Chapter 5

Machine Learning

So far, we told the computer exactly what to do: every problem was solved by a specific algorithm. However, in the real world, we might have to deal with messy data in order to understand its underlying function. In other words, it may be difficult to separate function from noise. Maybe a computer can do part of the job, and learn some of the parameters of a function? Welcome to machine learning!

5.1 Linear Regression

Definition 5.1 (Dataset, Input, Output)**.** *A **dataset** D is a set of n tuples (x, y) sampled from an unknown function*

$$f : x \mapsto y$$

*We call $x \in X$ an **input** and $y \in Y$ the corresponding **output** of f.*

Remarks:

- We want to learn a function $\hat{f}$ such that $\hat{f}(x) \approx f(x)$.
- Since we learn $\hat{f}$ from the dataset D, we may write $\hat{f}_D$.
- For example, if the dataset consists of points on a line, we choose $\hat{f}(x) = w_1 x + w_0$ and determine the parameters $w_i \in \mathbb{R}$.
- If the points in D do not line up perfectly, our approximation function $\hat{f}$ will have some error. Even if f is truly linear, there can still be some random noise that introduces error, e.g., measurement error. → notebook

Definition 5.2 (Approximation Error)**.** *The **approximation error** $Err(x)$ denotes the deviation of $\hat{f}$ from the unknown function f at some input x:*

$$Err(x) = f(x) - \hat{f}(x).$$

Remarks:

- For our linear function this amounts to $Err(x) = y - (w_1 x + w_0)$.
- We want to minimize this error over the entire dataset D. Hence, we choose $\hat{f}$ according to an objective function as follows.

Definition 5.3 (Squared Error Loss)**.** *The **loss function** is used to determine the real-valued parameters $\boldsymbol{w} = (w_0, w_1, ...)^T$ according to the dataset D. There are several options, the most common being the **squared error loss** function:*

$$L(\hat{f}, D) = \sum_{(x,y)\in D} Err(x)^2.$$

Remarks:

- By squaring the error, we ensure each term is positive. Squaring also weighs large errors more highly.
- Another natural choice for a loss function is the absolute error: → notebook

$$L_{abs}(\hat{f}, D) = \sum_{(x,y)\in D} |Err(x)|.$$

- However, the squared error loss is often preferred as it has both a closed-form solution and is differentiable. How do we find a solution for our linear function?

Lemma 5.4. *Let $\bar{x} = \frac{1}{n}\sum_D x$ and $\bar{y} = \frac{1}{n}\sum_D y$ be the average input and output of the dataset D. For a linear function $\hat{f}(x) = w_1 x + w_0$, the squared error loss is minimal for*

$$w_1^* = \frac{\sum_{(x,y)\in D}(y-\bar{y})(x-\bar{x})}{\sum_{(x,y)\in D}(x-\bar{x})^2},\ w_0^* = \bar{y} - w_1^*\bar{x}.$$

*We call these weights the **ordinary least-square** (OLS) estimates, as they minimize the squared error loss.*

Proof. For our linear function, the squared error loss amounts to

$$L(\hat{f}, D) = \sum_{(x,y)\in D} \left(y - (w_1 x + w_0)\right)^2.$$

We find the minimum loss by differentiating $L(\hat{f}, D)$ with respect to $\boldsymbol{w}$:

$$\begin{aligned}
&\frac{\partial L}{\partial w_0} = \sum_{(x,y)\in D} -2\left(y - (w_1 x + w_0)\right) \stackrel{!}{=} 0 \\
&\iff \sum_{(x,y)\in D} (y - w_1 x - w_0) = 0 \\
&\iff \sum_{(x,y)\in D} (y - w_1 x) = n w_0 \\
&\iff w_0 = \frac{1}{n}\sum_{(x,y)\in D} y - w_1 \frac{1}{n}\sum_{(x,y)\in D} x \\
&\iff w_0 = \bar{y} - w_1 \bar{x}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial L}{\partial w_1} &= \sum_{(x,y)\in D} -2x\left(y - (w_1 x + w_0)\right) \stackrel{!}{=} 0 \\
&\iff \sum_{(x,y)\in D} x(y - w_1 x - \bar{y} + w_1 \bar{x}) = 0 \\
&\iff \sum_{(x,y)\in D} x(y - \bar{y}) = w_1 \sum_{(x,y)\in D} x(x - \bar{x}).
\end{aligned}$$

Note that $\sum_D y = n\bar{y} = \sum_D \bar{y}$, so $\sum_D \bar{x}(y - \bar{y}) = 0$ and similarly we get $\sum_D \bar{x}(x - \bar{x}) = 0$. After subtracting "0" from both sides, we obtain

$$\begin{aligned}
&\sum_{(x,y)\in D} (x - \bar{x})(y - \bar{y}) = w_1 \sum_{(x,y)\in D} (x - \bar{x})(x - \bar{x}) \\
\iff & w_1 = \frac{\sum (y - \bar{y})(x - \bar{x})}{\sum (x - \bar{x})^2}.
\end{aligned}$$

□

Remarks:

- This also works if the input is not a single scalar x but a whole vector $\boldsymbol{x}$. This is known as linear regression.

Definition 5.5 (Linear Regression, Features, Weights)**.** *With **linear regression**, we search for a function $\hat{f}$ of the form*

$$\hat{f}(\boldsymbol{x}) = \sum_{i=1}^{d-1} w_i x_i + w_0,$$

*where the $x_i \in \mathbb{R}$ are the **features** of the input. The parameters $w_i \in \mathbb{R}$ are called the **weights** and need to be determined. We can write this in vector notation as*

$$\hat{f}(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x},$$

where $\boldsymbol{x} = (1, x_1, x_2, ..., x_{d-1})^T$ with an additional 1 to incorporate the weight w_0.

Remarks:

- Let us find the weights $\boldsymbol{w}$. In order to describe the closed-form solution concisely, we use a matrix notation where $\mathbf{X}$ is a matrix of the input features (the so-called *design matrix*). $\mathbf{X}$ has n rows, each row represents a feature vector $\boldsymbol{x} = (1, x_1, x_2, ..., x_{d-1})$. The outputs are given as a vector $\boldsymbol{y}$ of length n, where each value has a corresponding row in $\mathbf{X}$.

