to use any arbitrary topology but, for practical reasons, you should try a simple setup first as also dictated by Occam's razor.

Chapter 2

Building a Movie Recommendation Engine with Naïve Bayes

2.1 Getting started with classification

2.1.1 Binary classification

2.1.2 Multiclass classification

2.1.3 Multi-label classification

A typical approach to solving an n-label classification problem is to transform it into a set of n binary classification problems, where each binary classification problem is handled by an individual binary classifier.

2.2 Evaluating classification performance

Sometimes, a model has a higher average f1 score than another model, but a significantly low f1 score for a particular class; sometimes, two models have the same average f1 scores, but one has a higher f1 score for one class and a lower score for another class. In situations such as these, how can we judge which model works better? The **area under the curve (AUC)** of the **receiver operating characteristic (ROC)** is a consolidated measurement frequently used in binary classification.

2.3 Tuning models with cross-validation

> In k-fold cross-validation, k is usually set at 3, 5, or 10. If the training size is small, a large k (5 or 10) is recommended to ensure sufficient training samples in each fold. If the training size is large, a small value (such as 3 or 4) works fine since a higher k will lead to an even higher computational cost of training on a large dataset.

We will use the split() method from the StratifiedKFold class of scikit-learn to divide the data into chunks with preserved class distribution.

Chapter 3

Recognizing Faces with Support Vector Machine

3.1 Finding the separating boundary with SVM

We will continue with another great classifier, SVM, which is effective in cases with high-dimensional spaces or where the number of dimensions is greater than the number of samples.

A **hyperplane** is a plane of $n-1$ dimensions that separates the n-dimensional feature space of the observations into two spaces.

The optimal hyperplane is picked so that the distance from its nearest points in each space to itself is maximized. And these nearest points are the so-called **support vectors**.

3.2 Scenario 2 – determining the optimal hyperplane

The nearest point(s) on the positive side can constitute a hyperplane parallel to the decision hyperplane, which we call a positive hyperplane; on the other hand, the nearest point(s) on the negative side can constitute the negative hyperplane. The perpendicular distance between the positive and negative hyperplanes is called the margin, the value of which equates to the sum of the two aforementioned distances. A decision hyperplane is deemed optimal if the margin is maximized.

3.3 Scenario 4 – dealing with more than two classes

SVM and many other classifiers can be applied to cases with more than two classes. There are two typical approaches we can take, one-vs-rest (also called one-vs-all) and one-vs-one.

In the one-vs-rest setting, for a K-class problem, we construct K different binary SVM classifiers. For the k^{th} classifier, it treats the kth class as the positive case and

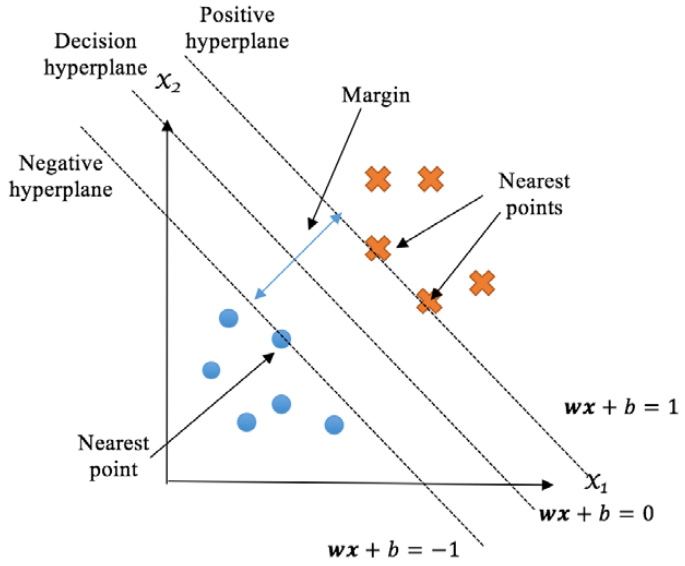

Figure 3.1: An example of an optimal hyperplane and distance margins

the remaining $K-1$ classes as the negative case as a whole; the hyperplane denoted as (w_k, b_k) is trained to separate these two cases. To predict the class of a new sample, x', it compares the resulting predictions $w_k x' + b_k$ from K individual classifiers from 1 to k. As we discussed in the previous section, the larger value of $w_k x' + b_k$ means higher confidence that x' belongs to the positive case. Therefore, it assigns x' to the class i where $w_i x' + b_i$ has **the largest value among all prediction results**:

$$y' = argmax_i(w_k x' + b_k)$$

In the one-vs-one strategy, we conduct a pairwise comparison by building a set of SVM classifiers that can distinguish data points from each pair of classes. This will result in $K(K-1)/2$ different classifiers.

