11 User Aspects of Machine Learning

Dealing with supervised machine learning problems in practice is to a great extent an engineering discipline where many practical issues have to be considered and where the available amount of work-hours to undertake the development is often the limiting resource. To use this resource efficiently, we need to have a well-structured procedure for how to develop and improve the model. Multiple actions can potentially be taken. How do we know which action to take and if it is really worth spending the time implementing it? Is it, for example, worth spending an extra week collecting and labelling more training data, or should we do something else? These issues will be addressed in this chapter. Note that the layout of this chapter is thematic and does not necessarily represent the sequential order in which the different issues should be addressed.

11.1 Defining the Machine Learning Problem

Solving a machine learning problem in practice is an iterative process. We train the model, evaluate the model, and from there suggest an action for improvement and then train the model again, and so on. To do this efficiently, we need to be able to tell whether a new model is an improvement over the previous one or not. One way to evaluate the model after each iteration would be to put it into production (for example running a traffic-sign classifier in a self-driving car for a few hours). Besides the obvious safety issues, this evaluation procedure would be very time consuming and cumbersome. It would most likely also be inaccurate since it could still be hard to tell whether the proposed change was an actual improvement or not.

A better strategy is to automate this evaluation procedure without the need to put the model into production each time we want to evaluate its performance. We do this by putting aside a *validation* dataset and a *test* dataset and evaluate the performance using a scalar *evaluation metric*. The validation and test datasets together with the evaluation metric will define the machine learning problem that we are solving.

Training, Validation, and Test Data

In Chapter 4, we introduced the strategy of splitting the available data into training data, validation data, and test data, as repeated in Figure 11.1.

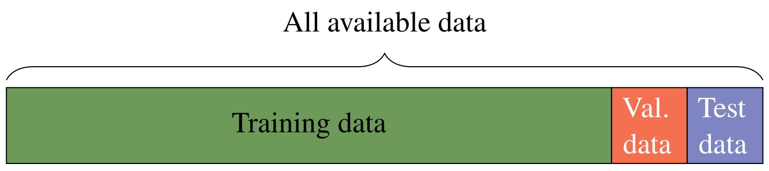


Figure 11.1: Splitting the data into training data, hold-out validation data, and test data.

- **Training data** is used for training the model.
- **Hold-out validation data** is used for comparing different model structures, choosing hyperparameters of the model, feature selection, and so on.
- **Test data** is used to evaluate the performance of the final model.

If the amount of available data is small, it is possible to perform k-fold cross-validation instead of putting aside hold-out validation data; the idea of how to use it in the iterative procedure is unchanged. To get a final estimate of the performance, test data is used.

In the iterative procedure, the hold-out validation data (or k-fold cross-validation) is used to judge if the new model is an improvement over the previous model. During this validation stage, we can also choose to train several new models. For example, if we have a neural network model and are interested in the number of hidden units to use in a certain layer, we can train several models, each with a different choice of hidden units. Afterwards, we pick the one that performs best on the validation data. The number of hidden units in a certain layer is one example of a hyperparameter. If we have more hyperparameteters that we want to evaluate, we can perform a grid search over these parameters. In Section 5.6, we discuss hyper-parameter optimisation in more detail.

Eventually, we will effectively have used the validation data to compare many models. Depending on the size of the validation data, we might risk picking a model that does particularly well on the validation data in comparison to completely unseen data. To detect this and to get a fair estimate of the actual performance of a model, we use the test data, which has been used neither during training nor validation. If the performance on the validation data is substantially better than the performance on the test data, we have overfitted on the validation data. The easiest solution in that case would be to extend the size of the validation data. The test data should not be used repeatedly as a part of the training and model selection procedure. If we start making major decisions based on our test data, then the model will be adapted to the test data, and we can no longer trust that the test performance is an objective measure of the actual performance of our model.

It is important that both the validation data and the test data always come from the same data distribution, namely the data distribution that we are expecting to see when we put the model into production. If they do not stem from the same distribution, we are validating and improving our model towards something that is not represented in the test data and hence are 'aiming for the wrong target'. Usually, the training data is also expected to come from the same data distribution as the test and validation data, but this requirement can be relaxed if we have good reasons to do so. We will discuss this further in Section 11.2.

When splitting the data into training, validation, and test, *group leakage* could be a potential problem. Group leakage can occur if the data points are not really stochastically independent but are ordered into different groups. For example, in the medical domain many, X-ray images may belong to the same patient. In this case, if we do a random split over the images, different images belonging to the same patient will most likely end up both in the training and the validation set. If the model learns the properties of a certain patient, then the performance on the validation data might be better than what we could expect in production.

The solution to the group leakage problem is to do group partitioning. Instead of doing a random split over the data points, we do the split over the groups that the data points belong to. In the medical example above, that would mean that we do a random split of the patients rather than of the medical images. By this, we make sure that the images for a certain patient only end up in one of the datasets, and the leakage of unintentional information from the training data to the validation and test data is avoided.

Even though we advocate the use of validation and test data to improve and assess the performance of the model, we should eventually also evaluate the performance of the model in production. If we realise that the model is performing systematically worse in production than on the test data, we should try to find the reason for why this is the case. If possible, the best way of improving the model is to update the test data and validation data such they actually represent what we expect to see in production.

Size of Validation and Test Datasets

How much data should we set aside as hold-out validation data and test data, or should we even avoid setting aside hold-out validation data and use k-fold cross-validation instead? This depends on how much data we have available, what performance difference we plan to detect, and how many models we plan to compare. For example, if we have a classification model with a 99.8% accuracy and want to know if a new model is even better, a validation dataset of 100 data points will not be able to tell that difference. Also, if we plan to compare many (say, hundreds or more) different hyperparameter values and model structures using 100 validation data points, we will most likely overfit to that validation data.

If we have, say, 500 data points, one reasonable split could be 60%–20%–20% (that is 300–100–100 data points) for training–validation–test. With such a small validation dataset, we cannot afford to compare several hyperparameter values and model structures or to detect an improvement in accuracy of 0.1%. In this situation, we are probably better off using k-fold cross-validation to decrease the risk of overfitting to the validation data. Be aware, however, that the risk of overfitting the training data still exists even with k-fold cross-validation. We also still need to set aside test data if we want a final unbiased estimate of the performance.