Theorem 5.6. *The ordinary least-square (OLS) estimates for the weight parameters $\boldsymbol{w}$ of a linear regression model are given by*

$$\boldsymbol{w}^* = \left(\mathbf{X}^T \mathbf{X}\right)^{-1} \mathbf{X}^T \boldsymbol{y}.$$

Proof. The squared error loss is $L(\hat{f}, D) = \sum_{(\boldsymbol{x},y)\in D} \left(y - \boldsymbol{w}^T\boldsymbol{x}\right)^2$ which can be rewritten in matrix form as

$$L(\hat{f}, D) = (\boldsymbol{y} - \mathbf{X}\boldsymbol{w})^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}).$$

Again, we can differentiate with respect to $\boldsymbol{w}$ to find the optimal weights:

$$\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{w}} &= -\left(\mathbf{X}^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{w})\right)^T - (\boldsymbol{y} - \mathbf{X}\boldsymbol{w})^T\mathbf{X} \stackrel{!}{=} \mathbf{0}^T \\
&\iff -\mathbf{X}^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}) - \mathbf{X}^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}) = \mathbf{0} \\
&\iff \mathbf{X}^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}) = \mathbf{0} \\
&\iff \mathbf{X}^T\boldsymbol{y} = \left(\mathbf{X}^T\mathbf{X}\right)\boldsymbol{w} \\
&\iff \boldsymbol{w} = (\mathbf{X}^T\boldsymbol{X})^{-1}\mathbf{X}^T\boldsymbol{y}.
\end{aligned}$$

□

Remarks:

- We must be careful when differentiating in matrix form, as we are differentiating sums.
- We assume in this proof that $\mathbf{X}^T\mathbf{X}$ is invertible. This is for example the case when $\mathbf{X}$ has full column rank, but might not be the case in general.
- We could have derived the result in Lemma 5.4 in matrix form as well. To see that the results are the same, just expand $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\boldsymbol{y}$. Alternatively, if you consider the training data to be normalized (i.e. $\bar{x} = \bar{y} = 0$), then $w_0 = 0$ and $w_1 = \frac{\sum xy}{\sum x^2} = (\sum xx)^{-1}\sum xy$.
- What if the relation between $\boldsymbol{x}$ and y is not just linear?

5.2 Feature Modeling

Definition 5.7 (Feature)**.** *A **feature** of the input can be any real-valued term depending on the input variables.*

Remarks:

- Note that we previously used the feature vector $\boldsymbol{x} = (1, x_1, x_2, ..., x_{d-1})^T$. However, a feature can be more complex. For example, if we know in advance that the quantity $\sin x_1\sqrt{x_2}$ is a good term to approximate $f(\boldsymbol{x})$, then we can include this term as a feature in our linear regression model. Also splines, radial basis functions or wavelets work out of the box.
- When modeling a problem, we often start with an "educated guess" about which family of functions $\mathcal{F}$ is well-suited to model the unknown function f. We then restrict ourselves to find the best $\hat{f} \in \mathcal{F}$.

Definition 5.8 (Model)**.** *We call the family of functions $\mathcal{F}$ chosen to approximate f a **model**. The function $\hat{f} \in \mathcal{F}$ is found by fitting the parameters of the model to "best" represent the dataset D.*

Remarks:

- For instance, we might want to restrict $\hat{f} \in \mathcal{F}$ to polynomials (of degree m), yielding → notebook

$$\hat{f}(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + ... + w_m x^m.$$

- This is called *polynomial regression*, even though it is just a special case of linear regression (and is solved the same way).
- Note that the linear regression method can handle any model $\mathcal{F}$, as long as the (unknown) parameters w_i are linear coefficients.
- For multi-dimensional input we can add all combinations of the variables up to the m^{th} power. For polynomial regression with degree m and input dimension d, this gives $\left(\!\binom{d}{m}\!\right) = \binom{d+m-1}{m}$ features! E.g. for $m = 2$, $d = 3$, we have the 6 features $\{1, x_1, x_2, x_1^2, x_2^2, x_1 x_2\}$.
- What if some input variables are not continuous? For example, we might have a categorical input variable, such as a city name. Should we encode the city as a single variable taking values 1, 2 and 3? This would establish an inherent ordering and scaling between the cities, which would be unreasonable in many cases. One way to deal with such inputs is to use so-called one-hot encoding.

Definition 5.9 (One-Hot Encoding)**.** *A **one-hot encoding** of a categorical input variable taking k values is a vector representation of length k consisting of* 0*'s and a single* 1 *indicating the corresponding category.*

Remarks:

- Note that using a one-hot encoding may increase the dimension of the model significantly as each category gets its own coefficient w_j.
- See Figure 5.10 for an example.

x_i
London
Budapest
Zurich
-
London

$\longrightarrow$

x_{i1}	x_{i2}	x_{i3}
0	1	0
1	0	0
0	0	1
0	0	0
0	1	0

Figure 5.10: One-hot encoding of a categorical variable that can take 3 values.

5.3 Generalization & Overfitting

What if we include more and more features? For instance, we can add more input features, higher degree terms and other engineered features. Eventually we might have more features than data points, and a linear model will be able to fit the training data perfectly.

Lemma 5.11. *Given $\mathbf{X} \in \mathbb{R}^{n \times d}$ with $d \geq n$, there is at least one solution $\boldsymbol{w}$ to $\mathbf{X}\boldsymbol{w} = \boldsymbol{y}$ for all $\boldsymbol{y} \in \mathbb{R}^n$, if and only if $\mathbf{X}$ has rank n.*

Proof. This is a standard result from Linear Algebra. The proof goes along the lines of: $\mathbf{X}$ has rank $n \iff$ there are n linearly independent columns of $\mathbf{X}$, but n linearly independent columns in $\mathbb{R}^{n \times d}$ form a basis of $\mathbb{R}^{n \times n}$ so (setting w for all other columns to 0) $\mathbf{X}\boldsymbol{w} = \boldsymbol{y}$ has a unique solution for all $\boldsymbol{y} \in \mathbb{R}^n$. □

Remarks:

- For example powers of a feature, $\{1, x_i, x_i^2, x_i^3, ...\}$ are linearly independent, so polynomial regression with high enough degree will always be able to fit training data perfectly. This is called *polynomial interpolation.*
- What is the problem with adding too many features? Overfitting. The fitted model will not generalize well to unseen data. Ultimately, our goal is to minimize the expected error over *all possible* data.

Definition 5.12 (Expected Loss)**.** *We assume all our data (including any unseen data) comes from some unknown distribution $(\boldsymbol{x}, y) \sim P(X, Y)$, where X denotes the entire input space and Y the output space. Then the* ***expected loss*** *is defined as*

$$L(\hat{f}) = \mathbb{E}_{\boldsymbol{x},y}\Big[L(\hat{f}, \boldsymbol{x})\Big] = \int L(\hat{f}, \boldsymbol{x})\, dP(\boldsymbol{x}, y),$$

where $L(\hat{f}, \boldsymbol{x})$ is the loss incurred by $\hat{f}$ at data point $(\boldsymbol{x}, y)$.

Remarks:

- Expected loss is also referred to as *risk.*
- Unfortunately, we cannot calculate the expected loss as we do not know the probability distribution P. Thus we also cannot directly minimize it.
- So far, we have instead minimized the loss on our dataset D. This is called the empirical loss (or empirical risk).

