In practice, learning algorithms are often specified up to one or more hyperparameters. These are special parameters (like $k$ in $k$-NN or the learning rate, the number of epochs, and the batch size in neural networks) whose value must be determined before the training phase can start. Crucially, setting the hyperparameters in the wrong way can lead to underfitting or overfitting.

A learning algorithm with a hyperparameter is not really an algorithm, but rather a family of learning algorithms, where $\Theta$ is the set of all possible hyperparameter values. Fix a learning problem $(D, \ell)$ and let $A_{\theta}(S)$ be the predictor output when $A_{\theta}$ is run on the training set $S$. Let $\ell_D(A_{\theta}(S))$ be the risk of the predictor $A_{\theta}(S)$, and let $\mathbb{E}[\ell_D(A_{\theta})]$ be the expected risk of $A_{\theta}(S)$ where the expectation is with respect to the random draw of the training set $S$ of given fixed size. Intuitively, $\mathbb{E}[\ell_D(A_{\theta})]$ measures the performance of $A_{\theta}$ on a typical training set of that size. Recall that the test set is used to estimate $\ell_D(A_{\theta}(S))$ for a given training set $S$. This is the typical performance of a predictor output by $A_{\theta}$ when the training set is $S$.

**Evaluating a learning algorithm using external cross-validation.** Assume for now the hyperparameter $\theta$ is fixed and focus on the problem of estimating $\mathbb{E}[\ell_D(A)]$. To do so we can use a technique called $K$-fold (external) cross-validation.

Let $S$ be our entire dataset. We partition $S$ in $K$ subsets (also known as *folds*) $D_1, \ldots, D_K$ of size $m/K$ each (assume for simplicity that $K$ divides $m$). The extreme case $K = m$ provides an estimate known as *leave-one-out*. Now let $S^{(k)} \equiv S \setminus D_k$. We call $D_k$ the **testing part** of the $k$-th fold while $S^{(k)}$ is the **training part**.

For example, if we partition $S = \{(x_1, y_1), \ldots, (x_{20}, y_{20})\}$ in $K = 4$ subsets

\[
D_1 = \{(x_1, y_1), \ldots, (x_5, y_5)\} \quad D_2 = \{(x_6, y_6), \ldots, (x_{10}, y_{10})\} \\
D_3 = \{(x_{11}, y_{11}), \ldots, (x_{15}, y_{15})\} \quad D_4 = \{(x_{16}, y_{16}), \ldots, (x_{20}, y_{20})\}
\]

then $S^{(2)} = \{(x_1, y_1), \ldots, (x_5, y_5), (x_{11}, y_{11}), \ldots, (x_{20}, y_{20})\}$.

The **$K$-fold CV estimate** of $\mathbb{E}[\ell_D(A)]$ is then computed as follows: we run $A$ on each training part $S^{(k)}$ of the folds $k = 1, \ldots, K$ and obtain the predictors $h_1 = A(S^{(1)})$, $\ldots$, $h_K = A(S^{(K)})$. We then compute the (scaled) errors on the testing part of each fold,

\[
\hat{\ell}_{D_k}(h_k) = \frac{K}{m} \sum_{(x,y) \in D_k} \ell(y, h_k(x)) .
\]

Finally, we compute the CV estimate by averaging these errors

\[
\frac{1}{K} \sum_{k=1}^{K} \hat{\ell}_{D_k}(h_k) .
\]
Tuning hyperparameters on given a training/test split. Next, we look at the problem of estimating
\[ \min_{\theta \in \Theta} \ell_D(h^{(\theta)}_S) \quad \text{where} \quad h^{(\theta)}_S = A_\theta(S) . \] (1)

This is the risk of the predictor resulting from running the algorithm with an optimally chosen hyperparameter on a given training set. As \( \Theta \) may be very large, possibly infinite, in practice the minimization is not over \( \Theta \), but over a suitably chosen subset of it (for example, if \( \Theta = [0, 1] \), then the subset could by a finite grid of equally spaced values in \( [0, 1] \)).

The estimate is computed by splitting the training data in two subsets \( S_{\text{train}} \) and \( S_{\text{dev}} \). The development set \( S_{\text{dev}} \) (also called validation set) is used as a surrogate test set. The algorithm is run on \( S_{\text{train}} \) once for each value of the hyperparameter in the chosen subset of \( \Theta \). The resulting predictors are tested on the dev set. In order to obtain the final predictor, the learning algorithm is run once more on the original training set \( S \) using the value of the hyperparameter corresponding to the predictor with smallest error on the validation set. The resulting predictor is finally evaluated on the test set to estimate the risk.

Tuning parameters via nested cross-validation. We now turn to the problem of estimating
\[ \mathbb{E}\left[ \min_{\theta \in \Theta} \ell_D(A_\theta) \right] \] (2)

This is the risk, averaged over all training sets \( S \) of given fixed size, of
\[ \min_{\theta \in \Theta} \ell_D(A_\theta(S)) \]
which is exactly what we estimate in (1) for a fixed training set \( S \).

A cheap way of estimating (2) is to compute an external cross-validation estimate for each algorithm in \( \{A_\theta : \theta \in \Theta\} \), and then using the lowest of such estimates for estimating (2). This tends to underestimate (2). In practice, however, this difference is typically small.

A better, though more computationally intensive estimate of (2) is computed through nested cross-validation. This works as follows: First, an external cross-validation is run on the given dataset. On each fold of this external CV, we run an “internal” cross-validation for each algorithm in \( \{A_\theta : \theta \in \Theta\} \). The \( A_\theta \) with best internal cross-validation estimate is retrained on the entire training part of the fold; the resulting predictor is then evaluated on the testing part of the fold. The value of the external cross-validation estimate is finally used as nested cross validation estimate for (2).

Note that in each run of internal cross-validation we optimize the hyperparameter \( \theta \) locally, on the training part of the external cross-validation fold. Hence, the nested cross-validation estimate is computed by averaging the performance of predictors obtained with potentially different values of their hyperparameter.