Machine Learning

Toghrul Karimov
Max Planck Institute for Software Systems

Last updated in 08.2025

Contents

1 Some basics

The most basic goal of machine learning is to learn useful information about a population from a finite sample of points that are drawn from it. Once we have a model, we can then use it to estimate various things about a fresh, previously unseen sample drawn from the same population. Let us look at a simple example.

Example 1.1. We sample N people from the population, and record their gender and height. We then model the distribution of height in both groups using normal distribution. Given that a person is male, we can then predict their height as the mean of the corresponding probability distribution. Given that a person has height x, we can predict their gender by determining the group in which x is a more likely/less extreme observation.

For the sake of an argument, suppose the true distributions of the height of males and females follow are $\mathcal{N}(178, 7.7^2)$ and $\mathcal{N}(163, 7.3^2)$, respectively. Moreover, 51% of the population is female and 49% is male. (These two assumptions are derived from the US population[1].) Consider the problem of predicting gender given height. Regardless of whether we have access to the true underlying distribution or only to a "reasonable" sample, the most natural algorithm is to determine a threshold t, and given height x, guess "male" if $x \geq t$ and "female" otherwise. In probabilistic terms, let X denote the height of a random person and Y their gender; these are random variables. We have that $\mathrm{P}(Y = \text{male}) = 0.49$ and $\mathrm{P}(Y = \text{female}) = 0.51$.

Suppose we choose the threshold $t = 175$. What is the accuracy of our model? That is, when we sample people randomly, what percentage of the time will our guess be correct? We have that

$$\begin{aligned}\mathrm{P}(\text{correct guess}) &= \mathrm{P}(Y = \text{male and } X \geq t) + \mathrm{P}(Y = \text{female and } X < t) \\ &= \mathrm{P}(X \geq t \mid Y = \text{male})\,\mathrm{P}(Y = \text{male}) + \mathrm{P}(X < t \mid Y = \text{female})\,\mathrm{P}(Y = \text{female}).\end{aligned}$$

[1] https://allendowney.blogspot.com/2012/01/some-of-my-best-friends-are-crackpots.html

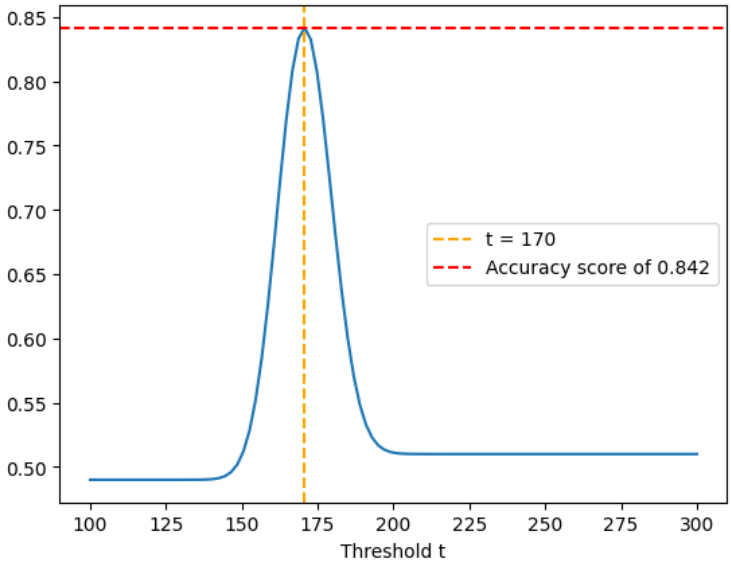


Figure 1: Accuracy of the simple threshold algorithm (over the full population) for predicting gender given height as a function of the threshold t

Using the cumulative distribution functions of $\mathcal{N}(178, 7.7^2)$ and $\mathcal{N}(163, 7.3^2)$, we conclude that the probability of our guess being correct is approximately 0.803. It can be shown that the best possible value is $t \approx 170$, which gives accuracy of approximately 0.842; see Figure 1. That is, any model/algorithm for predicting gender from height will be wrong at least 15.8 percent of the time when presented with a large amount of unseen data.

The previous example illustrates some of the most basic issues in machine learning. Firstly, how should we choose the value of the parameter t? How can we evaluate if our choice is good, and estimate its accuracy on the full population? Which family of models should we consider? Maybe instead of partitioning $\mathbb{R}$ into the two intervals $(-\infty, t), [t, \infty)$, we should compare the sample height against multiple different thresholds? We will answer these questions in the following lectures. For now, we mention that the perspective we took in in this lecture is called *statistical machine learning*: it attempts to model uncertainty using probability theory, and was the dominant approach to learning in the 20th century. In the 21th century, the dominant approach is to train a highly powerful model on a very large dataset; if it works, it works, but usually there is no strong mathematical justification for why it does.

2 Maximum likelihood estimate

Suppose we want to fit a model to our data (e.g. a normal distribution), but don't know which parameters we should choose (e.g. the mean and standard deviation). The paradigm of *maximum likelihood estimate* (MLE) states that we should choose the unknown parameters so that under our choice, the *likelihood* of observing the data we have is maximised. For discrete random variables, likelihood is the same as probability; let us consider the following example.

2.1 Bernoulli distribution

Example 2.1. Suppose we have a biased coin that comes up with heads with probability $0 < p < 1$. We toss it 10 times and observe 6 heads and 4 tails. (Of course, the argument applies to any number of observed heads.) Which hypothesis regarding the value of p should we choose?

Recall that the number of heads observed follows the binomial distribution with parameters $n = 10$ and p. Following the MLE paradigm, we write the probability of observing our data for each value of p as $\ell(p) = \binom{10}{6} p^6 (1-p)^4$. We want to find p that maximises this quantity. To this end, we take

the derivative with respect to p to obtain

$$\ell'(p) = \binom{10}{6}(6p^5(1-p)^4 - 4(1-p)^3p^6) = \binom{10}{6}p^5(1-p)^3(6-10p).$$

Setting the derivative to zero, we obtain that g has a single critical point at $p = \frac{6}{10}$, which is the intuitive value for p that anyone would predict given our observation of 6 heads and 4 tails. To formally argue that $p = \frac{6}{10}$ is a global maximum of g we need to (1) take the second derivative and show that its value is negative at the critical point, and (2) show that $\lim_{p\to 0}\ell(p)$ and $\lim_{p\to 1} g(p)$ are not greater than the value at the critical point; in our case both limits are in fact zero.