Definition 5.13 (Empirical Loss)**.** *We estimate the expected loss by the empirical loss on a dataset D, given by*

$$\hat{L}_D(\hat{f}) = \frac{1}{n} \sum_{(\boldsymbol{x},y) \in D} L(\hat{f}, \boldsymbol{x}).$$

Remarks:

- The empirical loss is exactly the (normalized) total loss from Definition 5.3.
- By the law of large numbers $\hat{L}_D(\hat{f}) \to L(\hat{f})$ for any fixed $\hat{f}$ almost surely as $n \to \infty$. Therefore the more data we have, the closer the empirical loss will be to the expected loss and so, the closer $\hat{f}$ can be to the true function f by minimizing the empirical loss.

- But how do we know how well our $\hat{f}$ performs on the rest of the domain X?
- We could take the empirical loss as our estimate. Unfortunately, since we fit our model to this data, it will inherently underestimate the expected loss.
- We could find new data for evaluation, but we usually just have one dataset to work with. However, we can sample a subset D_t from our dataset D and only train our model with this subset, while reserving the rest for evaluation.

Definition 5.14 (Train-Evaluation Split)**.** *Partitioning a dataset D into two disjoint subsets D_t and D_e (typically 80% to 20%) is called a* ***train-evaluation split****.*

Remarks:

- This is also called a *train-test* split. For ease of notation we will stick with evaluation.

Definition 5.15 (Training Loss, Evaluation Loss)**.** *We define the* ***training loss*** *as*

$$\hat{L}_t(\hat{f}) = \frac{1}{|D_t|} \sum_{(\boldsymbol{x},y)\in D_t} L(\hat{f}, \boldsymbol{x}).$$

Similarly, we define the ***evaluation loss*** *as*

$$\hat{L}_e(\hat{f}) = \frac{1}{|D_e|} \sum_{(\boldsymbol{x},y)\in D_e} L(\hat{f}, \boldsymbol{x}).$$

Remarks:

- Note that $\hat{f}$ depends on the training data, so $\hat{f} = \hat{f}_{D_t}$.
- We can use the evaluation dataset D_e to estimate how well the function $\hat{f}_{D_t}$ generalizes to new data.

Definition 5.16 (Overfitting, Underfitting)**.** *A model is* ***overfitting*** *when it fits the training dataset D_t too well, learning random patterns/noise that will not be present in new unseen data D_e. The model will not generalize well. Conversely, a model is* ***underfitting*** *when it is not expressive enough to approximate f. A more complex model $\mathcal{F}'$ should be tried.* → notebook

Remarks:

- $\hat{L}_e(\boldsymbol{w}) \gg \hat{L}_t(\boldsymbol{w})$ is a clear indication of overfitting.
- See Figure 5.17 for examples of overfitting and underfitting.

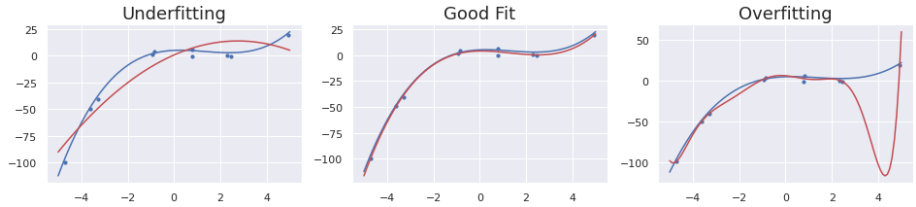

Figure 5.17: Examples of underfitting and overfitting

Remarks:

- There are several ways we can counter overfitting. We could try a simpler model, gather more training data, or introduce *regularization*.
- High training and evaluation errors could be a sign of underfitting. However, high errors could also indicate that the data is inherently noisy/random and cannot be fitted well.
- By repeating the train-evaluation split process multiple times we can get a more accurate evaluation of how well our model generalizes, and whether it is overfitting or underfitting. This is known as *cross validation*.

Definition 5.18 (Cross Validation)**.** *In k-fold cross validation, we randomly partition D into k equal sized subsets. We train a model on the union of $k-1$ of these subsets. Then we evaluate our model on the last, withheld subset. This is repeated k times, with each subset used $k-1$ times as part of the training data and once as evaluation data. This gives k evaluation scores, which are averaged to produce a single evaluation metric.*

Remarks:

- There are other types of cross validation. For example in *Monte Carlo cross validation*, D_e is randomly sampled from D each time.
- Cross validation can also be used for *model selection*: We first do a train-evaluation split and then use cross validation on the training set D_t to select our model $\mathcal{F}$. The best model is then evaluated on the as yet unseen evaluation data.
- The mean and variance of the error across splits can tell us more about the sources of error in our model.

5.4 Bias-Variance Tradeoff

Definition 5.19 (Bias, Variance)**.** ***Bias*** *is the expected error of a model, that is, how much the model $\mathcal{F}$ deviates from the target value on average. Formally,*

$$Bias^2[\mathcal{F}] = \mathbb{E}_{\boldsymbol{x}}\left[\mathbb{E}_{D,\boldsymbol{w}}\left[f(\boldsymbol{x}) - \hat{f}(\boldsymbol{x})\right]^2\right],$$

where the expectation is taken over $\boldsymbol{x}$ *and* D*. The* ***variance*** *of a model is the variance in its predictions when training on different random datasets. Formally, with* $\bar{f}(\boldsymbol{x}) = \mathbb{E}_{D,\boldsymbol{w}}\left[\hat{f}(\boldsymbol{x})\right]$*:*

$$Var[\mathcal{F}] = \mathbb{E}_{\boldsymbol{x},D,\boldsymbol{w}}\left[\left(\hat{f}(\boldsymbol{x}) - \bar{f}(\boldsymbol{x})\right)^2\right].$$

Remarks:

- If the bias is large, we know that our model $\mathcal{F}$ cannot approximate f well and we should consider a more expressive model, i.e., a more general family of functions $\mathcal{F}$. If on the other hand $f \in \mathcal{F}$, then the bias will be zero.
- Note that random noise will not contribute to the bias. On average (taking the expectation over D), the effects due to noise will cancel out. The bias really only covers systematic errors because our model is not able to match f exactly (even with the best weights). See Figure 5.20 (left).
- The variance tells us how sensitive our model is to the specific D. If the model is very sensitive, then $\boldsymbol{w}$ and ultimately the predictions will vary greatly depending on the dataset, in other words it will overfit. More noise will increase the variance. See Figure 5.20 (right).
- The simplest possible model is a constant, $\hat{f} = c$. This always has zero variance. Its predictions never change.