Many machine learning problems have substantially larger datasets. Assume we have a dataset of 1 000 000 data points. In this scenario, one possible split could be 98%–1%–1%, that is, leaving 10 000 data points for validation and test, respectively, unless we really care about the very last decimals in performance. Here, k-fold cross-validation is of less use in comparison to the scenario with just 500 data points, since having all 99% = 98% + 1% (training + validation) available for training would make a small difference in comparison to using 'only' 98%. Also, the price for training k models (instead of only one) with this amount of data would be much higher.

Another advantage of having a separate validation dataset is that we can allow the training data to come from a slightly different distribution than the validation and test dataset, for example if that would enable us to find a much larger training dataset. We will discuss this more in Section 11.2.

Single Number Evaluation Metric

In Section 4.5, we introduced additional metrics besides the misclassification rate, such as precision, recall, and F1-score for evaluating binary classifiers. There is no unique answer to which metric is the most appropriate. What metric to pick is rather a part of the problem definition. To improve the model quickly and in a more automated fashion, it is advisable to agree on a single number evaluation metric, especially if a larger team of engineers is working on the problem.

The single number evaluation metric together with the validation data are what defines the supervised machine learning problem. Having an efficient procedure in place where we can evaluate the model on the hold-out validation data (or by k-fold cross-validation) using the metric allows us to speed up the iterations since we can quickly see if a proposed change to the model improves the performance or not. This is important in order to manage an efficient workflow of trying out and accepting or rejecting new models.

That being said, beside the single number evaluation metric, it is useful to monitor other metrics as well to reveal the tradeoffs being made. For example, we might develop the model with different end users in mind who care more or less about different metrics, but for practical reasons, we only train one model to accommodate them all. If we, based on these tradeoffs, realise that the single number evaluation metric we have chosen does not favour the properties we want a good model to have, we can always change that metric.

Baseline and Achievable Performance Level

Before working with the machine learning problem, it is a good idea to establish some reference points for the performance level of the model. A baseline is a very simple model that serves as a lower expected performance level. A baseline can, for example, be to randomly pick an output value y_i from the training data and use that as the prediction. Another baseline for the regression problem is to take the mean of all output values in the training data and use that as the prediction. A corresponding baseline for a classification problem is to pick the most common class among class labels in the training data and use that for the prediction. For example, if we have a binary classification problem with 70% of the training data belonging to one class and 30% belonging to the other class, and we have chosen the accuracy as our performance metric, the accuracy for that baseline is 70%. The baseline is a lower threshold on the performance. We know that the model has to be better than this baseline.

Hopefully, the model will perform well beyond the naive baselines stated above. In addition, it is also good to define an achievable performance which is on par with the maximum performance we can expect from the model. For a regression problem, this performance is in theory limited by the irreducible error presented in Chapter 4, and for classification problems, the analogous concept is the so-called Bayes error rate. In practice, we might not have access to these theoretical bounds, but there are a few strategies for estimating them. For supervised problems that that are easily solved by human annotators, the human-level performance can serve as the achievable performance. Consider for example an image classification problem. If humans can identify the correct class with an accuracy of 99%, that serves as a reference point for what we can expect to achieve from our model. The achievable performance can also be based on what other state-of-the-art models on the same or a similar problem achieve. To compare the performance with the achievable performance gives us a reference point to assess the quality of the model. Also, if the model is close to the achievable performance, we might not be able to improve our model further.

11.2 Improving a Machine Learning Model

As already mentioned, solving a machine learning problem is an iterative procedure where we train, evaluate, and suggest actions for improvement, for instance by changing some hyperparameters or trying another model. How do we start this iterative procedure?

Try Simple Things First

A good strategy is to try simple things first. This could, for example, be to start with basic methods like k-NN or linear/logistic regression. Also, do not add extra adds-on like regularisation for the first model – this will come at a later stage when the basic model is up and running. A simple thing can also be to start with an already existing solution to the same or a similar problem, which you trust. For example, when building an image classifier, it can be simpler to start with an existing pretrained neural network and fine-tune one rather than handcrafting features from these images to be used with k-NN. Starting simple can also mean to consider only a subset of the available training data for the first model and then retrain the model on all the data if it looks promising. Also, avoid doing more data pre-processing than necessary for your first model, since we want to minimise the risk of introducing bugs early in the process. This first step not only involves writing code for learning your first simple model but also code for evaluating it on your validation data using your single number evaluation metric.

Trying simple things first allows us to start early with the iterative procedure of finding a good model. This is important since it might reveal important aspects of the problem formulation that we need to re-think before it makes sense to proceed with more complicated models. Also, if we start with a low-complexity model, it also reduces the risk of ending up with a too-complicated model, when a much simpler model would have been just as good (or even better).

Debugging your Model

Before proceeding, we should make sure that the code we have is producing what we are expecting it to do. The first obvious check is to make sure that the code runs without any errors or warnings. If it does not, use a debugging tool to spot the error. These are the easy bugs to spot.

The trickier bugs are those where the code is syntactically correct and runs without warnings but is still not doing what we expect it to do. The procedure of how to debug this depends on the model you have picked, but there are a few general tips:

- *Compare with baseline.* Compare you model performance on validation data with the baselines you have stated (see Section 11.1). If we do not manage to beat these baselines or are even worse than them, the code for training and evaluating the model might not be working as expected.
- *Overfit a small subset.* Try to overfit the model on a very small subset (e.g. as small as two data points) of the training data and make sure that we can achieve the best possible performance evaluated on that training data subset. If it is a parametric model, also aim for the lowest possible training cost.

When we have verified to the best of our ability that the code is bug-free and does what it is expected to do, we are ready to proceed. There are many actions that could be taken to improve the model – for example, changing the type of model, increasing/decreasing model complexity, changing input variables, collecting more data, correcting mislabelled data (if there is any), etc. What should we do next? Two possible strategies for guiding us to meaningful actions to improve the solution are by trading *training error and generalisation gap* or by applying *error analysis*.

