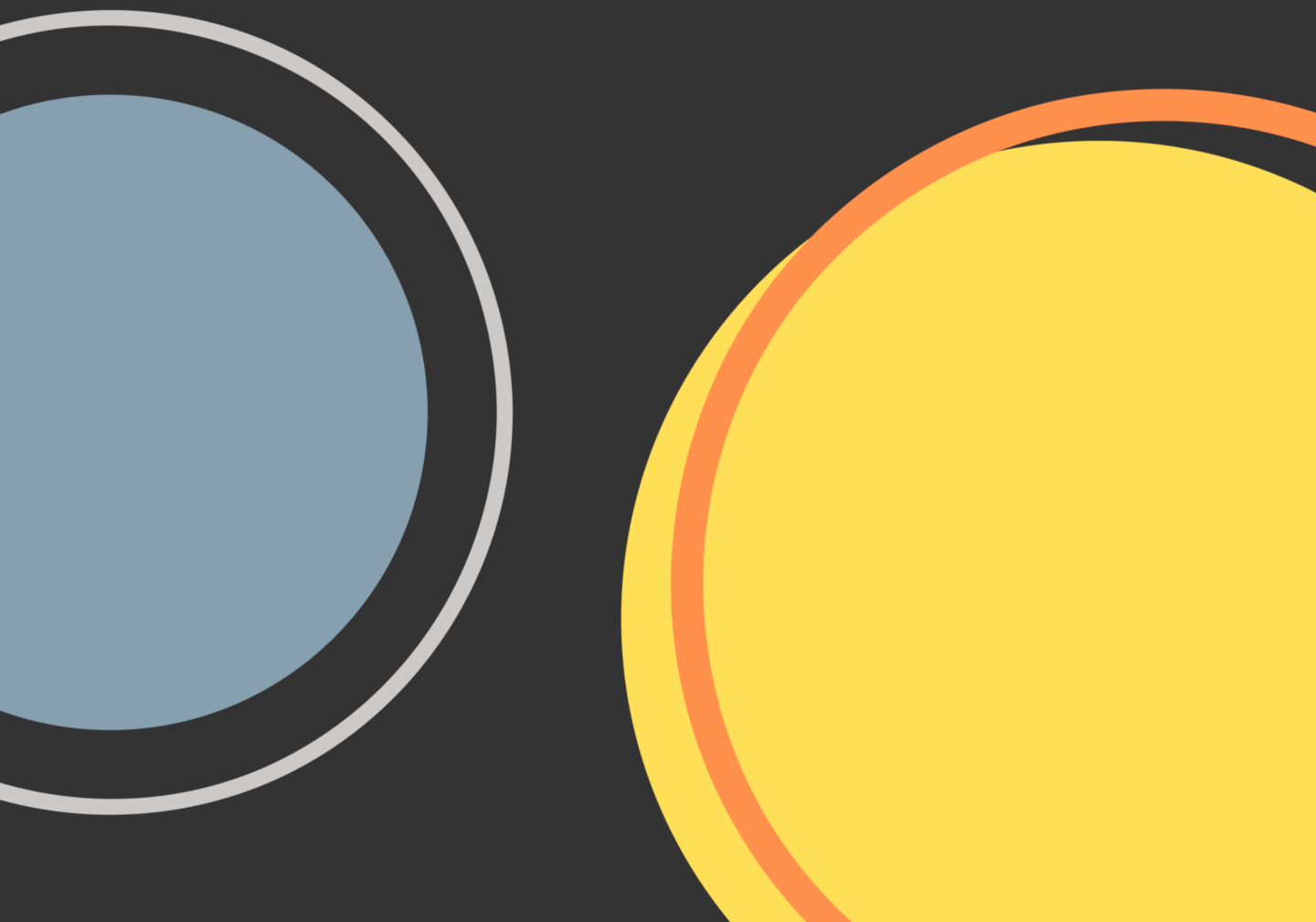
Andriy Burkov

THE HUNDRED-PAGE MACHINE LEARNING BOOK



"All models are wrong, but some are useful."
— George Box

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

8 Advanced Practice

This chapter contains the description of techniques that you could find useful in your practice in some contexts. It's called "Advanced Practice" not because the presented techniques are more complex, but rather because they are applied in some very specific contexts. In many practical situations, you will most likely not need to resort to using these techniques, but sometimes they are very helpful.