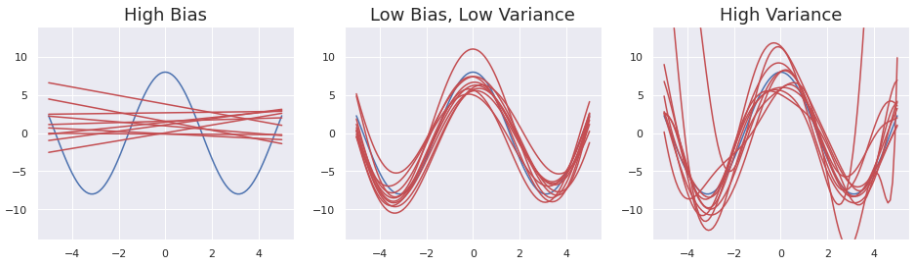

Figure 5.20: Fitting polynomials of increasing degree to the cosine function within the range $[-5, 5]$. The first order model (left) is too simple and has a high bias. On the other hand the 11^{th} order model (right) is too complex and has a high variance. The 6^{th} order model (middle) has low bias and low variance. In each iteration $f(x) = 8\cos(x)$, 25 inputs were sampled uniformly from $[-7, 7]$ and y was calculated with additional Gaussian noise. Predictions were then plotted on the interval $[-5, 5]$.

Theorem 5.21 (Bias-Variance Decomposition)**.** *We can decompose the mean squared error (*MSE*) of a model into its squared bias and its variance:*

$$\text{MSE}(\mathcal{F}) = Bias^2[\mathcal{F}] + Var[\mathcal{F}]$$

where the mean squared error (or expected squared error) equals:

$$\text{MSE}(\mathcal{F}) = \mathbb{E}_{\boldsymbol{x},D,\boldsymbol{w}}\left[\left(y - \hat{f}(\boldsymbol{x})\right)^2\right]$$

Remarks:

- In general our dataset will not cover the whole input space and the data will contain some noise. This means bias or variance or both will generally be greater than zero.
- However, there is an inherent tradeoff between bias and variance. A more complex model will be able to approximate f better, giving lower bias. But this also allows it to fit noise better leading to higher variance, since the noise it fits is random. On the other hand a simpler model will generally have higher bias and lower variance.

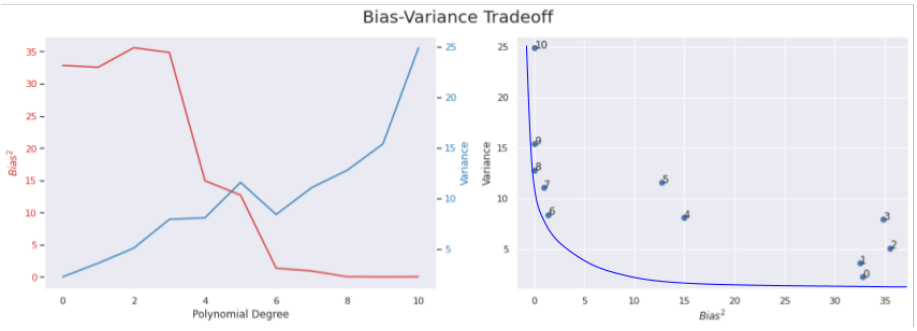

Figure 5.22: Fitting polynomials of increasing degree to the cosine function within the range $[-5, 5]$. On the left we see that as the degree (complexity) of the model increases, the bias decreases and the variance increases. The plot on the right shows the bias variance tradeoff frontier.

- If we have high variance, regularization is one way of reducing the total error. Regularization focuses on decreasing the variance at the potential expense of increasing the bias.

5.5 Regularization

How about including all the features we may possibly want, but charging a cost for each non-zero weight w_i? This should help us reduce overfitting to noise, but still allows us to fit f well. This is what Lasso does.

Definition 5.23 (Lasso Regression)**.** *Lasso regression minimizes*

$$\min_{\boldsymbol{w}} \left\{ \frac{1}{n} \sum_{(\boldsymbol{x},y)\in D} (y - \boldsymbol{w}^T\boldsymbol{x})^2 + \lambda \sum_{i=0}^{d-1} |w_i| \right\}$$

Remarks:

- The regularization parameter λ weighs the parameter cost versus the MSE loss. It is a so-called *hyperparameter* that can be tuned.

Definition 5.24 (Hyperparameter)**.** *A* ***hyperparameter*** *is a parameter that controls the training process and whose value has to be chosen in advance.*

Remarks:

- The degree m in polynomial regression was also a hyperparameter.
- Cross validation can be used for hyperparameter tuning.
- For each hyperparameter you can define a set of values. Exhaustively iterating through all combinations of these values is called *grid search*.

Remarks:

- Lasso introduces bias into the regression solution by guiding the weights to be close to zero. This can reduce variance considerably relative to the OLS solution, see Figure 5.25.

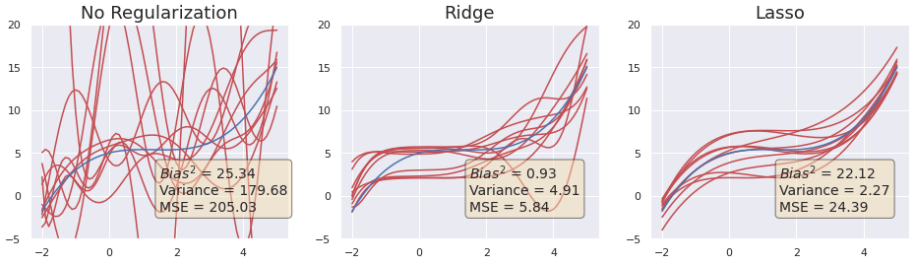

Figure 5.25: Fitting a polynomial of high degree to a cubic function with and without regularization. We see that variance and total error drops when regularization is added.

Remarks:

- When using regularization, features should be normalized (subtract mean, divide by standard deviation). This ensures that the coefficient of a feature is not influenced by its magnitude.
- The target vector $\boldsymbol{y}$ should also be centered around 0, so that the intercept w_0 does not count towards the total cost.
- Ridge has the same objective, but Ridge uses the L^2 norm for the cost of the weights, whereas Lasso uses the L^1 norm. In other words, we replace $|w_i|$ with w_i^2.
- Ridge has a closed form solution, just like OLS. However Lasso does not, so we have to solve it differently, for example by *gradient descent*.

5.6 Gradient Descent

The OLS method had a closed form solution (Lemma 5.4 and Theorem 5.6), but often we do not have a closed form solution to our optimization problem. An alternative is to minimize the loss function using gradient based methods. If we can calculate the gradient of the loss function with respect to the weights, then we can perturb the weights in the right direction to decrease the loss. We can repeat this process until reaching a minimum.