Training Error vs. Generalisation Gap

With the notation from Chapter 4, the training error E_{train} is the performance of the model on training data, and the validation error $E_{\text{hold-out}}$ is the performance on hold-out validation data. In the validation step, we are interested in changing the model such that $E_{\text{hold-out}}$ is minimised. We can write the validation error as a sum of the training error and the generalisation gap:

$$E_{\text{hold-out}} = E_{\text{train}} + \underbrace{(E_{\text{hold-out}} - E_{\text{train}})}_{\approx \text{ generalisation gap}}. \tag{11.1}$$

In words, the generalisation gap is approximated by the difference between the validation error $E_{\text{hold-out}}$ and the training error E_{train}.[1]

We can easily compute the training error E_{train} and the generalisation gap $E_{\text{hold-out}} - E_{\text{train}}$; we just have to evaluate the error on the training data and validation data, respectively. By computing these quantities, we can get good guidance for what changes we may consider for the next iteration.

As we discussed in Chapter 4, if the training error is small and the generalisation gap is big (E_{train} small, $E_{\text{hold-out}}$ big), we have typically overfitted the model. The opposite situation, big training error and small generalisation gap (both E_{train} and $E_{\text{hold-out}}$ big), typically indicates underfitting.

If we want to reduce the generalisation gap $E_{\text{hold-out}} - E_{\text{train}}$ (reduce overfitting), the following actions can be explored:

- *Use a less flexible model.* If we have a very flexible model, we might start overfitting to the training data, that is, E_{train} is much smaller than $E_{\text{hold-out}}$. If we use a less flexible model, we also reduce this gap.

- *Use more regularisation.* Using more regularisation will reduce the flexibility of the model and hence also reduce the generalisation gap. Read more about regularisation in Section 5.3

- *Early stopping* For models that are trained iteratively, we can stop the training before reaching the minimum. One good practice is to monitor $E_{\text{hold-out}}$ during training and stop if it starts increasing; see Example 5.7.

- *Use bagging*, or use more ensemble members if we already are using it. Bagging is a method for reducing the variance of the model, which typically also means that we reduce the generalisation gap; see more in Section 7.1.

- *Collect more training data.* If we collect more training data, the model is less prone to overfit that extended training dataset and is forced to only focus on aspects which generalise to the validation data.

[1]This can be related to (4.11), if approximating $\bar{E}_{\text{train}} \approx E_{\text{train}}$ and $\bar{E}_{\text{new}} \approx E_{\text{hold-out}}$. If we use k-fold cross validation instead of hold-out validation data, we use $\bar{E}_{\text{new}} \approx E_{k\text{-fold}}$ when computing the generalisation gap.

If we want to reduce the training error E_{train} (reduce underfitting), the following actions can be considered:

- *Use a more flexible model* that is able to fit the training data better. This can be changing a hyperparameter in the model we are considering – for example decreasing k in k-NN– or changing the model to a more flexible one – for example by replacing a linear regression model by a deep neural network.
- *Extend the set of input variables.* If we suspect that there are more input variables that carry information, we might want to extend the data with these input variables.
- *Use less regularisation.* This can, of course, only be applied if regularisation is used at all.
- *Train the model for longer.* For models that are trained iteratively, we can reduce E_{train} by training for longer.

It is usually a balancing act between reducing the training error and the generalisation gap, and measures to decrease one of them might result in an increase of the other. This balancing act is also related to the bias–variance tradeoff discussed in Example 4.3.

We summarise the above discussion in Figure 11.2. Fortunately, evaluating E_{train} and $E_{\text{hold-out}}$ is cheap. We only have to evaluate the model on the training data and the validation data, respectively. Yet, it gives us good advice on what actions to take next. Besides suggesting what action to explore next, this procedure also tells us what *not* to do: If $E_{\text{train}} \gg E_{\text{hold-out}} - E_{\text{train}}$, collecting more training data will most likely not help. Furthermore, if $E_{\text{train}} \ll E_{\text{hold-out}} - E_{\text{train}}$, a more flexible model will most likely not help.

Learning Curves

Of the different methods mentioned to reduce the generalisation gap, collecting more training data is often the simplest and most reliable strategy. However, in contrast to the other techniques, collecting and labelling more data is often significantly more time consuming. Before collecting more data, we would like to tell how much improvement we can expect. By plotting learning curves, we can get such an indication.

In a learning curve, we train models and evaluate E_{train} and $E_{\text{hold-out}}$ using different sizes of training dataset. For example, we can train different models with 10%, 20%, 30%, ... of the available training data and plot how E_{train} and $E_{\text{hold-out}}$ vary with the amount of training data. By extrapolating these plots, we can get an indication of the improvement on the generalisation gap that we can expect by collecting more data.

In Figure 11.3, two sets of learning curves for two different scenarios are depicted. First, note that previously we evaluated E_{train} and $E_{\text{hold-out}}$ only for the rightmost

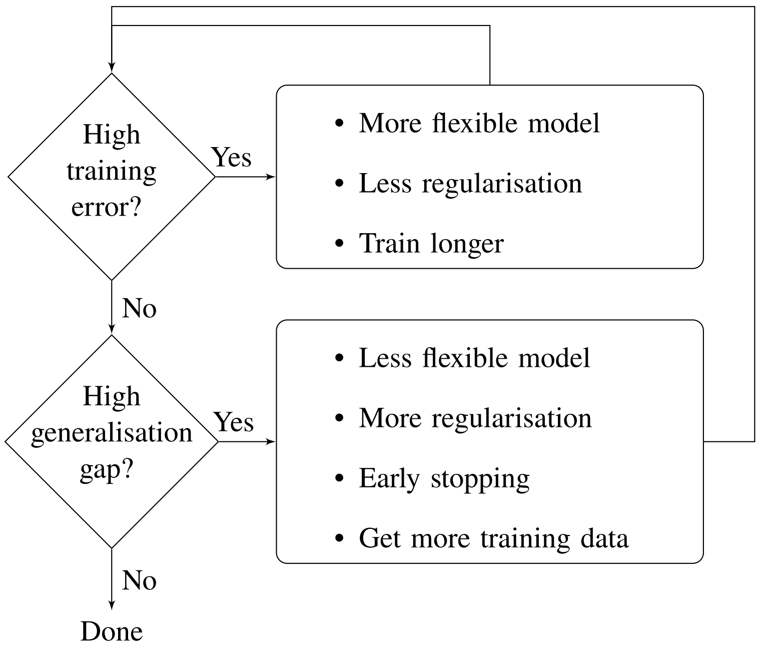


Figure 11.2: The iterative procedure of improving a model based on the decomposition of the validation error into the training error and generalisation gap.