8.1 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 soft margin, you can define a cost 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.

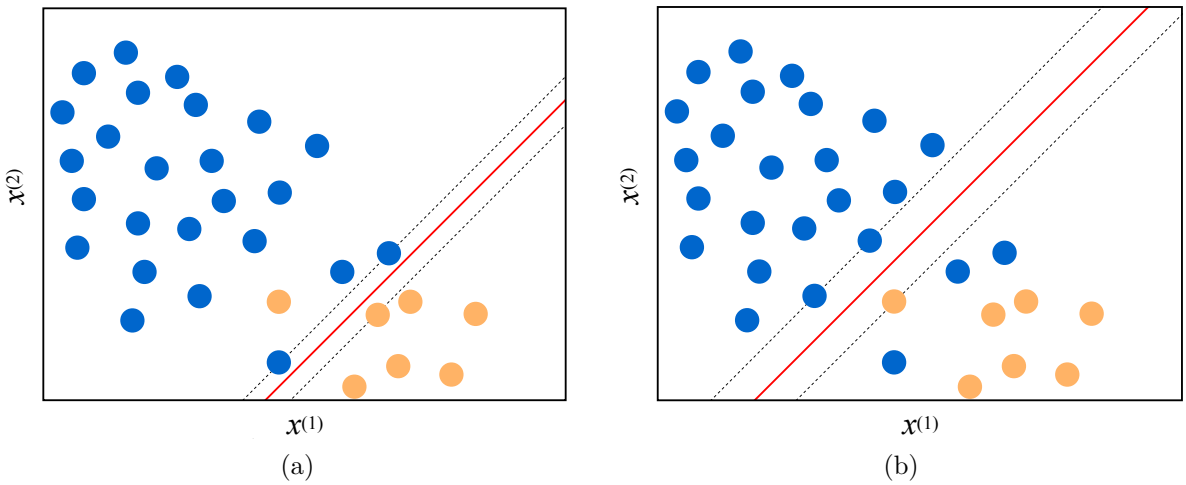


Figure 1: An illustration of an imbalanced problem. (a) Both classes have the same weight; (b) examples of the minority class have a higher weight.

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 1a. This problem is observed for most learning algorithms applied to **imbalanced datasets**.

If you set the cost 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 cost of misclassification of some examples of the majority class, as illustrated in Figure 1b.

Some SVM implementations allow you to provide 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 $\mathcal{S}_k$) 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 a random number in the range $[0, 1]$.

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

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.

8.2 Combining Models

Ensemble algorithms, like Random Forest, typically combine models of the same nature. They boost performance by combining hundreds of weak models. In practice, we can sometimes get an additional performance gain by combining strong models made with different learning algorithms. In this case, we usually use only two or three models.

Three typical ways to combine models are 1) averaging, 2) majority vote and 3) stacking.

Averaging works for regression as well as those classification models that return classification scores. You simply apply all your models—let's call them **base models**—to the input $\mathbf{x}$ and then average the predictions. To see if the averaged model works better than each individual algorithm, you test it on the validation set using a metric of your choice.

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

Stacking 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 and f_2, both predicting the same set of classes. To create a training example $(\hat{\mathbf{x}}_i, \hat{y}_i)$ for the stacked model, set $\hat{\mathbf{x}}_i = [f_1(\mathbf{x}), f_2(\mathbf{x})]$ and $\hat{y}_i = y_i$.

If some of your base models return not just a class, but also a score for each class, you can use these values as features too.

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.

The reason that combining multiple models can bring better performance is that when several uncorrelated strong 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 of a different nature — for example, SVMs 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.

8.3 Training Neural Networks

In neural network training, one challenging aspect is how to convert your data into the input the network can work with. If your input is images, first of all, you have to resize all images so that they have the same dimensions. After that, pixels are usually first standardized and then normalized to the range $[0, 1]$.

Texts have to be tokenized (that is, split into pieces, such as words, punctuation marks, and other symbols). For CNN and RNN, each token is converted into a vector using the one-hot encoding, so the text becomes a list of one-hot vectors. Another, often better way to represent tokens is by using **word embeddings**. For a multilayer perceptron, to convert texts to vectors the bag of words approach may work well, especially for larger texts (larger than SMS messages and tweets).