2.2 Univariate normal distribution

Next, let us consider the case of a univariate normal distribution. Suppose we are given observations $x_1, \ldots, x_m \in \mathbb{R}$, and want to find the normal distribution that best fits our data. For continuous random variables, comparing the probabilities of observing m particular points does not make sense: such probabilities are always zero. Instead, we use the *likelihood*. For $x \in \mathbb{R}$, the likelihood of observing x in the model given by μ, σ^2 is defined as

$$\mathcal{L}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$$

where the right-hand side is the probability density function of the normal distribution with mean μ and variance σ^2 evaluated at the point x. Observe that the likelihood is always positive but should not be interpreted as a probability: in particular, it can be greater than 1. The likelihood of observing $x_1, \ldots, x_m$ is then simply defined as

$$\mathcal{L}(x_1, \ldots, x_m; \mu, \sigma^2) = \prod_{i=1}^{m} \mathcal{L}(x_i; \mu, \sigma^2) = \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^m e^{-\frac{1}{2\sigma^2}\sum_{i=1}^m (x_i-\mu)^2}.$$

This definition is justified by the fact that $x_1, \ldots, x_m$ are assumed to be independent: recall that $P(A \text{ and } B) = P(A)P(B)$ for independent events A, B. Hence to perform an MLE estimate on our data $x_1, \ldots, x_m$ we have to solve the following optimisation problem. Find $\mu \in \mathbb{R}$ and $\sigma^2 > 0$ that maximises the value of $\mathcal{L}(x_1, \ldots, x_m; \mu, \sigma^2)$. We next show how to do this. We additionally assume that it is not the case that $x_1 = \cdots = x_m$, because otherwise it is not possible to conclude anything from the data about the variance. Mathematically, without this assumption we get that $\mathcal{L}(x_1, \ldots, x_m; \mu, \sigma^2)$ does not have global minimum because we can choose $\mu = x_1 = \cdots = x_m$ and make $\mathcal{L}(x_1, \ldots, x_m; \mu, \sigma^2)$ arbitrarily small by taking $\sigma^2 \to 0$.

Because the natural logarithm $\log\colon (0, \infty) \to \mathbb{R}$ is a strictly increasing function (i.e. $\log(x) < \log(y)$ if and only if $x < y$), our problem is equivalent to maximising the log-likelihood

$$\begin{aligned}
\ell(x_1, \ldots, x_m; \mu, \sigma^2) &= \log(\mathcal{L}(x_1, \ldots, x_m; \mu, \sigma^2)) \\
&= m \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \frac{1}{2\sigma^2}\sum_{i=1}^m (x_i - \mu)^2 \\
&= -\frac{m}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^m (x_i - \mu)^2
\end{aligned}$$

which we simply denote by ℓ. Note that in the expression above, every occurrence of σ is squared. Hence we can think in terms of variance and not standard deviation. We now apply calculus: every local maximum must satisfy

$$\begin{aligned}
\frac{\partial \ell}{\partial \mu} &= -\frac{1}{\sigma^2}\sum_{i=1}^m (x_i - \mu) = 0 \\
\frac{\partial \ell}{\partial \sigma^2} &= -\frac{m}{2\sigma^2} + \frac{1}{2\sigma^4}\sum_{i=1}^m (x_i - \mu)^2 = 0
\end{aligned}$$

where we have applied the Chain Rule for differentiation. From the first equation,

$$-\frac{1}{\sigma^2}\sum_{i=1}^m (x_i - \mu) = 0 \;\Leftrightarrow\; \sum_{i=1}^m (x_i - \mu) = 0 \;\Leftrightarrow\; \sum_{i=1}^m x_i = m\mu \;\Leftrightarrow\; \mu = \frac{1}{m}\sum_{i=1}^m x_i.$$

That is, at all critical points of ℓ (which as a function of μ and σ^2) the value of μ is the *sample mean*, which is natural and intuitive. On the other hand,

$$-\frac{m}{2\sigma^2} + \frac{1}{2\sigma^4}\sum_{i=1}^{m}(x_i-\mu)^2 = 0 \Leftrightarrow \sum_{i=1}^{m}(x_i-\mu)^2 = m\sigma^2 \Leftrightarrow \sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu)^2.$$

That is, at every critical point σ^2 is equal to the *sample variance*. This is still intuitive, but not as unequivocal as the critical point we obtained for μ: a different but well-known estimate of the population variance σ^2 given $x_1, \ldots, x_m$ is the *unbiased sample variance* $\frac{1}{m-1}\sum_{i=1}^{m}(x_i-\mu)^2$, which is often preferred in practice.

So far we have shown that ℓ has a single critical point given by

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{1}$$

and

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu)^2. \tag{2}$$

It remains to determine the nature of this point. To this end, we use the *second partial derivative test*. We have that

$$\begin{aligned}
\frac{\partial^2 \ell}{\partial\mu\,\partial\sigma^2} &= \frac{\partial^2 \ell}{\partial\mu\,\partial\sigma^2} = \frac{1}{\sigma^4}\sum_{i=1}^{m}(x_i-\mu) \\
\frac{\partial^2 \ell}{\partial\mu^2} &= \frac{m}{\sigma^2} \\
\frac{\partial^2 \ell}{\partial(\sigma^2)^2} &= \frac{m}{2\sigma^4} - \frac{1}{\sigma^6}\sum_{i=1}^{m}(x_i-\mu)^2
\end{aligned}$$

where we have again used the Chain Rule. At our critical point, the value of $\frac{\partial^2\ell}{\partial\mu^2}$ is positive, and the quantity

$$D(\mu,\sigma^2) = \frac{\partial^2\ell}{\partial\mu^2}\cdot\frac{\partial^2\ell}{\partial(\sigma^2)^2} - \frac{\partial^2\ell}{\partial\mu\,\partial\sigma^2}\cdot\frac{\partial^2\ell}{\partial\sigma^2\,\partial\mu}$$

is equal to $\frac{m^2}{2\sigma^6}$, which is positive. By the second partial derivative test, we conclude that (μ,σ^2) defined by the equations (1) and (2) is a local maximum. Finally, observe that for any fixed μ, we have that $\ell \to -\infty$ as $\sigma^2 \to \infty$. Hence our unique critical point is the (unique) global maximum.