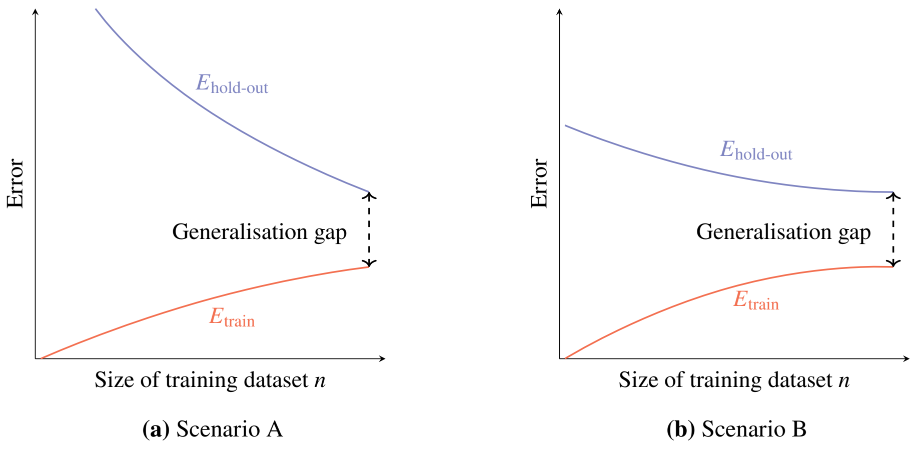


Figure 11.3: Learning curve for two different scenarios. In Scenario A, we can expect an improvement in the generalisation gap by collecting more training data, whereas in Scenario B, we are less likely to see an immediate improvement by adding more data.

point in these graphs using all our available data. However, these plots reveal more information about the impact of the training dataset size on the performance of the model. In the two scenarios depicted in Figure 11.3, we have the same values for E_{train} and $E_{\text{hold-out}}$ if we train the model using all the available training data. However, by extrapolating the learning curves for E_{train} and $E_{\text{hold-out}}$ in these two scenarios, it is likely that in Scenario A, we can reduce the generalisation gap much more than we can in Scenario B. Hence, in Scenario A, collecting more training data is more beneficial than in Scenario B. By extrapolating the learning curves, you can also answer the question of how much extra data is needed to reach some desired performance. Plotting these learning curves does not require much extra effort we only have to train a few more models on subsets of our training data. However, it can provide valuable insight on whether it is worth the extra effort of collecting more training data and how much extra data you should collect.

Error Analysis

Another strategy to identify actions that can improve the model is to perform *error analysis*. Below we only describe error analysis for classification problems, but the same strategy can be applied to regression problems as well.

In error analysis, we manually look at a subset, say 100 data points, of the validation data that the model classified incorrectly. Such an analysis does not take much time but might give valuable clues as to what type of data the model is struggling with and how much improvement we can expect by fixing these issues. We illustrate the procedure with an example.

Example 11.1 Error analysis applied to vehicle detection

Consider a classification problem of detecting cars, bicycles, and pedestrians in an image. The model takes an image as input and outputs one of the four classes `car`, `bike`, `pedestrian`, or `other`. Assume that the model has a classification accuracy of 90% on validation data.

When looking at a subset of 100 images that were misclassified in the validation data, we make the following observations:

- All 10 images of class `pedestrian` that were incorrectly classified as `bike` where taken in dark conditions with the pedestrian being equipped with safety reflectors.
- 30 images were substantially tilted.
- 15 images were mislabelled.

From this observation we can conclude:

- If we launch a project for improving the model to classify pedestrians with safety reflectors as `pedestrian` and not incorrectly as `bike`, an improvement of at most a ~1% (a tenth of the 10% classification error rate) can be expected.
- If we improve the performance on tilted images, an improvement of at most ~3% can be expected.
- If we correct all mislabelled data, an improvement of at most ~1.5% can be expected.

Following the example, we get an indication on what improvement we can expect by tackling these three issues. These numbers should be considered as the maximal possible improvement. To prioritise which aspect to focus on, we should also consider what strategies are available for improving them, how much progress we expect to make applying these strategies, and how much effort we would have to invest fixing these issues.

For example, to improve the performance on tilted images, we could try to extend the training data by augmenting it with more tilted images. This strategy could be investigated without too much extra effort by augmenting the training data with tilted versions of the training data points that we already have. Since this could be applied fairly quickly and has a maximal performance increase of 3%, it seems to be a good thing to try out.

To improve the performance on the images of pedestrians with safety reflectors, one approach would be to collect more images in dark conditions of pedestrians with safety reflector. This obviously requires some more manual work, and it can be questioned if it is worth the effort since it would only give a performance improvement of at most 1%. However, for this application, you could also argue that this 1% is of extra importance.

Regarding the mislabelled data, the obvious action to take to improve on this issue is to manually go through the data and correct these labels. In the example above, we may say it is not quite worth the effort to get an improvement of 1.5%. However, assume that we have improved the model with other actions to an accuracy of 98.0% on validation data and that still 1.5% of the total error is due to mislabelled data; this issue is now quite relevant to address if we want to improve the model further. Remember, the purpose of the validation data is to choose between different models. This purpose is degraded when the majority of the reported error on validation is due to incorrectly labelled data rather than the actual performance of the model.

There are two levels of ambition for correcting the labels:

(i) Go through all data points in the validation/test data and correct the labels.

(ii) Go through all data points, including the training data, and correct the labels.

The advantage of approach (i), in comparison to approach (ii), is the lower amount of work it requires. Assume, for example, that we have made a 98%–1%–1% split of training-validation-test data. Then there is 50 times less data to process in comparison to approach (ii). Unless the mislabeling is systematic, correcting the labels in the training data will not necessarily pay off. Also, note that correcting labels in only test and validation data does not necessarily increase the performance of a model in production, but it will give us a fairer estimate of the actual performance of the model.