Definition 5.26 (Gradient Descent)**.** *Given a loss function $L(\hat{f}(\boldsymbol{w}), D)$, repeatedly perform the update*

$$w_j := w_j - \alpha \frac{\partial}{\partial w_j} L(\hat{f}, D)$$

simultaneously for $j = 0, 1, ..., d$, where hyperparameter α is the ***learning rate****.*

Remarks:

- The partial derivative tells us which weights to decrease or increase. If $\frac{\partial}{\partial w_j} L(\hat{f}, D)$ is positive, then the loss is increasing in w_j, so we decrease w_j to lower the loss.
- The gradient, i.e., the vector of partial derivatives, $\nabla L(\hat{f}(\boldsymbol{w}), D)$ points in the direction of steepest ascent, with the negation pointing in the direction of steepest descent.
- A (loss) function can have multiple minima, and gradient descent may well converge to a local minimum rather than finding the global minimum. The minimum reached depends on the initial starting point and the learning rate α. See Figure 5.27.

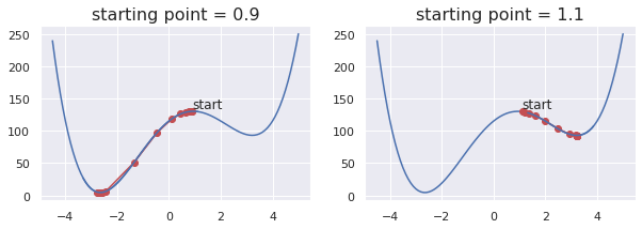

Figure 5.27: Gradient descent with different initialization points. On the (left) the initialization is favorable and we reach the global minimum of the polynomial function. However on the (right) gradient descent gets stuck in a local minimum.

Remarks:

- The *learning rate* α is a hyperparameter that controls the size of each update step. If the learning rate is too high, the algorithm might jump beyond the optimum $\boldsymbol{w}$; if it is too low the algorithm will be very slow to converge. See Figure 5.28 for examples. Often a decaying learning rate is used for efficiency at the beginning and accuracy towards the end of training.

→ notebook

Figure 5.28: Gradient descent with different learning rates for finding the minimum of a quadratic function. On the (left) the learning rate is very low so convergence is slow and on the (right) the learning rate is too high so gradient descent diverges.

Remarks:

- Gradient descent is very similar to Newton's method for optimization, but in Newton's method the learning rate is not a hyperparameter, but 1 over the second derivative of the loss function, $\frac{\partial^2}{\partial w_j^2} L(\hat{f}, D)$. Newton's method often converges in fewer steps, but unfortunately the second derivative is usually hard to calculate.
- Any differentiable loss function can be optimized with gradient descent. If the loss function is convex (see Definition 1.28), then the global minimum will eventually be reached (with a suitable learning rate).
- When the training dataset is large, it is often too costly to calculate $\frac{\partial}{\partial w_j} L(\hat{f}, D)$ over the whole dataset D just to make a single update step. In practice the training data is often shuffled and split into minibatches D_i (subsets of equal size) and $\frac{\partial}{\partial w_j} L(\hat{f}, D_i)$ is used in the update step. This is called *stochastic gradient descent* (SGD).
- Even though we have a closed form solution for minimizing the loss in linear regression, let's derive the corresponding update rule to see how this works.

Theorem 5.29 (LMS Rule)**.** *The **Least Mean Squares** (LMS) update rule for linear regression is given by*

$$w_j := w_j + \alpha \left(y - \boldsymbol{w}^T \boldsymbol{x}\right) x_j$$

And the batch update rule is given by

$$w_j := w_j + \alpha \sum_{(\boldsymbol{x}, y) \in D} \left(y - \boldsymbol{w}^T \boldsymbol{x}\right) x_j$$

Proof. We derive the batch update rule. Recall the squared error loss function

$$L(\hat{f}, D) = (\boldsymbol{y} - \mathbf{X}\boldsymbol{w})^T (\boldsymbol{y} - \mathbf{X}\boldsymbol{w}).$$

And recall from Theorem 5.6 that the derivative with respect to $\boldsymbol{w}$ is given by

$$\frac{\partial L}{\partial \boldsymbol{w}} = -2\mathbf{X}^T (\boldsymbol{y} - \mathbf{X}\boldsymbol{w})$$

Substituting into the gradient descent formula, we get the update rule

$$\begin{aligned} w_j &:= w_j + 2\alpha \left(\mathbf{X}^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{w})\right)_j \\ &= w_j + 2\alpha \sum_{(\boldsymbol{x},y)\in D} \left(y - \boldsymbol{w}^T\boldsymbol{x}\right) x_j \end{aligned}$$

If we scale the learning rate by a half we get the LMS batch update rule. And taking a single training sample for D we get the single update rule. □

5.7 Logistic Regression

So far we have discussed how to learn a function $f: x \mapsto y.$ when $y \in \mathbb{R}$. In this section we introduce a method for binary classification, where y is either 0 or 1, i.e., $y \in \{0, 1\}$. Similar to linear regression, we learn a linear function $\boldsymbol{w}^T\boldsymbol{x}$, but now we classify all samples with $\boldsymbol{w}^T\boldsymbol{x} > 0$ as 1, all samples with $\boldsymbol{w}^T\boldsymbol{x} < 0$ as 0. In other words we find a hyperplane to separate the classes. There are different approaches to do so, we present a statistically motivated approach called logistic regression. The key idea is simple: We take the output of the linear function and squash it into the range $[0, 1]$. We treat this squashed output as a probability. The more sample $\boldsymbol{x}$ is in the direction of vector $\boldsymbol{w}$, the higher the probability that sample $\boldsymbol{x}$ belongs to class 1.

Definition 5.30 (Binary Logistic Regression)**.** *We want to find an approximation* $\hat{f}(\boldsymbol{x}) \approx f(\boldsymbol{x}) = y \in \{0, 1\}$*. We choose the following form for* $\hat{f}$*:* → notebook

$$\hat{f}(\boldsymbol{x}) = \psi(\boldsymbol{w}^T\boldsymbol{x}) = \frac{1}{1 + \exp(-\boldsymbol{w}^T\boldsymbol{x})}$$

where ψ *is called the* ***logistic*** *or* ***sigmoid*** *function.*

Remarks:

- The logistic function ψ squashes inputs from $[-\infty, \infty]$ into the range $[0, 1]$. Other functions, such as the *probit link function*, could be used instead.
- $\hat{f}(\boldsymbol{x})$ can be seen as an estimate of the probability that $y = 1$. Therefore, we usually classify $\boldsymbol{x}$ as positive (1) if $\hat{f}(\boldsymbol{x}) > 0.5$ and as negative (0) if $\hat{f}(\boldsymbol{x}) < 0.5$.
- This defines a *decision boundary* $\psi(\boldsymbol{w}^T\boldsymbol{x}) = 0.5$, or $\boldsymbol{w}^T\boldsymbol{x} = 0$, where everything on one side of the boundary is classified as positive and everything on the other side as negative. Samples on the boundary can be classified arbitrarily as positive or negative. The decision boundary is linear in the features.
- As in linear regression, we again can add higher order features, e.g., x^2 or $sin(x)$, to learn non-linear decision boundaries.
- How do we find the optimal values for $\boldsymbol{w}$? We could minimize the squared error loss (Definition 5.3). However, this would not guarantee predictions between 0 and 1 and we could not interpret the predictions as probabilities.