2.3 Maximum a posteriori estimate

Suppose we have fit a probabilistic model to our data, e.g. using MLE. More concretely, suppose we have got the random variables $X_1, \ldots, X_k, Y$ (where we want to predict Y given $X_1, \ldots, X_k$), and have determined the joint probability distribution function $f_y(x_1, \ldots, x_k)$ for every possible value of y; in case of Example 1.1, we have $k = 1$, X_1 is the height, and f_{male} and f_{female} are both normal distributions. Given $X_1 = x_1, \ldots, X_k = x_k$, we can simply predict the value of Y to be y such that the likelihood of observing $X_1 = x_1, \ldots, X_k = x_k$ assuming $Y = y$, i.e. $f_y(x_1, \ldots, x_k)$, is maximised. But what if some values of y are very rare? To maximise accuracy, we want to somehow predict such y less frequently that other, more common values of Y. The solution is the maximum a posterior estimate (MAP), which is derived from Bayes' theorem. We have that

$$\mathrm{P}(Y = y \mid X_1 = x_1, \ldots, X_k = x_k) = \frac{\sum_y f_y(x_1, \ldots, x_k)\cdot \mathrm{P}(Y = y)}{f(x_1, \ldots, x_k)}.$$

Here $f(x_1, \ldots, x_k)$ is the joint distribution function of $X_1, \ldots, X_k$, but note that it does not depend on y. Hence we choose y to maximise $\sum_y f_y(x_1, \ldots, x_k)\cdot \mathrm{P}(Y = y)$. Note that the summation is over all values Y can take, and we need the value of $\mathrm{P}(Y = y)$: the latter can be estimated as the proportion of points with $Y = y$ in our dataset.

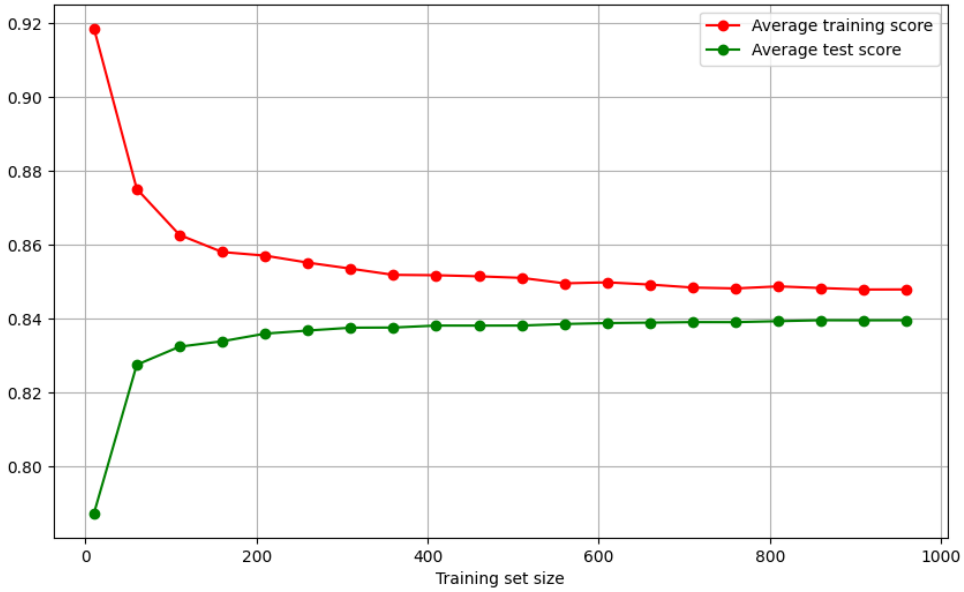


Figure 2: Learning curves for the threshold classifier predicting gender given height

3 Choosing parameters by minimising a loss function

Let us go back to Example 1.1, where we want to calculate the best possible threshold t given observations $x_1, \ldots, x_m$ (heights) and $y_1, \ldots, y_m$ (gender) We can do this by defining a *loss function* ℓ of t, and then find t that minimises it. For example, we can define ℓ to calculate the proportion of the points among $x_1, \ldots, x_m$ that would be incorrectly classified:

$$\ell(t) = \frac{1}{m}(\#\{i \mid y_i = \text{female and } x_i \geq t\} + \#\{i \mid y_i = \text{female and } x_i \geq t\}).$$

Alternatively, we can minimise the following loss function:

$$\ell(t) = \frac{1}{m}\left(\sum_{y_i=\text{female}} \max(0, x_i - t) + \sum_{y_i=\text{male}} \max(0, t - x_i).\right)$$

This is known as the *hinge loss*. Intuitively, we do not just count the number of misclassified points, but impose a penalty (distance to t) for the incorrectly classified points that becomes larger the more "badly misclassified" the point is. The *accuracy* or the *score* associate with t is then $1 -$ loss. Figure 2 illustrates what happens if we try to find the optimal value of t where the number of samples m becomes larger and larger. For each sample size m, we first randomly generate $m/2$ males and $m/2$ females, with heights distributed according to the normal distributions $\mathcal{N}(178, 7.7^2)$ and $\mathcal{N}(163, 7.3^2)$, respectively. We then find t that minimises the first loss function above. (The situation for the hinge loss is similar.) For the chosen t, we calculate the accuracy on the training data (i.e. the *train score*; again computed as one minus the loss), as well as the probability that a randomly samples person will be classified correctly (i.e. the *test score*), which can be computed exactly as shown in Section 1. We then repeat this experiment many $N = 10000$ times, and compute the average the train (red dots) and test scores (green dots).