Applying the data cleaning to validation and test data only, as suggested in approach (i), will result in the training data coming from a slightly different distribution than the validation and test data. However, if we are eager to correct the mislabelled data in the training data as well, a good recommendation would still be to start correcting validation and test data only, and then use the techniques in the following section to see how much extra performance we can expect by cleaning the training data as well before launching that substantially more labor-intensive data cleaning project.

In some domains, for example medical imaging, the labelling can be difficult, and two different lablers might not agree on the label for the very same data point. This agreement between labellers is also called inter-rater reliability. It can be wise to check this metric on a subset of your data by assigning multiple labellers for that data. If the inter-rater reliability is low, you might want to consider addressing this issue. This can, for example, be done by assigning multiple labellers to all data points in the validation and test data and, if you can afford the extra labelling cost, also to the training data. For the samples where labellers do not agree, the majority vote can be used for these labels.

Mismatched Training and Validation/Test Data

As already pointed out in Chapter 4, we should strive to let the training data come from the same distribution as the validation and test data. However, there are situations where, for different reasons, we can accept the training data coming from a slightly different distribution than the validation and test data. One reason was presented in the previous section where we chose to correct mislabelled data in the validation and test data but not necessarily to invest the time to do the same correction to the training data.

Another reason for mismatched training and validation/test data is that we might have access to another, substantially larger dataset which comes from a slightly different distribution than the data we care about but is similar enough that the advantage of having a larger training data outweighs the disadvantage of that data mismatch. This scenario is further described in Section 11.3.

If we have a mismatch between training data and validation/test data, that mismatch contributes to yet another error source of the final validation error $E_{\text{hold-out}}$ that we care about. We want to estimate the magnitude of that error source. This can be done by revising the training–validation–test data split. From the training data, we can carve out a separate *training-validation* dataset, see Figure 11.4. That dataset is neither used for training nor for validation. However, we do evaluate the performance of our model on that dataset as well. As before, the remaining part of the training data is used for training, the validation data is used for comparing different model structures, and test data is used for evaluating the final performance of the model.

This modified data split also allows us to revise the decomposition in (11.1) to include this new error source:

$$E_{\text{hold-out}} = E_{\text{train}} + \underbrace{(E_{\text{train-val}} - E_{\text{train}})}_{\approx \text{ generalisation gap}} + \underbrace{(E_{\text{hold-out}} - E_{\text{train-val}})}_{\approx \text{ train-val mismatch}}, \tag{11.2}$$

where $E_{\text{train-val}}$ is the performance of the model on the new training-validation data and where, as before, $E_{\text{hold-out}}$ and E_{train} are the performances on the validation and

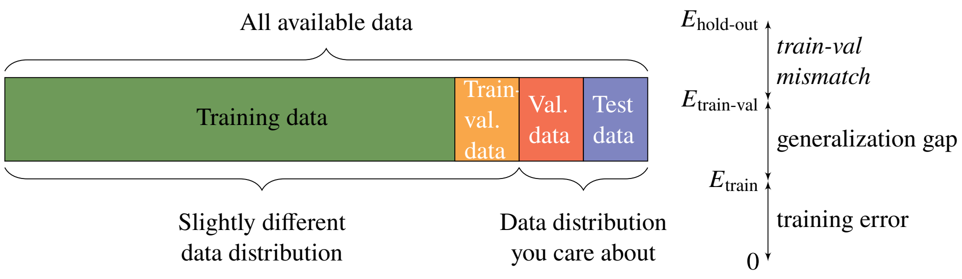


Figure 11.4: Revising the training–validation–test data split by carving out a separate training-validation dataset from the training data.

training data, respectively. With this new decomposition, the term $E_{\text{train-val}} - E_{\text{train}}$ is an approximation of the generalisation gap, that is, how well the model generalises to unseen data *of the same distribution* as the training data, whereas the term $E_{\text{hold-out}} - E_{\text{train-val}}$ is the error related to the training-validation data mismatch. If the term $E_{\text{hold-out}} - E_{\text{train-val}}$ is small in comparison to the other two terms, it seems likely that the training-validation data mismatch is not a big problem and that it is better to focus on techniques reducing the other training error and the generalisation gap as we talked about earlier. On the other hand, if $E_{\text{hold-out}} - E_{\text{train-val}}$ is significant, the data mismatch does have an impact, and it might be worth investing time reducing that term. For example, if the mismatch is caused by the fact that we only corrected labels in the validation and test data, we might want to consider correcting labels for the training data as well.

11.3 What If We Cannot Collect More Data?

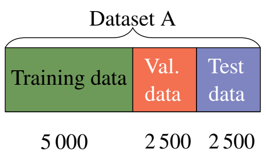
We have seen in Section 11.2 that collecting more data is a good strategy to reduce the generalisation gap and hence reduce overfitting. However, collecting labelled data is usually expensive and sometimes not even possible. What can we do if we cannot afford to collect more data but still want to benefit from the advantages that a larger dataset would give? In this section, a few approaches are presented.

Extending the Training Data with Slightly Different Data

As already mentioned, there are situations where we can accept the training data coming from a slightly different distribution than the validation and test data. One reason to accept this is if we then would have access to a substantially larger training dataset.

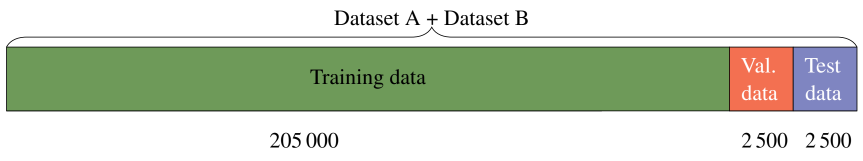
Consider a problem with 10 000 data points, representing the data that we would also expect to get when the model is deployed in production. We call this Dataset A. We also have another dataset with 200 000 data points that come from a slightly different distribution but which is similar enough that we think exploiting information from that data can improve the model. We call this Dataset B. Some options to proceed would be the following:

- **Option 1** Use only Dataset A and split it into training, validation, and test data.