- Instead we apply the logistic function, interpret the output as a probability and choose the model that maximizes the likelihood of generating exactly the labels in our training data.

Definition 5.31 (Bernoulli Likelihood function)**.**

$$\mathcal{L}(\boldsymbol{w}) = \prod_{(\boldsymbol{x},y)\in D} P(y \mid \boldsymbol{x}, \boldsymbol{w}) = \prod_{(\boldsymbol{x},1)\in D} \psi(\boldsymbol{w}^T\boldsymbol{x}) \prod_{(\boldsymbol{x},0)\in D} (1 - \psi(\boldsymbol{w}^T\boldsymbol{x}))$$

Remarks:

- $\mathcal{L}(\boldsymbol{w})$ is the probability of observing the vector of outputs $\boldsymbol{y}$ given input data $\mathbf{X}$ and parameters $\boldsymbol{w}$. Intuitively, we want to choose $\boldsymbol{w}$ such that we maximize this probability, i.e., we want to choose the parameter values that make our observations the most likely to occur. This is called Maximum Likelihood Estimation (MLE).

Lemma 5.32 (Logistic Regression Loss Function or Log Loss)**.** *Assuming the y's in $\boldsymbol{y}$ are independent and identically distributed Bernoulli with parameters $p = \hat{f}(\boldsymbol{x}) = \psi(\boldsymbol{w}^T\boldsymbol{x})$, maximizing $\mathcal{L}(\boldsymbol{w})$ is equivalent to minimizing the following loss function:*

$$L(\hat{f}, D) = -\frac{1}{n} \sum_{(\boldsymbol{x},y)\in D} \Big[y \log(\hat{f}(\boldsymbol{x})) + (1-y) \log(1 - \hat{f}(\boldsymbol{x})) \Big]$$

Proof.

$$\begin{aligned}
\mathcal{L}(\boldsymbol{w}) &= \prod_{(\boldsymbol{x},1)\in D} \psi(\boldsymbol{w}^T\boldsymbol{x}) \prod_{(\boldsymbol{x},0)\in D} (1 - \psi(\boldsymbol{w}^T\boldsymbol{x})) \\
&= \prod_{(\boldsymbol{x},y)\in D} (\psi(\boldsymbol{w}^T\boldsymbol{x}))^y (1 - \psi(\boldsymbol{w}^T\boldsymbol{x}))^{1-y} \\
&= \prod_{(\boldsymbol{x},y)\in D} \hat{f}(\boldsymbol{x})^y (1 - \hat{f}(\boldsymbol{x}))^{1-y}
\end{aligned}$$

In practice we maximize the logarithm of the likelihood function because the product simplifies to a sum, which prevents numerical problems and makes differentiation simpler. Taking the logarithm does not change the optimal $\boldsymbol{w}$:

$$\begin{aligned}
\log \mathcal{L}(\boldsymbol{w}) &= \sum_{(\boldsymbol{x},y)\in D} \Big[y \log(\hat{f}(\boldsymbol{x})) + (1-y) \log(1 - \hat{f}(\boldsymbol{x})) \Big] \\
&= -n \cdot L(\hat{f}, D)
\end{aligned}$$

Therefore $\operatorname{argmax}_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \operatorname{argmin}_{\boldsymbol{w}} L(\hat{f}, D)$.

□

Remarks:

- The scaling factor n does not change the optimum $\boldsymbol{w}$, but it averages the value of the loss across samples, hence allowing for better comparison across models.
- If $y = 1$ and $\hat{f} = 1$, then $L = -\log(\hat{f}) = -\log(1) = 0$, i.e, our classifier is correct with perfect "confidence", and hence the loss is zero. If on the other hand $y = 1$ and $\hat{f} = 0$, then $L = -\log(\hat{f}) = -\log(0) \approx \infty$, i.e., our classifier is wrong with perfect "confidence", which incurs very high cost. Similarly if $y = 0$ and $\hat{f} = 0$, then $L = 0$, and if $y = 0$ and $\hat{f} = 1$, then $L = \infty$.
- Unfortunately, there is in general no closed form solution for logistic regression. However, we can use gradient descent (Definition 5.26) to minimize $L(\hat{f}, D)$. But first we need to calculate the gradient.

Lemma 5.33 (Gradient of the Log Loss)**.** *The gradient of $L(\hat{f}, D)$ from Lemma 5.32 with respect to w_j is given by*

$$\frac{\partial L}{\partial w_j} = \frac{1}{n} \sum_{(\boldsymbol{x},y)\in D} \Big[\hat{f}(\boldsymbol{x}) - y\Big] \cdot x_j$$

Proof. First we calculate the derivative of the logistic function:

$$\begin{aligned}
\frac{d}{dz}\psi(z) &= \frac{d}{dz}\left[\frac{1}{1+e^{-z}}\right] \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} \\
&= \psi(z) \cdot (1 - \psi(z))
\end{aligned}$$

Using this we can calculate the derivative of the loss for a single training sample, $L(\hat{f}, \boldsymbol{x})$, with respect to w_j:

$$\begin{aligned}
&\frac{\partial}{\partial w_j} L(\hat{f}, \boldsymbol{x}) = -\frac{\partial}{\partial w_j}\Big[y \log(\hat{f}(\boldsymbol{x})) + (1-y)\log(1-\hat{f}(\boldsymbol{x}))\Big] \\
&= -\frac{\partial}{\partial w_j}\left[y \log(\psi(\boldsymbol{w}^T\boldsymbol{x})) + (1-y)\log(1-\psi(\boldsymbol{w}^T\boldsymbol{x}))\right] \\
&= -\left[\frac{y}{\psi(\boldsymbol{w}^T\boldsymbol{x})} - \frac{1-y}{1-\psi(\boldsymbol{w}^T\boldsymbol{x})}\right] \cdot \frac{\partial}{\partial w_j}\psi(\boldsymbol{w}^T\boldsymbol{x}) && \text{chain rule} \\
&= -\left[\frac{y}{\psi(\boldsymbol{w}^T\boldsymbol{x})} - \frac{1-y}{1-\psi(\boldsymbol{w}^T\boldsymbol{x})}\right] \cdot x_j \cdot \psi(\boldsymbol{w}^T\boldsymbol{x})(1-\psi(\boldsymbol{w}^T\boldsymbol{x})) && \text{chain rule} \\
&= -\left[\frac{y - \psi(\boldsymbol{w}^T\boldsymbol{x})}{\psi(\boldsymbol{w}^T\boldsymbol{x})(1-\psi(\boldsymbol{w}^T\boldsymbol{x}))}\right] \cdot x_j \cdot \psi(\boldsymbol{w}^T\boldsymbol{x})(1-\psi(\boldsymbol{w}^T\boldsymbol{x})) \\
&= -\Big[y - \psi(\boldsymbol{w}^T\boldsymbol{x})\Big] \cdot x_j \\
&= \Big[\hat{f}(\boldsymbol{x}) - y\Big] \cdot x_j
\end{aligned}$$