The choice of specific neural network architecture is a difficult one. For the same problem, like seq2seq learning, there is a variety of architectures, and new ones are proposed almost every year. I recommend researching state of the art solutions for your problem using Google Scholar or Microsoft Academic search engines that allow searching for scientific publications using keywords and time range. If you don't mind working with less modern architecture, I recommend looking for implemented architectures on GitHub and finding one that could be applied to your data with minor modifications.

In practice, the advantage of a modern architecture over an older one becomes less significant as you preprocess, clean and normalize your data, and create a larger training set. Modern neural network architectures are a result of the collaboration of scientists from several labs and companies; such models could be very complex to implement on your own and usually require much computational power to train. Time spent trying to replicate results from a recent scientific paper may not be worth it. This time could better be spent on building the solution around a less modern but stable model and getting more training data.

Once you decided on the architecture of your network, you have to decide on the number of layers, their type, and size. It is recommended to start with one or two layers, train a model and see if it fits the training data well (has a low bias). If not, gradually increase the size of each layer and the number of layers until the model perfectly fits the training data. Once this is the case, if the model doesn't perform well on the validation data (has a high variance), you should add regularization to your model. If, after adding regularization, the model doesn't fit the training data anymore, slightly increase the size of the network. Continue iteratively until the model fits both training and validation data well enough according to your metric.

8.4 Advanced 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 is in the range $[0, 1]$ and it has to be found experimentally by tuning it on the validation data.

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. As you remember from the section about gradient descent in Chapter 4, as the number of epochs increases, the cost decreases. The decreased cost means that the model fits the training data well. However, at some point, 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 an 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 called **data augmentation**. This technique is often used to regularize models that work with images. Once you have your original labeled training set, you can create a synthetic example from an original example by applying various transformations to the original image: zooming it slightly, rotating, flipping, darkening, and so on. You keep the original label in these synthetic examples. In practice, this often results in increased performance of the model.

8.5 Handling Multiple Inputs

Often in practice, you will work with multimodal data. For example, your input could be an image and text and the binary output could indicate whether the text describes this image.

It's hard to adapt **shallow learning** algorithms to work with multimodal data. However, it's not impossible. You could train one shallow model on the image and another one on the text. Then you can use a model combination technique we discussed above.

If you cannot divide your problem into two independent subproblems, you can try to vectorize each input (by applying the corresponding feature engineering method) and then simply 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 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.

8.6 Handling Multiple Outputs

In some problems, you would like to predict multiple outputs for one input. We considered multi-label classification in the previous chapter. 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.

8.7 Transfer Learning

Transfer learning is probably where neural networks have a unique advantage over the shallow models. In transfer learning, you pick an existing model trained on some dataset, and you adapt this model to predict examples from another dataset, different from the one the model was built on. This second dataset is not like holdout sets you use for validation and test. It may represent some other phenomenon, or, as machine learning scientists say, it may come from another statistical distribution.

For example, imagine you have trained your model to recognize (and label) wild animals on a big labeled dataset. After some time, you have another problem to solve: you need to build a model that would recognize domestic animals. With shallow learning algorithms, you do not have many options: you have to build another big labeled dataset, now for domestic animals.

With neural networks, the situation is much more favorable. Transfer learning in neural networks works like this:

1. You build a deep model on the original big dataset (wild animals).
2. You compile a much smaller labeled dataset for your second model (domestic animals).

3. You remove the last one or several layers from the first model. Usually, these are layers responsible for the classification or regression; they usually follow the embedding layer.
4. You replace the removed layers with new layers adapted for your new problem.
5. You "freeze" the parameters of the layers remaining from the first model.
6. You use your smaller labeled dataset and gradient descent to train the parameters of only the new layers.

Usually, there is an abundance of deep models for visual problems available online. You can find one that has high chances to be of use for your problem, download that model, remove several last layers (the quantity of layers to remove is a hyperparameter), add your own prediction layers and train your model.

Even if you don't have an existing model, transfer learning can still help you in situations when your problem requires a labeled dataset that is very costly to obtain, but you can get another dataset for which labels are more readily available. Let's say you build a document classification model. You got the taxonomy of labels from your employer, and it contains a thousand categories. In this case, you would need to pay someone to a) read, understand and memorize the differences between categories and b) read up to a million documents and annotate them.