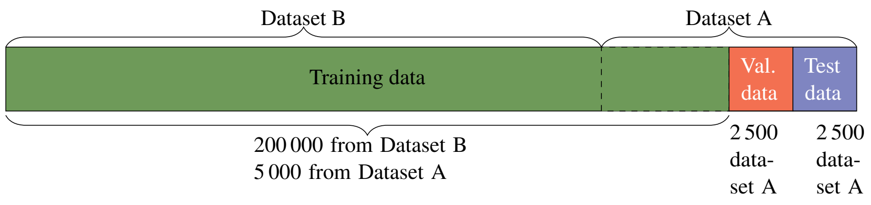
The advantage of this option is that we only train, validate, and evaluate on Dataset A, which is also the type of data that we want our model to perform well on. The disadvantage is that we have quite few data points, and we do not exploit potentially useful information in the larger Dataset B.

- **Option 2** Use both Dataset A and Dataset B. Randomly shuffle the data and split it into training, validation, and test data.



The advantage over option 1 is that we have a lot more data available for training. However, the disadvantage is that we mainly evaluate the model on data from Dataset B, whereas we want our model to perform well on data from Dataset A.

- **Option 3** Use both Dataset A and Dataset B. Use data points from Dataset A for validation data and test data and some in the training data. Dataset B only goes into the training data.



Similar to option 2, the advantage is that we have more training data in comparison to option 1, and in contrast to option 2, we now evaluate the model on data from Dataset A, which is the data we want our model to perform well on. However, one disadvantage is that the training data no longer has the same distribution as the validation and test data.

From these three options, we would recommend either option 1 or 3. In option 3, we exploit the information available in the much larger Dataset B but evaluate only on the data we want the model to perform well on (Dataset A). The main disadvantage with option 3 is that the training data no longer comes from the same distribution as the validation data and test data. In order to quantify how big an impact this mismatch has on the final performance, the techniques described in Section 11.2 can be used. To push the model to do better on data from Dataset A during training, we can also consider giving data from Dataset A a higher weight in the cost function than data from Dataset B, or simply upsample the data points in Dataset A that belong to the training data.

There is no guarantee that adding Dataset B to the training data will improve the model. If that data is very different from Dataset A, it can also do harm, and we might be better off just using Dataset A as suggested in option 1. Using option 2 is generally not recommended since we would then (in contrast to option 1 and 3) evaluate our model on data which is different from that which we want it to perform well on. Hence, if the data in Dataset A is scarce, prioritise putting it into the validation and test datasets and, if we can afford it, some of it in the training data.

Data Augmentation

Data augmentation is another approach to extending the training data without the need to collect more data. In data augmentation, we construct new data points by duplicating the existing data with invariant transformations. This is especially common for images, where such invariant transformations can be cropping, rotation, vertical flipping, noise addition, colour shift, and contrast change. For example, if we vertically flip an image of a cat, it still displays a cat; see the examples Figure 11.5. One should be aware that some objects are not invariant to some of these operations. For example, a flipped image of a digit is not a valid transformation. In some cases such operations can even make the object resemble an object from another class. If we were to flip an image of a '6' both vertically and horizontally, that image would resemble a '9'. Hence, before applying data augmentation, we need to know and understand the problem and the data. Based on that knowledge we can identify valid invariants and suggest which transformations that can be applied to augment the data that we already have.

To apply data augmentation offline before the training would increase the required amount of storage and is hence only recommended for small datasets. For many models and training procedures, we can instead apply it online during training. For example, if we train a parametric model using stochastic gradient descent (see Section 5.5), we can apply the transformation directly on the data that goes into the current mini-batch without the need to store the transformed data.

Transfer Learning

Transfer learning is yet another technique that allows us to exploit information from more data than the dataset we have. In transfer learning we use the knowl-

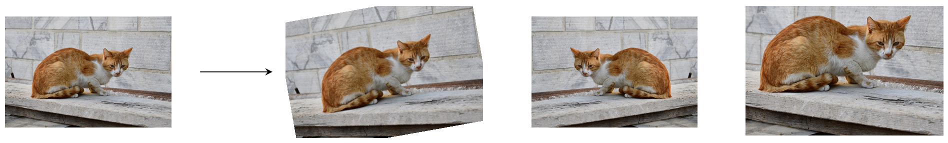


Figure 11.5: Example of data augmentation applied to images. An image of a cat has been reproduced by tilting, vertical flipping and cropping.

source: Image of cat is reprinted from `https://commons.wikimedia.org/wiki/File:Stray_Cat,_Nafplio.jpg` and is in the public domain.

edge from a model that has been trained on a different task with a different dataset and then apply that model in solving a different, but slightly related, problem.

Transfer learning is especially common for sequential model structures such as the neural network models introduced in Chapter 6. Consider an application where we want to detect whether a certain type of skin cancer is malignant or benign, and for this task we have 100 000 labelled images of skin cancer. We call this Task A. Instead of training the full neural network from scratch on this data, we can reuse an already pretrained network from another image classification task (Task B), which preferably has been trained on a much larger dataset, not necessarily containing images even resembling skin cancer tumors. By using the weights from the model trained for Task B and only train the last few layers on the data for Task A, we can get a better model than if the whole model would had been trained on only the data for Task A. The procedure is also displayed in Figure 11.6. The intuition is that the layers closer to the input accomplish tasks that are generic for all types of images, such as extracting lines, edges and corners in the image, whereas the layers closer to the output are more specific to the particular problem.

In order for transfer learning to be applicable, we need the two tasks to have the same type of input (in the example above, images of the same dimension). Further, for transfer learning to be an attractive option, the task that we transfer from should have been trained on substantially more data than the task we transfer to.

Learning from Unlabelled Data

We can also improve our model by learning from an additional (typically much larger) dataset without outputs, so called unlabelled data. Two families of such methods are semi-supervised learning and self-supervised learning. In our description below, we call our original dataset with both inputs and output Dataset A and our unlabelled dataset Dataset B.