We see that as the sample size goes to infinity, the test score of the classifier we train approaches 0.842, which is the theoretically best value; see again Section 1. The curves depicted in Figure 2 are culled the *learning curves*; if we are truly learning something generalisable from our data, then these curves should converge as we increase the amount of available training data.

4 Overfitting and cross validation

Overfitting means that our model is learning noise specific to the training data and performs poorly on unseen data. We will see that it is very easy to overfit; it is much more difficult to learn relations between the variables that generalise to unseen data. Our running example in this section is the following.

Example 4.1. Suppose we have two features, Feature 1 (denoted Z_1) and Feature 2 (denoted Z_2), and two classes red and blue. We have 50 samples from each of the two classes; each sample is a two-dimensional vector. The 100 samples are shown in Figure 3. We want to build a classifier that takes an unseen observation of Z_1, Z_2, and guesses whether the point is red or blue.

In fact, for red points, Z_1, Z_2 are drawn independently from the normal distribution with variance 1 and means $1, 0$, respectively. For blue points, on the other hand, Z_1, Z_2 are drawn independently from the normal distribution with variance 1 and means $-1, 0$, respectively. Note that usually we do not have access to this information when building our classifier. Using probability theory it can be shown that the best possible classifier for our problem is simply the line $x = 0$, which has accuracy (over unseen data) of approximately 0.84. That is, any possible classifier will misclassify roughly 1 in 6 unseen points when tested on a large amount of data.

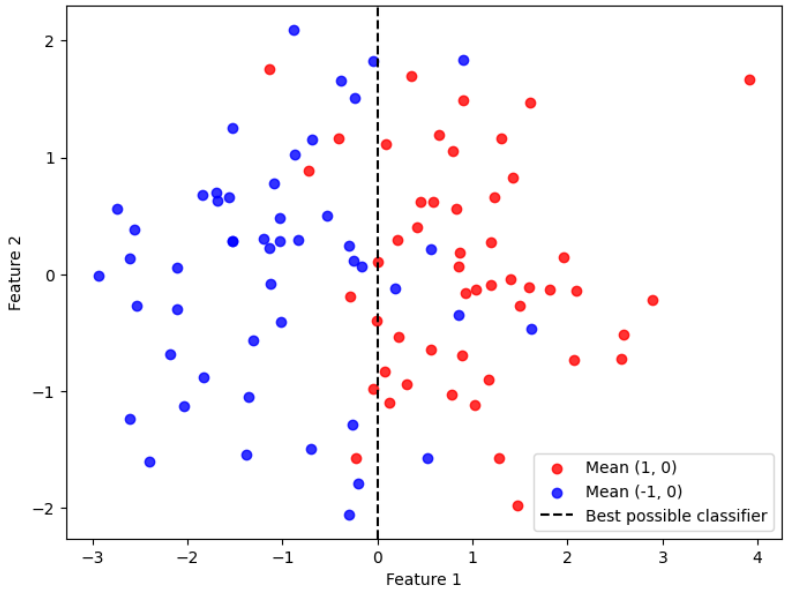
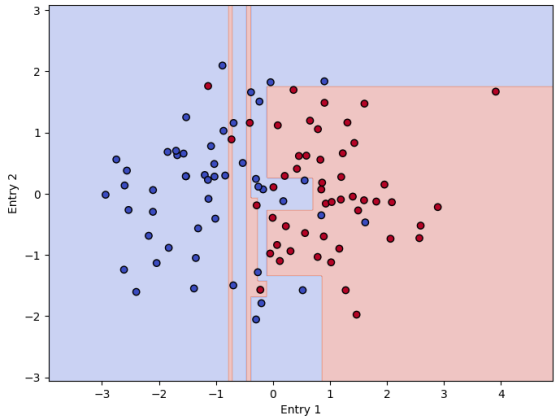


Figure 3: Samples of (Z_1, Z_2), 50 for each class.

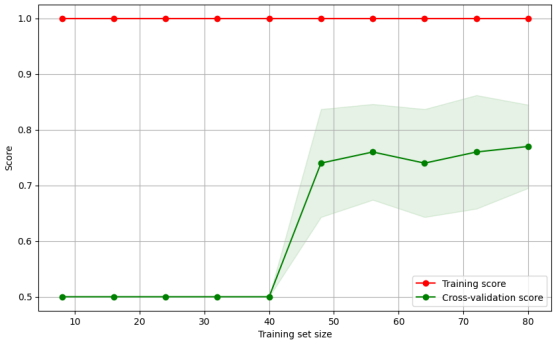
Suppose we have decided that we will build a model that draws many vertical and horizontal lines in the two-dimensional grid, and colours each resulting rectangle either blue or red; the prediction is then made based on the square into which the unseen point falls. That is, we will build a *decision tree*. We can choose the tree that minimises some loss function (e.g. the number of misclassified points). How can we then estimate the test error of our final model, when we do not have access to any new unseen points?

The first approach is to simply split our original data into two parts: the training data and the test data. The ratio of the split is often chosen to be 70/30, 80/20, or 90/10. We choose the parameters of the model that minimise the error with respect to the training set, and estimate its error on the test set. This gives us an idea about how the model will perform on unseen data. We can then merge the two sets of data and retrain the model on the full data.

