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

$$f : x \rightarrow y$$

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

Remarks:
• We want to learn a function $f$ such that $f(x) \approx f(x)$.
• Since we learn $f$ from the dataset $D$, we may write $f_D$.
• For example, if the dataset consists of points on a line, we choose $f(x) = w_1x + w_0$ and determine the parameters $w \in \mathbb{R}$.
• If the points in $D$ do not line up perfectly, a linear approximation function $f$ will have some error. Even if $f$ is truly linear, there can still be some random noise that introduces error, e.g., some measurement error.

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

$$Err(x) = f(x) - f(x).$$

Remarks:
• For our linear function this amounts to $Err(x) = y - (w_1x + w_0)$.
• We want to minimize this error over the entire dataset $D$. Hence, we choose $f$ according to an objective function as follows.

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

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

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: $\rightarrow$ notebook

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

• However, the squared error loss is often preferred as it has both a closed-form solution and is everywhere differentiable. How do we find such 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 $f(x) = w_1x + w_0$, the squared error loss is minimal for

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

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

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

$$L(f, D) = \sum_{(x, y) \in D} (y - (w_1x + w_0))^2.$$

We find the minimum loss by differentiating $L(f, D)$ with respect to $w$:

$$\frac{\partial L}{\partial w_1} = \sum_{(x, y) \in D} 2(y - (w_1x + w_0)) (x - \bar{x}) = 0$$

$$\iff \sum_{(x, y) \in D} (y - w_1x - w_0) = 0$$

$$\iff \sum_{(x, y) \in D} (y - w_1x) = nw_0$$

$$\iff w_0 = \frac{1}{n} \sum_{(x, y) \in D} y - w_1 \bar{x} \sum_{(x, y) \in D} x$$

$$\iff w_0 = \bar{y} - w_1 \bar{x}$$

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

$$f : x \rightarrow y$$

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

Remarks:
• We want to learn a function $f$ such that $f(x) \approx f(x)$.
• Since we learn $f$ from the dataset $D$, we may write $f_D$.
• For example, if the dataset consists of points on a line, we choose $f(x) = w_1x + w_0$ and determine the parameters $w \in \mathbb{R}$.
• If the points in $D$ do not line up perfectly, a linear approximation function $f$ will have some error. Even if $f$ is truly linear, there can still be some random noise that introduces error, e.g., some measurement error.

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

$$Err(x) = f(x) - f(x).$$

Remarks:
• For our linear function this amounts to $Err(x) = y - (w_1x + w_0)$.
• We want to minimize this error over the entire dataset $D$. Hence, we choose $f$ according to an objective function as follows.

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

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

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: $\rightarrow$ notebook

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

• However, the squared error loss is often preferred as it has both a closed-form solution and is everywhere differentiable. How do we find such 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 $f(x) = w_1x + w_0$, the squared error loss is minimal for

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

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

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

$$L(f, D) = \sum_{(x, y) \in D} (y - (w_1x + w_0))^2.$$

We find the minimum loss by differentiating $L(f, D)$ with respect to $w$:
5.2 Feature Modeling

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

$$L(f, D) = (y - Xw)^T (y - Xw).$$

Again, we can differentiate with respect to $w$ to find the optimal weights:

$$\frac{\partial L}{\partial w} = -2X^T (y - Xw) = 0 \iff X^T (y - Xw) = 0 \iff X^T y = X^T X w \iff w = (X^T X)^{-1} X^T y.$$

Remarks:
- We must be careful when differentiating in matrix form, as we are differentiating sums.
- We assume in this proof that $X^T X$ is invertible. This is for example the case when $X$ 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 $(X^T X)^{-1} X^T y$. Alternatively, if you consider the training data to be normalized ($\bar{x} = \bar{y} = 0$), then $w_0 = 0$ and $w_1 = \frac{\sum x_i y_i}{\sum x_i^2} = (\sum x_i)^{-1} \sum x_i y_i$.
- What if the relation between $x$ 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 $x = (1, x_1, x_2, \ldots, x_{d-1})^T$. However, a feature can be more complex. For example, if we know in advance that the quantity $\sin(x_1)$ is a good term to approximate $f(x)$, 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 $F$ is well-suited to model the unknown function $f$. We then restrict ourselves to find the best $f \in F$.

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

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

$$w^* = (X^T X)^{-1} X^T y.$$
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 $X \in \mathbb{R}^{n \times d}$ with $d \geq n$, there is at least one solution $u$ to $Xu = y$ for all $y \in \mathbb{R}^n$, if and only if $X$ has rank $n$.

Proof. This is a standard result from Linear Algebra. The proof goes along the lines of:

- Note that the linear regression method can handle any model $F$, 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 $m^k$ power. For polynomial regression with degree $m$ and input dimension $d$, this gives $\binom{m+k}{k}$ features! E.g. for $m=2$, $d=3$, we have the 6 features $\{x_1, x_2, x_1^2, x_1 x_2, x_2^2, x_2 x_3\}$.
- 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 $w_i$.
- See Figure 5.10 for an example.

<table>
<thead>
<tr>
<th>City</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Budapest</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Zurich</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>n/a</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.10: One-hot encoding of a categorical variable that can take 3 values (as well as n/a).

5.3 GENERALIZATION & OVERFITTING

<table>
<thead>
<tr>
<th>City</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Budapest</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Zurich</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>n/a</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

5.3.3 GENERALIZATION & OVERFITTING

Remarks:
- For instance, we might want to restrict $\hat{f} \in F$ to polynomials of degree $m$, yielding

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

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 $F$, 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 $m^k$ power. For polynomial regression with degree $m$ and input dimension $d$, this gives $\binom{m+k}{k}$ features! E.g. for $m=2$, $d=3$, we have the 6 features $\{x_1, x_2, x_1^2, x_1 x_2, x_2^2, x_2 x_3\}$.
- 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 $w_i$.
- See Figure 5.10 for an example.