In semi-supervised learning, we formulate and train a generative model for the inputs in both Dataset A and Dataset B. The generative model of the inputs

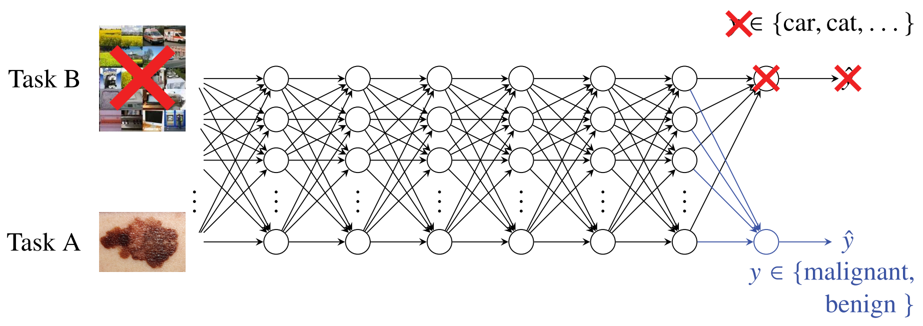


Figure 11.6: In transfer learning, we reuse models that have been trained on a different task than the one we are interested in. Here we reuse a model which has been trained on images displaying all sorts of classes, such as cars, cats, and computers, and later train only the last few layers on the skin cancer data which is the task we are interested in.
source: Skin cancer sample is reprinted from `https://visualsonline.cancer.gov/details.cfm?imageid=9186` and is in the public domain.

and the supervised model for Dataset A are then trained jointly. The idea is that the generative model on the inputs, which is trained on the much larger dataset, improves the performance of the supervised task. Semi-supervised learning is further described in Chapter 10.

In self-supervised learning, we instead use Dataset B in a very similar way as we do in transfer learning described previously. Hence, we pretrain the model based on Dataset B and then fine-tune that model using Dataset A. Since Dataset B does not contain any outputs, we automatically generate outputs for Dataset B and pretrain our model with these generated outputs. The automatically generated outputs can, for example, be a subset of the input variables or a transformation thereof. As in transfer learning, the idea is that the pretrained model learns to extract features from the input data which then can be used to improve the training of the supervised task that we are interested in. Also, if we don't have an additional unlabelled Dataset B, we can also use self-supervised learning on the inputs in Dataset A as the pretraining before training on the supervised task on that dataset.

11.4 Practical Data Issues

Besides the amount and distribution of data, a machine learning engineer may also face other data issues. In this section, we will discuss some of the most common ones; outliers, missing data, and if some features can be removed.

Outliers

In some applications, a common issue is *outliers*, meaning data points whose outputs do not follow the overall pattern. Two typical examples of outliers are sketched in Figure 11.7. Even though the situation in Figure 11.7 looks simple, it can be quite hard to find outliers when the data has more dimensions and is harder to

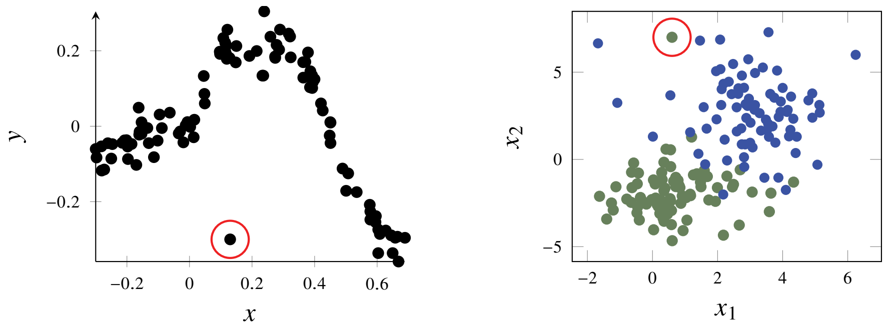


Figure 11.7: Two typical examples of outliers (marked with red circle) in regression (left) and classification (right), respectively.

visualise. The error analysis discussed in Section 11.2, which amounts to inspecting misclassified data points in the validation data, is a systematic way to discover outliers.

When facing a problem with outliers, the first question to ask is whether the outliers are meant to be captured by the model or not. Do the encircled data points in Figure 11.7 describe an interesting phenomenon that we would like to predict, or are they irrelevant noise (possibly originating from a poor data collection process)? The answer to this question depends on the actual problem and ambition. Since outliers by definition (no matter their origin) do not follow the overall pattern, they are typically hard to predict.

If the outliers are not of any interest, the first thing we should do is to consult the data provider and identify the reason for the outliers and if something could be changed in the data collection process to avoid these outliers, for example replacing a malfunctioning sensor. If the outliers are unavoidable, there are basically two approaches one could take. The first approach is to simply delete (or replace) the outliers in the data. Unfortunately this means that one has to first find the outliers, which can be hard, but sometimes some thresholding and manual inspection (that is, look at all data points whose output value is smaller/larger than some value) can help. Once the outliers are removed from the data, one can proceed as usual. The second approach is to instead make sure that the learning algorithm is robust against outliers, for example by using a robust loss function such as absolute error instead of squared error loss (see Chapter 5 for more details). Making a model more robust amounts, to some extent, to making it less flexible. However, robustness amounts to making the model less flexible in a particular way, namely by putting less emphasis on the data points whose predictions are severely wrong.

If the outliers are of interest to the prediction, they are not really an issue but rather a challenge. We have to use a model that is flexible enough to capture the behavior (small bias). This has to be done with care since very flexible models have a high risk of also overfitting to noise. If it turns out that the outliers in a classification problem are indeed interesting and in fact are from an underrepresented class, we are rather facing an imbalanced problem; see Section 4.5.

Missing Data

A common practical issue is that certain values are sporadically missing in the data. Throughout this book so far, the data has always consisted of complete input-output pairs $\{\mathbf{x}_i, y_i\}_{i=1}^n$, and *missing data* refers to the situation where some (or a few) values from either the input $\mathbf{x}_i$ or the output y_i, for some i, are missing. If the output y_i is missing, we can also refer to it as *unlabelled* data. It is a common practice to denote missing data in a computer with `NaN` (not a number), but less obvious codings also exists, such as 0. Reasons for missing data could, for instance, be a malfunctioning sensor or similar issues at data collection time, or that certain values for some reason have been discarded during the data processing.

As for outliers, a sensible first option is to figure out the reason for the missing data. By going back to the data provider, this issue could potentially be fixed and the missing data recovered. If this is not possible, there is no universal solution for how to handle missing data. There is, however, some common practice which can serve as a guideline. First of all, if the output y_i is missing, the data point is useless for supervised machine learning[2] and can be discarded. In the following, we assume that the missing values are only in the input $\mathbf{x}_i$.