But it is possible the particular split into train+test data that we get will contain lots of abnormal points. To alleviate this issue, we can repeat the splitting procedure many times, and then average the error on the test and training sets across all splits. A computationally efficient way to do this is *k-fold cross validation*. We randomly partition our data into k sets of equal size. At the ith step, where $1 \leq i \leq k$, we designate one of the ith set as the test set. We train our model on the union of the other $k-1$ sets, choosing the parameters that minimise the training error (i.e. maximise the training score), and evaluate it on the ith set to compute the *validation score*. After the k iterations,



(a) Decision boundaries of a model that overfits the training data



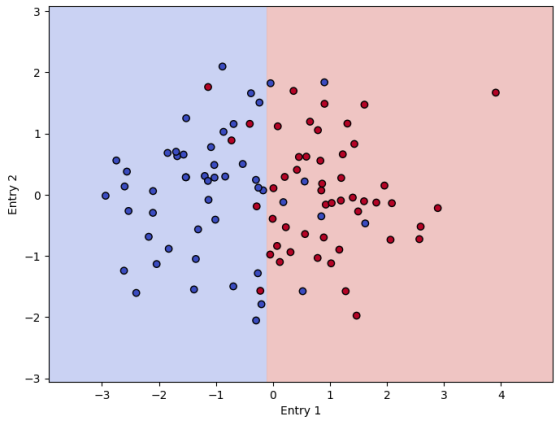
(b) The learning curves for the model that is allowed to partition $\mathbb{R}^2$ using arbitrarily many vertical and horizontal lines. The shaded area represents ± 1 standard deviation computed from the list of 10 validation scores.

Figure 4

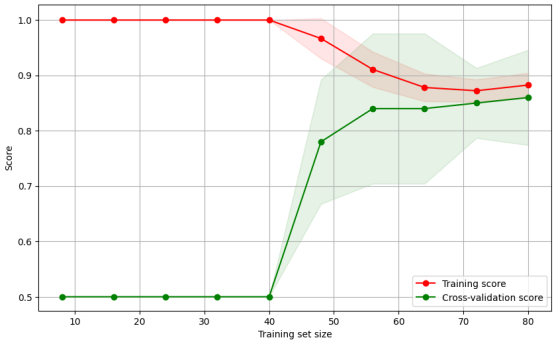
we have two lists of length k: the training scores and the validation score. At the end, we average the k validation scores to compute the *cross validation score*, and retrain the model on the full data. The value of k is usually chosen to be 5 or 10, and in the extreme case can be equal to the number of data points we have; this is known as leave-one-out cross validation.

Let us go back to our problem of partitioning $\mathbb{R}^2$ into a red set and a blue set (Example 4.1). Figure 4a illustrates a possible model that almost perfectly classifies all the data we have. Is such a model a reasonable choice? We see that the model has learned some noise, e.g. the two nonsensical narrow pink strips. But how can we detect such overfitting when we do not have access to the true underlying distribution? Figure 4b gives an answer. We progressively use more and more of our 100 data points, at each step performing 10-fold cross validation. The role of the cross-validation score is analogous to the role of the test score when we do know the underlying distribution/are able to sample as many new points as we like. Comparing Figure 4b, we see that our classifier is not learning anything new as give to it more and more data. Moreover, the two curves do not converge. We conclude that our approach is not sound: our models are just memorising noise.

We therefore change our approach. Now we will search for the best partition constructed from a single horizontal or vertical line. That is, we will learn a decision tree that can ask only a single question: on which side of the separating line is the given data point?



(a) A constrained model that fits our training data



(b) Learning curves for the constrained family of models

Figure 5

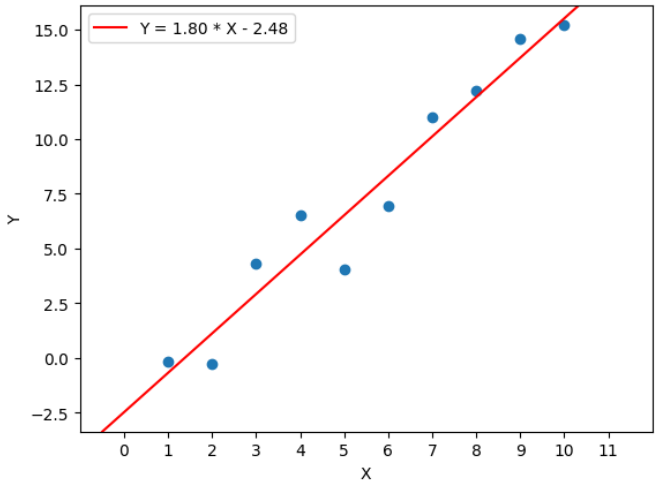


Figure 6: Sample data with $m = 10$ and the best-fitting line

Figure 5a depicts the partition that bets fits our full data of 100 points. We see that it is reasonably close to the best possible separation line $x = 0$. But how well should we expect it to perform on unseen data? Figure 5b shows the learning curves obtained by performing 10-fold cross validation on increasingly large subsets of our data. We see that the training score and the cross validation score converge, which is what should happen in a reasonable model. In short, we alleviated the problem of overfitting by restricting the degrees of freedom that our models have. To conclude our discussion on overfitting, we give a quote from John von Neumann: " With four parameters I can fit an elephant, and with five I can make him wiggle his trunk." When the model has many degrees of freedom, it will use it fully to learn the training data as well as possible to minimise the loss function.

5 Linear regression

5.1 The univariate setting

Let us start with the one-dimensional case. Suppose we have two real-valued quantities, x and y, and we want to model the dependence between them as $y = ax + b$ for some $a, b \in \mathbb{R}$. Further suppose that we have m observations $(x_1, y_1), \ldots, (x_m, y_m)$. Figure 6 displays a sample with $m = 10$ points, as well as the line defined by $a = 1.80$ and $b = -2.48$. In the least squares approach, we try to find a, b that minimise the *mean squared error*, given by

$$E(a, b) = \frac{1}{m} \sum_{i=1}^{m} (y_i - (ax_i + b_i))^2.$$

Remark 1. Similarly, *root mean squared error*, *negative mean squared error*, and *negative root mean squared error* are defined as $-E(a, b)$, $\sqrt{E(a, b)}$, and $-\sqrt{E(a, b)}$, respectively. For MSE and root MSE, a lower value is better. For negative MSE and negative root MSE, a higher value is better, which matches how we try to maximise the train/test score (i.e. accuracy level) in the context of classification.