For a classifier associated with classes i and j, the hyperplane denoted as (w_{ij}, b_{ij}) is trained only on the basis of observations from i (can be viewed as a positive case) and j (can be viewed as a negative case); it then assigns the class, either i or j, to a new sample, x', based on the sign of $w_{ij}x' + b_{ij}$. Finally, the class with the highest number of assignments is considered the predicting result of x'. **The winner is the class that gets the most votes**.

Although one-vs-one requires more classifiers, $K(K-1)/2$, than one-vs-rest (K), each pairwise classifier only needs to learn on a small subset of data, as opposed to the entire set in the one-vs-rest setting. As a result, training an SVM model in the one-vs-one setting is generally more memory efficient and less computationally expensive, and hence is preferable for practical use[1].

[1]Chih-Wei Hsu and Chih-Jen Lin's A comparison of methods for multiclass support vector machines

3.4 Scenario 5 – solving linearly non-separable problems with kernels

The most popular kernel function is probably the **radial basis function (RBF)** kernel (also called the Gaussian kernel), which is defined as follows:

$$K(x^{(i)},x^{(j)}) = \exp(-\frac{||x^{(i)}-x^{(j)}||}{2\sigma^2}) = \exp(-\gamma||x^{(i)}-x^{(j)}||^2)$$

Here, $\gamma = \frac{1}{2\sigma^2}$. In the Gaussian function, the standard deviation σ controls the amount of variation or dispersion allowed: the higher the σ (or the lower the γ), the larger the width of the bell, and the wider the range is that data points are allowed to spread out over. Therefore, γ as the **kernel coefficient** determines how strictly or generally the kernel function fits the observations. A large γ implies a small variance allowed and a relatively exact fit on the training samples, which might lead to overfitting. On the other hand, a small γ implies a high variance allowed and a loose fit on the training samples, which might cause underfitting.

Some other common kernel functions include the **polynomial** kernel

$$K(x^{(i)},x^{(j)}) = (x^{(i)}x^{(j)}+\gamma)^d$$

and the **sigmoid** kernel:

$$K(x^{(i)},x^{(j)}) = \tanh(x^{(i)}x^{(j)}+\gamma)$$

In the absence of prior knowledge of the distribution, the RBF kernel is usually preferable in practical usage, as there is an additional parameter to tweak in the polynomial kernel (polynomial degree d) and the empirical sigmoid kernel can perform approximately on a par with the RBF, but only under certain parameters. Hence, we come to a debate between the linear (also considered no kernel) and the RBF kernel given a dataset.

3.5 Choosing between linear and RBF kernels

Of course, linear separability is the rule of thumb when choosing the right kernel to start with. However, most of the time this is very difficult to identify, unless you have sufficient prior knowledge of the dataset, or its features are of low dimensions (1 to 3).

> Some general prior knowledge that is commonly known includes that text data is often linearly separable, while data generated from the XOR function is not.

Now, let's look at the following three scenarios where the linear kernel is favored over RBF.

(IEEE Transactions on Neural Networks, March 2002, Volume 13, pp. 415-425)

Table 3.1: Rules for choosing between linear and RBF kernels

Scenario	Linear	RBF
Prior knowledge	If linearly separable	If nonlinearly separable
Visualizable data of 1 to 3 dimension(s)	If linearly separable	If nonlinearly separable
Both the number of features and number of instances are large.	First choice	
Features >> Instances	First choice	
Instances >> Features	First choice	
Others		First choice

Scenario 1 Both the number of features and the number of instances are large (more than 104 or 105). Since the dimension of the feature space is high enough, additional features as a result of RBF transformation will not provide a performance improvement, but this will increase the computational expense.

Scenario 2 The number of features is noticeably large compared to the number of training samples. Apart from the reasons stated in scenario 1, the RBF kernel is significantly more prone to overfitting.

Scenario 3 The number of instances is significantly large compared to the number of features. For a dataset of low dimension, the RBF kernel will, in general, boost the performance by mapping it to a higher-dimensional space. However, due to the training complexity, it usually becomes inefficient on a training set with more than 106 or 107 samples.

The rules for choosing between linear and RBF kernels can be summarized as Table 3.1:

3.6 Classifying face images with SVM

3.6.1 Building an SVM-based image classifier

There is another SVM classifier, LinearSVC, from scikit-learn. How is it different from SVC? LinearSVC is similar to SVC with linear kernels, but it is implemented based on the liblinear library, which is better optimized than libsvm with the linear kernel, and its penalty function is more flexible.

> In general, training with the LinearSVC model is faster than SVC. This is because the liblinear library with high scalability is designed for large datasets, while the libsvm library with more than quadratic computation complexity is not able to scale well with more than 10^5 training instances. But again, the LinearSVC model is limited to only linear kernels.