The easiest way to handle missing data is to discard the entire data points ('rows in $\mathbf{X}$') where data is missing. That is, if some feature is missing in $\mathbf{x}_i$, the entire input $\mathbf{x}_i$ and its corresponding output value y_i are discarded from the data, and we are left with a smaller dataset. If the dataset that remains after this procedure still contains enough data, this approach can work well. However, if this would lead to too small a dataset, it is of course problematic. More subtle, but also important, is the situation when the data is missing in a systematic fashion, for example that missing data is more common for a certain class. In such a situation, discarding data points with missing data would lead to a mismatch between reality and training data, which may degrade the performance of the learned model further.

If missing data is common, but only for certain features, another easy option is to not use those features ('column of $\mathbf{X}$') which are suffering from missing data. Whether or not this is a fruitful approach depends on the situation.

Instead of discarding the missing data, it is possible to impute (fill in) the missing values using some heuristics. Say, for example, that the jth feature x_j is missing from data point $\mathbf{x}_i$. A simple imputation strategy would be to take the mean or median of x_j for all other data points (where it is not missing) or the mean or median of x_j for all data points of the same class (if it is a classification problem). It is also possible to come up with more complicated imputation strategies, but each imputation strategy implies some assumptions about the problem. Those assumptions might or might not be fulfilled, and it is hard to guarantee that imputation will help the performance in the end. A poor imputation can even degrade the performance compared to just discarding the missing data.

[2]The 'partly labelled data' problem is a semi-supervised problem, which is introduced in Chapter 10 but not covered in depth by this book.

Some methods are actually able to handle missing data to some extent (which we have not discussed in this book), but under rather restrictive assumptions. Such an example is that the data is 'completely missing at random', meaning that which data is missing is completely uncorrelated to what value it, and other features and the output, would have had, had it not been missing. Assumptions like these are very strong and rarely met in practice, and the performance can be severely degraded if those assumptions are not fulfilled.

Feature Selection

When working with a supervised machine learning problem, the question of whether all available input variables/features contribute to the performance is often relevant. Removing the right feature is indeed a type of regularisation which can possibly reduce overfitting and improve the performance, and the data collection might be simplified if a certain variable does not even have to be collected. Selecting between the available features is an important task for the machine learning engineer.

The connection between regularisation and feature selection becomes clear by considering L^1 regularisation. Since the main feature of L^1 regularisation is that the learned parameter vector $\widehat{\boldsymbol{\theta}}$ is sparse, it effectively removes the influence of certain features. If using a model where L^1 regularisation is possible, we can study $\widehat{\boldsymbol{\theta}}$ to see which features we simply can remove from the dataset. However, if we cannot or prefer not to use L^1 regularisation, we can alternatively use a more manual approach to feature selection.

Remember that our overall goal is to obtain a small new data error E_{new}, which we for most methods estimate using cross-validation. We can therefore always use cross-validation to tell whether we gain or lose by including a certain feature in $\mathbf{x}$. Depending on the amount of data, evaluating all possible combinations of removed features might not be a good idea, either due to computational aspects or the risk of overfitting. There are, however, some rules of thumb that can possibly give us some guidance on which features we should investigate more closely for whether they contribute to the performance or not.

To get a feeling for the different features, we can look at the correlation between each feature and the output and thereby get a clue about which features might be more informative about the output. If there is little correlation between a feature and the output, it is possibly a useless feature, and we could investigate further if we can remove it. However, looking one feature at a time can be misleading, and there are cases where this would lead to the wrong conclusion – for example the case in Example 8.1.

Another approach is to explore whether there are redundant features, with the reasoning that having two features that (essentially) contain the same information will lead to an increased variance compared to having only one feature with the same information. Based on this argument, one may look at the pairwise correlation between the features and investigate removing features that have a high correlation to other features. This approach is somewhat related to PCA (Chapter 10).

11.5 Can I Trust my Machine Learning Model?

Supervised machine learning presents a powerful family of all-purpose general black-box methods and has demonstrated impressive performance in many applications. The main argument for supervised machine learning is, frankly, that it works well empirically. However, depending on the requirements of the application, supervised machine learning also has a potential shortcoming, in that it relies on ‘repeating patterns seen in training data’ rather than ‘deduction from a set of carefully written rules’.

Understanding Why a Certain Prediction was Made

In some applications, there might be an interest in ‘understanding’ why a certain prediction was made by a supervised machine learning model, for example in medicine or law. Unfortunately, the underlying design philosophy in machine learning is to deliver good predictions rather than to explain them.

With a simpler model, like the ones in Chapters 2–3, it can to some degree be possible for an engineer to inspect the learned model and explain the ‘reasoning’ behind it for a non-expert. For more complicated models, however, it can be a rather hard task.

There are, however, methods at the forefront of research, and the situation may look different in the future. A related topic is that of so-called adversarial examples, which essentially amounts to finding an input $\mathbf{x}'$ which is as close as possible to $\mathbf{x}$ but gives a different prediction. In the image classification setting, it can, for example, be the problem of having a picture of a car being predicted as a dog by only changing a few pixel values.

Worst Case Guarantees

In the view of this book, a supervised machine learning model is good if it attains a small E_{new}. It is, however, important to remember that E_{new} is a *statistical* claim, under the assumption that the training and/or test data resembles the reality which the model will face once it is put into production. And even if that non-trivial assumption is satisfied, there are no claims about how badly the model will predict in the worst individual cases. This is indeed a shortcoming of supervised machine learning and potentially also a show-stopper for some applications.

Simpler and more interpretable models, like logistic regression and trees, for example, can be inspected manually in order to deduce the ‘worst case’ that could happen. By looking at the leaf nodes in a regression tree, as an example, it is possible to give an interval within which all predictions will lie. With more complicated models, like random forests and deep learning, it is very hard to give any worst case guarantees about how inaccurate the model can be in its predictions when faced with some particular input. However, an extensive testing scheme might reveal some of the potential issues.

11.6 Further Reading

The user aspects of machine learning is a fairly under-explored area, both in academic research publications and in standard textbooks on machine learning. Two exceptions are Ng (2019) and Burkov (2020), from which parts of this chapter have been inspired. Regarding data augmentation, see Shorten and Khoshgoftaar (2019) for a review of different techniques for images.

Some of the research on ‘understanding’ why a certain prediction was made by a machine learning method is summarised by Guidotti et al. (2018).