To minimise the MSE, we simply take partial derivatives with respect to a and b to obtain

$$\frac{\partial E}{\partial a} = -\frac{2}{m} \sum_{i=1}^{m} x_i (y_i - (ax_i + b_i))$$
$$\frac{\partial E}{\partial b} = -\frac{2}{m} \sum_{i=1}^{m} (y_i - (ax_i + b_i)).$$

Writing μ_x, μ_y for the sample means of x and y, respectively, and setting both derivatives to zero, we

obtain that at the unique critical point,

$$a = \frac{\frac{1}{m-1}\sum_{i=1}^{m}(x_i - \mu_x)(y_i - \mu_y)}{\frac{1}{m-1}\sum_{i=1}^{m}(x_i - \mu_x)^2}$$

which is the *sample covariance* divided by *sample variance of* X. On the other hand, we have

$$b = \mu_y - a\mu_x$$

which ensures that $\mu_y = a\mu_x + b$. That is, the least squares regression line always passes through the point whose coordinates are the sample means.

Instead of minimising MSE, we could also minimise the *mean absolute error* (MAE), which is given by $\frac{1}{m}\sum_{i=1}^{m}|y_i - (ax_i + b)|$. The MAE has the disadvantage that it is not everywhere differentiable. The choice between MSE and MAE is also motivated by how we want to punish very bad predictions: is being 10 units off twice as bad being 5 units off, or is it much worse?

5.2 The multivariate case

Now suppose $x = \mathbb{R}^k$ and $y \in \mathbb{R}$. That is, we have k-dimensional input, and we want to predict a single real value; this is the setting of *multiple linear regression*. The linear model in this case is $y = a^\top \cdot x + b$, where $\mathbf{a}$ is a vector with k entries and b is a real number. Again suppose we have m observations $(x_1, y_1), \ldots, (x_m, y_m) \in \mathbb{R}^k \times \mathbb{R}$. Construct the *data matrix* $X \in \mathbb{R}^{m\times(k+1)}$: the ith row consists of 1 followed by the entries of x_i. Similarly, let $Y \in \mathbb{R}^m$ be the vector whose ith entry is y_i. We further write $w = (b; a) \in \mathbb{R}^{k+1}$. Then our problem is to find w that makes Xw as close to Y as possible. Formally, we want to find w that minimises $\frac{1}{m}\|Xw - Y\|_2^2$ (where $\|\cdot\|_2$ denotes the Euclidean distance in $\mathbb{R}^k$), which is again the MSE. This value of w satisfies

$$X^\top Xw = X^\top y.$$

Assuming the columns of X are linearly independent, then we can compute

$$w = (X^\top X)^{-1}X^\top y.$$

5.3 Ridge and lasso regression

When k, the number features, is large, there will be many possible values of w that fit our data (X, Y) well. Which one should we choose? Additionally, the larger the value of k, there more opportunities for overfitting there are. How should avoid overfitting? The answer is that, in addition to the (mean squared) error in predicting y, we will impose a penalty on "using" w. Write $w = (w_1, \ldots, w_k)$. Then

- in ridge regression, we find w that minimises $\frac{1}{m}\|Xw - Y\|_2^2 + \alpha\|w\|_2^2$, and
- in lasso regression, we minimise $\frac{1}{m}\|Xw - Y\|_2^2 + \alpha\|w\|_1$.

In both cases, $\alpha > 0$ is a *hyper-parameter* (also known as a *meta-parameter*) that needs to be learned, either using cross-validation or a separate *validation set*, before we can learn w. Let us look at an example how to do this.

We work with the red wine quality dataset (`https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv`). The data contains 1599 samples and 12 measurements per sample: fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, alcohol, and quality. We will model quality in terms of the other 11 measurements using multiple linear regression.

We first split the data into a train set (1279 samples) and a test set (320 samples). We compute $w \in \mathbb{R}^{12}$ using the train set, and evaluate it on the test set to obtain the negative RMSE of -0.62. The coefficients of w (recall that the data matrix has an extra column, and the first entry of w is b, the intercept) are

$$14.355, 0.023, -1.001, -0.141, 0.007, -1.807, 0.006, -0.004, -10.352, -0.394, 0.841, 0.282.$$

Now let us try ridge regression. To determine a good value for α, we perform 5-fold cross validation on the training set, to obtain $\alpha = 0.5$. Finally, we evaluate the trained model the test set, to obtain

the negative RMSE of -0.63, which is almost identical to that of the naive linear regression. However, the coefficients of w are

$$4.041, 0.017, -1.021, -0.159, 0.001, -1.464, 0.006, -0.004, -0.019, -0.405, 0.781, 0.295$$

which are smaller than those of the naive linear regression. The coefficients that are zero indicate that the particular column can be dropped. That is, we can construct a simpler model with a simpler performance. This is known as *feature selection*, and is a way to constrain the model to only relevant information, thus mitigating the risk of overfitting.

The feature selection effect is more extreme with the lasso regression: it has tendency to set as many coefficients to zero as possible. In this case, we get $\alpha = 0.01$, negative RMSE of -0.65 on the test set, and coefficient vector w that is

$$2.407, 0.037, -0.786, 0.000, -0.001, -0.000, 0.006, -0.003, -0.000, -0.000, 0.346, 0.303.$$

It seems like only 4 of the 11 measurements (fixed acidity, volatile acidity, sulphates, alcohol) are relevant for predicting the wine quality.