To save on labeling so many examples, you could consider using Wikipedia pages as the dataset to build your first model. The labels for a Wikipedia page can be obtained automatically by taking the category the Wikipedia page belongs to. Once your first model has learned to predict Wikipedia categories, you can "fine tune" this model to predict the categories of your employer's taxonomy. You will need much fewer annotated examples for your employer's problem than you would need if you started solving your original problem from scratch.

8.8 Algorithmic Efficiency

Not all algorithms capable of solving a problem are practical. Some can be too slow. Some problems can be solved by a fast algorithm; for others, no fast algorithms can exist.

The subfield of computer science called *analysis of algorithms* is concerned with determining and comparing the complexity of algorithms. **Big O notation** is used to classify algorithms according to how their running time or space requirements grow as the input size grows.

For example, let's say we have the problem of finding the two most distant one-dimensional examples in the set of examples $\mathcal{S}$ of size N. One algorithm we could craft to solve this problem would look like this (here and below, in Python):

```
def find_max_distance(S):
    result = None
    max_distance = 0
    for x1 in S:
        for x2 in S:
            if abs(x1 - x2) >= max_distance:
```

```
                max_distance = abs(x1 - x2)
                result = (x1, x2)
    return result
```

In the above algorithm, we loop over all values in $\mathcal{S}$, and at every iteration of the first loop, we loop over all values in $\mathcal{S}$ once again. Therefore, the above algorithm makes N^2 comparisons of numbers. If we take as a unit time the time the comparison, abs and assignment operations take, then the time complexity (or, simply, complexity) of this algorithm is at most $5N^2$. (At each iteration, we have one comparison, two abs and two assignment operations.) When the complexity of an algorithm is measured in the worst case, big O notation is used. For the above algorithm, using big O notation, we write that the algorithm's complexity is $O(N^2)$; the constants, like 5, are ignored.

For the same problem, we can craft another algorithm like this:

```
def find_max_distance(S):
    result = None
    min_x = float("inf")
    max_x = float("-inf")
    for x in S:
        if x < min_x:
            min_x = x
        if x > max_x:
            max_x = x
    result = (max_x, min_x)
    return result
```

In the above algorithm, we loop over all values in $\mathcal{S}$ only once, so the algorithm's complexity is $O(N)$. In this case, we say that the latter algorithm is *more efficient* than the former.

An algorithm is called efficient when its complexity is polynomial in the size of the input. Therefore both $O(N)$ and $O(N^2)$ are efficient because N is a polynomial of degree 1 and N^2 is a polynomial of degree 2. However, for very large inputs, an $O(N^2)$ algorithm can be slow. In the big data era, scientists often look for $O(\log N)$ algorithms.

From a practical standpoint, when you implement your algorithm, you should *avoid using loops whenever possible*. For example, you should use operations on matrices and vectors, instead of loops. In Python, to compute $\mathbf{wx}$, you should write,

```
import numpy
wx = numpy.dot(w,x)
```

and not,

```
wx = 0
for i in range(N):
    wx += w[i]*x[i]
```

Use appropriate data structures. If the order of elements in a collection doesn't matter, use set instead of list. In Python, the operation of verifying whether a specific example $\mathbf{x}$ belongs to $\mathcal{S}$ is efficient when $\mathcal{S}$ is declared as a set and is inefficient when $\mathcal{S}$ is declared as a list.

Another important data structure that you can use to make your Python code more efficient is dict. It is called a dictionary or a hashmap in other languages. It allows you to define a collection of key-value pairs with very fast lookups for keys.

Using libraries is generally most reliable - you should only write your own code if you are a researcher or it is truly required. Scientific Python packages like numpy, scipy, and scikit-learn were built by experienced scientists and engineers with efficiency in mind. They have many methods implemented in the C programming language for maximum efficiency.

If you need to iterate over a vast collection of elements, use *generators* that create a function that returns one element at a time rather than all the elements at once.

Use the *cProfile* package in Python to find inefficiencies in your code.

Finally, when nothing can be improved in your code from the algorithmic perspective, you can further boost the speed of your code by using:

- *multiprocessing* package to run computations in parallel, and
- *PyPy*, *Numba* or similar tools to compile your Python code into fast, optimized machine code.