And since differentiation and finite summation are interchangeable, i.e.,

$$\frac{d}{dx}\sum g(x) = \sum \frac{d}{dx} g(x)$$

we get the gradient for the total loss, $L(\hat{f}, D)$, as:

$$\frac{\partial L(\hat{f}, D)}{\partial w_j} = \frac{1}{n} \sum_{(\boldsymbol{x}, y) \in D} \left[\hat{f}(\boldsymbol{x}) - y\right] \cdot x_j$$

□

Remarks:

- We can then use gradient descent (Def. 5.26) to update the model parameters.
- We can also add lasso or ridge regularization to prevent our model from overfitting (Def. 5.23).
- What if y can take on more than two values? A straightforward way to extend binary logistic regression to $k > 2$ classes is to train k separate logistic regression models, one for each class. We then choose the class with the highest probability score. This is a general method for extending binary classifiers to multinomial problems, and is referred to as One versus Rest (OvR).
- There are other ways to extend logistic regression to the multinomial case. For example, we can use the softmax function instead of the sigmoid function.

Definition 5.34 (Softmax Regression)**.** ***Softmax regression*** *with $k \geq 2$ classes chooses the following functional form for $\hat{f}$:*

$$\hat{f}(\boldsymbol{x})_i = \sigma(\boldsymbol{w}_{(1)}^T \boldsymbol{x}, \boldsymbol{w}_{(2)}^T \boldsymbol{x}, ..., , \boldsymbol{w}_{(k)}^T \boldsymbol{x})_i = \frac{\exp(\boldsymbol{w}_{(i)}^T \boldsymbol{x})}{\sum_{j=1}^{k} \exp(\boldsymbol{w}_{(j)}^T \boldsymbol{x})} \qquad \text{for } i = 1, \ldots, k$$

where σ is called the ***softmax function****.*

Remarks:

- $\hat{f}(\boldsymbol{x})$ is a vector of length k, with elements summing up to 1. It can be seen as a vector of probabilities, with $\hat{f}(\boldsymbol{x})_i \approx \mathbb{P}(f(\boldsymbol{x}) = i)$.
- A different set of linear weights, $\boldsymbol{w}_{(i)}$, is learned for each class i. Since the probabilities sum up to 1, one set of weights is "redundant".
- For $k = 2$, softmax regression reduces exactly to binary logistic regression with $\boldsymbol{w} = \boldsymbol{w}_{(2)} - \boldsymbol{w}_{(1)}$.
- We also get the same form for the loss function:

Lemma 5.35 (Softmax Regression Loss Function)**.** *Assuming the y's in Y are independent and identically distributed categorical random variables with parameters $p_i = \sigma(\boldsymbol{w}_{(i)}^T \boldsymbol{x})$ for $i = 1, \ldots, k$, then maximizing the likelihood $\mathcal{L}(\boldsymbol{w})$ is equivalent to minimizing the following loss function:*

$$L(\hat{f}, D) = -\frac{1}{n} \sum_{(\boldsymbol{x},y)\in D} \sum_{i=1}^{k} \Big[\mathbb{1}\{y = i\} \log(\hat{f}(\boldsymbol{x})_i) \Big]$$

where $\boldsymbol{w}$ is the set of all weights $\{\boldsymbol{w}_{(i)}\}_{i=1,\ldots k}$ and $\mathbb{1}\{\cdot\}$ is the indicator function.

Remarks:

- Note that for $k = 2$ this loss is identical to the loss for logistic regression in Lemma 5.32, only the notation has been changed to use the indicator function.
- In order to learn non-linear decision boundaries with logistic regression, we need to do feature engineering. This can be challenging and time consuming, and it also makes the resulting model more difficult to interpret. In the following we introduce a different type of model that addresses these issues: Decision Trees.

5.8 Tree-Based Methods

So far we have considered regression models based on the form $\boldsymbol{w}^T \boldsymbol{x}$ where the output is essentially a weighted sum of the input features, potentially squashed by a sigmoid function. Binary decision trees introduce hierarchy: We keep partitioning the feature space into smaller regions. In order to make a prediction $\hat{f}(\boldsymbol{x})$, we simply start at the root of the decision tree and apply the decision rules until we reach a leaf, which then determines the output value.

```
1 def predict(self, x): # self is current node, initially root
2     if self.is_leaf():
3         return self.value
4     if x[self.feature] ≤ self.threshold:
5         self.left.predict(x)
6     else:
7         self.right.predict(x)
```

Algorithm 5.36: Decision tree algorithm.

Remarks:

- For classification, in Line 3 we return the majority class of the leaf. For regression, we return the average value of the leaf.
- Decision trees can learn non-linear decision boundaries and are easy to interpret and visualize. → notebook

- Decision trees are binary trees that contain nodes V, where each internal (non-leaf) node has an associated splitting rule. Every node v corresponds to a subset $D_v \subseteq D$.
- How are splitting rules defined?

Definition 5.37 (Decision Tree Splitting Rule)**.** *A **splitting rule** of an internal node v is given by a tuple (i, t), where i is the index of a feature and t is a threshold value. A split defines **left** and **right** subsets*

$$D_{v,l} = \{\boldsymbol{x} \mid x_i \leq t\}, \quad D_{v,r} = \{\boldsymbol{x} \mid x_i > t\}$$

Remarks:

- Unlike everything so far, a splitting rule here is based on a single feature value and not a linear combination of features. As such all splits are axis-aligned.
- Now we know how to use a decision tree to make predictions, but how do we build a decision tree in the first place? We do so by finding good splitting rules. And we find good splitting rules by minimizing a loss function of course!

Definition 5.38 (Regression tree loss: MSE)**.** *For node v with samples D_v the mean squared error (*MSE*) loss is defined as:*

$$L(D_v) = \frac{1}{|D_v|} \sum_{y \in D_v} (y - \bar{y})^2$$

where the prediction $\bar{y}$ of a node is the average of the target values of all samples in D_v.