6 Writing your custom classifier in sklearn

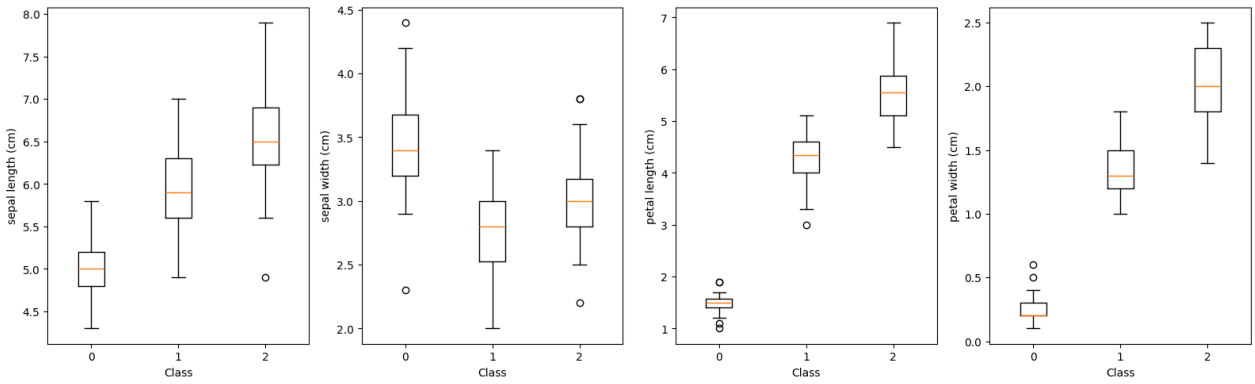
See the practical below.

Predicting iris species by looking at a single attribute

```
import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import learning_curve

from scipy import stats
```

We first plot the distribution of each feature across the three species

```
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names

# Plot box plots for each feature
fig, axes = plt.subplots(nrows=1, ncols=4, figsize=(16, 5))

for i, col in enumerate(feature_names):
    data = X[:, i]
    data1 = data[y == 0]
    data2 = data[y == 1]
    data3 = data[y == 2]
    axes[i].boxplot([data1, data2, data3], tick_labels=[0,1,2])
    axes[i].set_xlabel('Class')
    axes[i].set_ylabel(col)

plt.tight_layout()
plt.show()
```



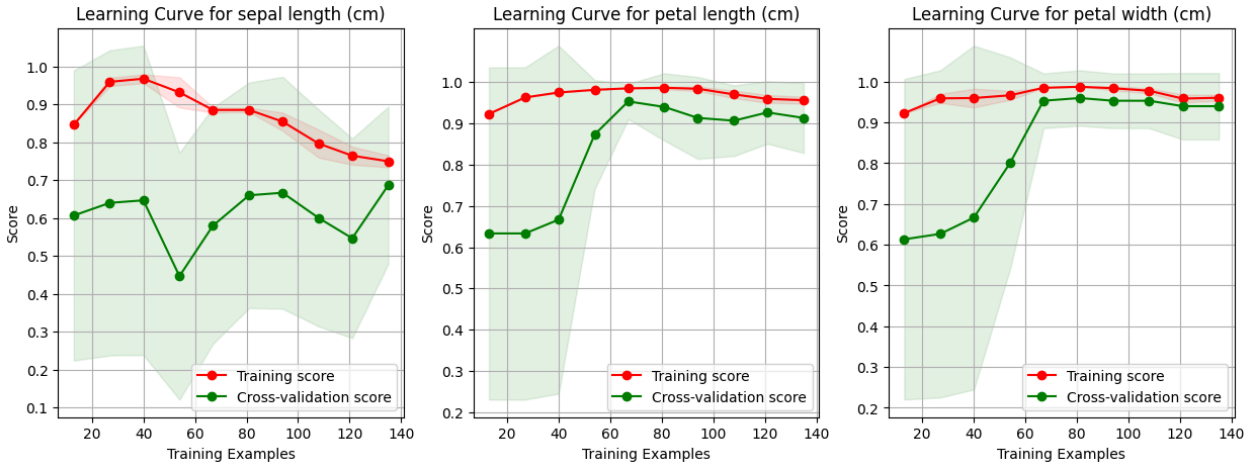
We see that petal length and petal width (and, to a lesser extent, sepal length) across the three classes are well-separated. We next write a classifier implementing the `Estimator` interface of `sklearn` that learns to partition the real line into three intervals and guesses the label of an unseen point by checking the interval into which the point falls.

```
from sklearn.base import BaseEstimator
import numpy as np

class SimpleShrub(BaseEstimator):

    #labels are the three classes to be learned, in the correct order
    #with respect to the partition of the real line that we want
    def __init__(self, labels=[]):
        assert len(labels) == 3

        self.labels = labels
        self.t_1 = None
        self.t_2 = None

    #X should be a sequence and
    #y should contain (only) terms from labels
    def fit(self, X, y=None):
        #check that X is a one-dimensional array
        assert len(np.shape(X)) == 1
        errors = {}

        for c_1 in X:
          for c_2 in X:
            if c_1 > c_2:
              continue
            else:
              error_0 = np.shape(X[(X < c_1) & (y != 0)])[0]
              error_1 = np.shape(X[(X >= c_1) & (X < c_2) & (y != 1)])[0]
              error_2 = np.shape(X[(X >= c_2) & (y != 2)])[0]
              error = error_0 + error_1 + error_2
              errors[(c_1,c_2)] = error

        self.t_1, self.t_2 = min(errors, key=errors.get)
        self.is_fitted_ = True

        return self

    def predict(self, X):
        #check that X is a one-dimensional array
        assert len(np.shape(X)) == 1

        return self.predict_one(self, X)

    def score(self, X, y):
        #check that X is a one-dimensional array
        assert len(np.shape(X)) == 1

        return np.mean(self.predict(X) == y)

    #predicts labels[0] if < t_1, labels[1] if in [t_1,t_2), and
    #labels[2] if >= t_2
    @np.vectorize
    def predict_one(self, x):
        if x < self.t_1:
          return self.labels[0]
        elif x < self.t_2:
          return self.labels[1]
        else:
          return self.labels[2]
```

We will next train 3 classifiers, based on sepal length, petal length, and petal width. For each classifier we will use cross validation with k = 10 on increasingly larger

training data. We will then plot the learning curves.

```
selected_features = [0, 2, 3]

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(15, 5))

for i, f in enumerate(selected_features):

  clf = SimpleShrub(labels=[0, 1, 2])
  feature_data = X[:, f]
  train_sizes, train_scores, test_scores = learning_curve(clf, feature_data, y, cv=10,
  train_scores_mean = np.mean(train_scores, axis=1)
  train_scores_std = np.std(train_scores, axis=1)
  test_scores_mean = np.mean(test_scores, axis=1)
  test_scores_std = np.std(test_scores, axis=1)

  axes[i].set_title("Learning Curve for " + feature_names[f])
  axes[i].set_xlabel("Training Examples")
  axes[i].set_ylabel("Score")
  axes[i].set_yticks(np.linspace(0.1,1,10))
  axes[i].fill_between(train_sizes, train_scores_mean - train_scores_std,
                train_scores_mean + train_scores_std, alpha=0.1,
                color="r")
  axes[i].fill_between(train_sizes, test_scores_mean - test_scores_std,
                test_scores_mean + test_scores_std, alpha=0.1, color="g")
  axes[i].plot(train_sizes, train_scores_mean, 'o-', color="r",
         label="Training score")
  axes[i].plot(train_sizes, test_scores_mean, 'o-', color="g",
         label="Cross-validation score")

  axes[i].legend(loc="best")
  axes[i].grid()

fig.show()
```



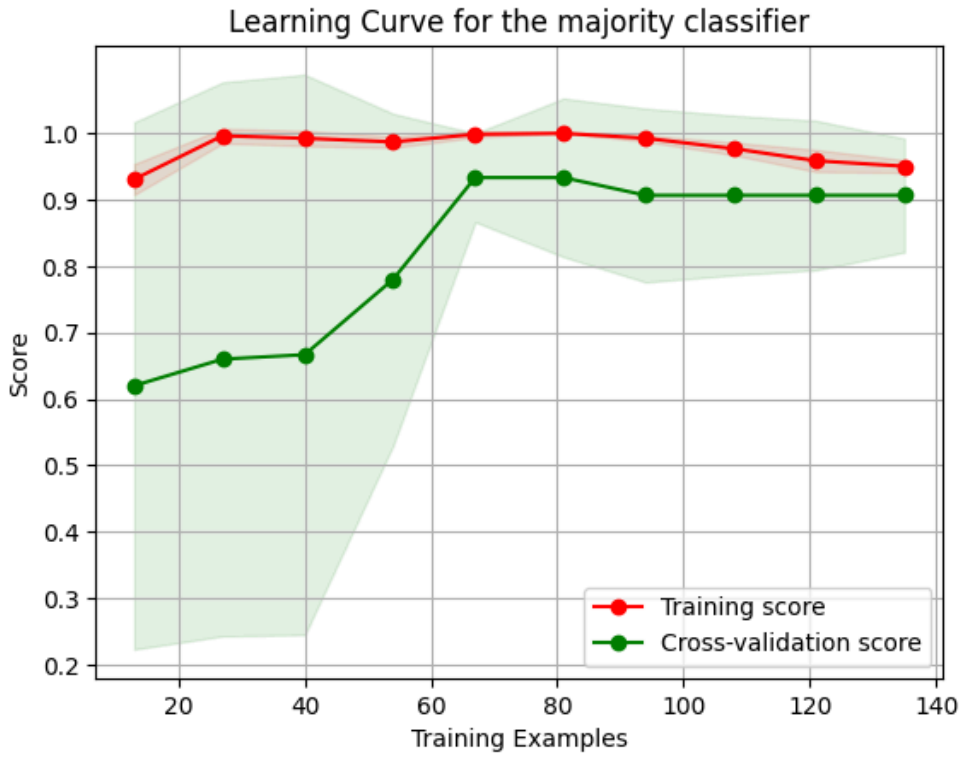
Finally, we write a new classifier that, on input data with 3 target classes, performs the split-into-three analysis on each feature and takes a majority vote

```
from sklearn.base import BaseEstimator
import numpy as np

class MajorityShrub(BaseEstimator):

    #labels are the three classes to be learned, in the correct order
```

```
    #with respect to the partition of the real line that we want
    #predictor is the list of indices that will be used to make a prediction;
    #all other features will be discardrd
    def __init__(self, labels=[], predictors=[]):
        assert len(labels) == 3
        assert len(predictors) > 0

        self.predictors = predictors
        self.labels = labels
        self.t_1 = None
        self.t_2 = None
        self.clfs = {}

    #X should be a sequence and

    #y should contain (only) terms from labels
    def fit(self, X, y=None):
        for i in self.predictors:
          clf = SimpleShrub(labels=self.labels)
          feature_data = X[:, i]
          clf.fit(feature_data, y)
          self.clfs[i] = clf

        return self

    def predict(self, X):
        predictions = [self.clfs[i].predict(X[:,i]) for i in self.clfs]
        return stats.mode(predictions)[0]

    def score(self, X, y):
      return np.mean(self.predict(X) == y)
```

We now test the new classifier in the same way as before

```
clf = MajorityShrub(labels=[0, 1, 2], predictors=selected_features)

train_sizes, train_scores, test_scores = learning_curve(clf, X, y, cv=10, train_sizes=r
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

fig, ax = plt.subplots(nrows=1, ncols=1)

ax.set_title("Learning Curve for the majority classifier")
ax.set_xlabel("Training Examples")
ax.set_ylabel("Score")
ax.set_yticks(np.linspace(0.1,1,10))
ax.fill_between(train_sizes, train_scores_mean - train_scores_std,
                train_scores_mean + train_scores_std, alpha=0.1,
                color="r")
ax.fill_between(train_sizes, test_scores_mean - test_scores_std,
                test_scores_mean + test_scores_std, alpha=0.1, color="g")
ax.plot(train_sizes, train_scores_mean, 'o-', color="r",
        label="Training score")
ax.plot(train_sizes, test_scores_mean, 'o-', color="g",
        label="Cross-validation score")
ax.legend(loc="best")
ax.grid()

fig.show()
```



We see that the performance did not improve with respect to the classifiers that only look at the petal length/width. To obtain an improvement, one approach is be more clever about how we combine the simple predictions, e.g. by assigning different importance (weights) to each simple prediction based on how sure the predictor is.