Remarks:

- The MSE is the variance of the target value. The aim is to create splits that lower the total variance in the leaves.
- For classification the loss function is a measure of purity. We call a node with all samples belonging to the same class perfectly pure. We aim to find splits that successively increase the purity of the nodes. The most common measures of purity are *entropy* and *Gini impurity*.
- We now have a measure of loss for each node, but we need to combine the loss of the left and right subsets to decide what the next split should be. One could consider many things here: minimizing the total loss, minimizing the maximum loss, making balanced splits, etc. One natural choice is to minimize the weighted average loss.

Definition 5.39 (CART loss function)**.** *To find the best split (i^*, t^*) at node v, CART minimizes the following loss function:*

$$L(i, t) = \frac{|D_{v,l}|}{|D_v|} L(D_{v,l}) + \frac{|D_{v,r}|}{|D_v|} L(D_{v,r})$$

where the loss $L(\cdot)$ measures the impurity or error of the resulting left and right subsets.

Remarks:

- The weights ensure that all training samples have equal contribution. For example, for regression trees the weighted MSE loss reduces to the mean squared error over D_v. This is not the same as $L(D_v)$, since we use the two new mean values $\bar{y}_l$ and $\bar{y}_r$ to calculate the errors.
- CART trees are built recursively using binary splits, with every split minimizing above loss function.
- One could keep splitting until all the leaves contain single samples, like a binary search tree (Definition 4.2). This would give 100% accuracy on the training data, but would not generalize well to new data. Stopping criteria can be used to halt splitting. Typical stopping criteria are: maximum tree depth, minimum number of samples in a node, minimum decrease in the value of the loss function.
- Instead of stopping criteria, it is generally better to grow a large tree and then prune it back. This way we might add some useful additional splits and have the chance to remove less useful splits. Even if all additional splits are not helpful, we can still remove them when pruning. In essence pruning allows us to see some steps into the future, before finalizing our tree.
- CART is a greedy algorithm that finds good solutions, but is unlikely to find the optimal solution. Finding the optimal tree (e.g. a minimum depth decision tree) is NP-hard.
- One big advantage of decision trees is their ease of interpretability, but they suffer from high variance and overfit easily. A common technique to improve decision trees is to use many trees in an ensemble.

Definition 5.40 (Bootstrap Sample)**.** *A bootstrap sample D_b is obtained by drawing n samples from dataset D uniformly with replacement.*

Remarks:

- In expectation, every D_b contains $1 - 1/e = 63.2\%$ samples from D.
- What happens if we draw many bootstrap samples and use each to train a separate model?

Definition 5.41 (Bootstrap Aggregating or Bagging)**.** *Bagging is an ensemble algorithm where q models are trained on q bootstrap samples D_b. The models are combined either by averaging the outputs (regression) or by majority vote (classification).*

Remarks:

- Bagging ensembles can be built with any base learners, including decision trees. In general, an ensemble method can turn many low-bias high-variance base learners into a single low-bias low-variance model.

- The use of bootstrap samples de-correlates the individual learners, i.e., they will generally make different mistakes which can be averaged out.

- If we use decision trees as our base learners we can do even better. What if we add even more randomness to bagging by subsampling the features that can be chosen for finding the best splits?

Definition 5.42 (Random Forest)**.** *A random forest is a bagging ensemble of q decision trees. The learners $\hat{f}_b$ are trained in such a way that at every node only a random subset of the features can be used for finding the best split.*

Remarks:

- The random subsampling of features for each split further de-correlates the individual trees, making random forests powerful models that can achieve both relatively low bias and low variance.

5.9 Evaluation

How can we know whether our models are performing well? For regression we can simply measure the models MSE (or absolute error) on the evaluation set (Definition 5.15). However, for classification, the value of the loss function is not intuitive; it does not directly tell us how good our model is at classifying samples. To get a sense of the performance of a classifier, the most important tool is the confusion matrix.

Definition 5.43 (Confusion Matrix)**.** *A confusion matrix, also known as error matrix, visualizes the performance of a classifier on a given dataset. Rows represent the actual class labels, and columns contain the predictions of the classifier.*

		Predicted label	
		p	**n**
True label	**p′**	True Positive TP	False Negative FN
	n′	False Positive FP	True Negative TN

Remarks:

- From the confusion matrix we can derive different metrics to summarize the performance of a classifier:

accuracy	$\mathsf{ACC} = \frac{\mathsf{TP}+\mathsf{TN}}{\mathsf{P}+\mathsf{N}}$
positive predictive value (precision)	$\mathsf{PPV} = \frac{\mathsf{TP}}{\mathsf{TP}+\mathsf{FP}}$
true positive rate (recall)	$\mathsf{TPR} = \frac{\mathsf{TP}}{\mathsf{TP}+\mathsf{FN}}$
false positive rate	$\mathsf{FPR} = \frac{\mathsf{FP}}{\mathsf{FP}+\mathsf{TN}}$
F1 score	$F_1 = 2 \cdot \frac{\mathsf{PPV} \cdot \mathsf{TPR}}{\mathsf{PPV}+\mathsf{TPR}}$

Remarks:

- Confusion matrices and the derived metrics can be used with more than two classes. However, the derived metrics can then be computed in several different ways. The *macro* method first computes the metrics for each class and then averages these values. The *micro* method aggregates the predictions from all samples and computes the metrics directly.

- Another very common way of evaluating binary classifiers ($k = 2$) is the Receiver Operator Characterstic (ROC) Curve and the derived Area Under Curve (AUC) metric.

Definition 5.44 (Receiver Operator Characteristic (ROC) Curve)**.** *The ROC Curve plots the TPR against the FPR. Given a binary classifier $\hat{f}$, one can order the data points by $\hat{f}(\boldsymbol{x})$, and then plot the TPR and FPR for every possible classification threshold $\tau_{pr} \in [0, 1]$. The resulting graph is called the ROC Curve.*

Remarks:

- τ_{pr} is the threshold for which if $\hat{f}(\boldsymbol{x}) \geq \tau_{pr}$ we classify a sample as positive, and otherwise as negative.

- Trade-off between TPR and FPR: If we want 100% TPR (recall), we can simply classify every sample as positive. However, we would then also (wrongly) classify every negative sample as positive, leading to a high FPR. An increase in the TPR generally leads to an increase in the FPR.

- A perfect classifier would achieve $TPR = 1$ at $FPR = 0$, i.e., the curve would "hug" the top left corner.

- The ROC curve of a random classifier follows the diagonal, i.e., $TPR = FPR$.

- The area under the ROC curve (AUC) is often used as a convenient summary of a classifier's performance

- Intuitively, the AUC metric tells us the following: Given a random negative sample $\boldsymbol{x}_N$ and a random positive sample $\boldsymbol{x}_P$, what is the probability that $\hat{p}(\boldsymbol{x}_P) > \hat{p}(\boldsymbol{x}_N)$, i.e, the classifier will assign higher probability to the random positive sample than to the negative sample.

Chapter Notes

This chapter was written in collaboration with Gino Brunner and Béni Egressy.

Bibliography