<table>
<thead>
<tr>
<th>City</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Budapest</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Zurich</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>n/a</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.10: One-hot encoding of a categorical variable that can take 3 values (as well as n/a).

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.
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

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

Similarly, we define the evaluation loss as

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

**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 \( F' \) should be tried.

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

5.4 Bias-Variance Tradeoff

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

\[
\text{Bias}^2[F] = E_{D_t}[\text{err}](\hat{f}(x))
\]
where the expectation is taken over \( x \) and \( D \). The variance of a model is the variance in its predictions when training on different random datasets. Formally, with \( f(x) = \mathbb{E}_{D,W} f(x) \):

\[
\text{Var}(F) = \mathbb{E}_{x,D,W} \left( \hat{f}(x) - f(x) \right)^2.
\]

**Remarks:**
- If the bias is large, we know that our model \( F \) cannot approximate \( f \) well and we should consider a more expressive model, i.e., a more general family of functions \( F \). If on the other hand \( f \in F \), 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 \( w \) 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 high bias. The 6th order model (middle) has low bias and low variance. The 11th order model (right) is too complex and has a high variance. The 6th 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:

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

**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.

**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 allow us to fit \( f \) well. This is what Lasso does.

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

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

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

\[
\text{MSE}(\hat{f}) = \mathbb{E}_{x,D,W} \left( \hat{f}(x) - f(x) \right)^2.
\]

**Remarks:**
- 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.
Remarks:

- The regularization parameter $\lambda$ 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.](image)

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 $y$ 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 $|w_j|$ with $w_j^2$.
- 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 $L(\hat{f}(w), D)$, repeatedly perform the update

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

simultaneously for $j = 0, 1, \ldots, d$, where hyperparameter $\alpha$ 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, $\nabla L(\hat{f}(w), D)$ 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 converge to a local minimum rather than finding the global minimum. The minimum reached depends on the initial starting point and the learning rate $\alpha$. 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 loss function. However on the (right) gradient descent gets stuck in a local minimum.](image)

Remarks:

- The learning rate $\alpha$ 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 $w$, 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.
5.7 Logistic Regression

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

\[ w_j := w_j + \alpha (X_j^T (y - Xw)) \]

\[ = w_j + 2\alpha \sum_{x \in D} (y - w^T x) x_j \]

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 \( f : x \rightarrow y \), 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 \( w^T x \), but now we classify all samples with \( w^T x > 0 \) as 1, all samples with \( w^T x < 0 \) 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 \( x \) is in the direction of vector \( w \), the higher the probability that sample \( x \) belongs to class 1.

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

\[ \hat{f}(x) = \psi(w^T x) = \frac{1}{1 + e^{-w^T x}} \]

where \( \psi \) is called the logistic or sigmoid function.

Remarks:

- The logistic function \( \psi \) squashes inputs from \([-\infty, \infty]\) into the range \([0, 1]\). Other functions, such as the probit link function, could be used instead.
- \( \hat{f}(x) \) can be seen as an estimate of the probability that \( y = 1 \). Therefore, we usually classify \( x \) as positive (1) if \( \hat{f}(x) > 0.5 \) and as negative (0) if \( \hat{f}(x) < 0.5 \).
- This defines a decision boundary \( \psi(w^T x) = 0.5 \), or \( w^T x = 0 \), 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 \( w^T \)? We could minimize the linear regression squared error loss (Definition 5.3). However, this would give predictions beyond \([0, 1]\).
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),**

\[
L(w) = \prod_{(x,y) \in D} P(y | x, w) = \prod_{(x,y) \in D} \psi(w^T x) \prod_{(x,y) \in D} (1 - \psi(w^T x))
\]

**Remarks:**

- \(L(w)\) is the probability of observing the vector of outputs \(y\) given input data \(X\) and parameters \(w\). Intuitively, we want to choose \(w\) 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 \(y\) are independent and identically distributed Bernoulli with parameters \(p = \hat{f}(x) = \psi(w^T x)\), maximizing \(L(w)\) is equivalent to minimizing the following loss function:

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

**Proof.**

\[
L(w) = \prod_{(x,y) \in D} \psi(w^T x) \prod_{(x,y) \in D} (1 - \psi(w^T x))
\]

\[
= \prod_{(x,y) \in D} \psi(w^T x)^y (1 - \psi(w^T x))^{1-y}
\]

\[
= \prod_{(x,y) \in D} \hat{f}(x)^y (1 - \hat{f}(x))^{1-y}
\]

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 \(w\):

\[
\log L(w) = \sum_{(x,y) \in D} [y \log(\hat{f}(x)) + (1 - y) \log(1 - \hat{f}(x))]
\]

Therefore \(\arg\max_w L(w) = \arg\min_w L(\hat{f}, D)\). 

5.7. LOGISTIC REGRESSION

**Remarks:**

- The scaling factor \(n\) does not change the optimum \(w\), 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

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

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

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

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

\[
\frac{\partial L(\hat{f}, x)}{\partial w_j} = -\frac{1}{\psi(w^T x)^y} \left[ y \log(\psi(w^T x)) + (1 - y) \log(1 - \psi(w^T x)) \right]
\]

\[
= - \frac{y}{\psi(w^T x)^y} \left[ 1 - \psi(w^T x) \right] \frac{\partial}{\partial w_j} \psi(w^T x) \quad \text{chain rule}
\]

\[
= - \frac{y}{\psi(w^T x)^y} \left[ 1 - \psi(w^T x) \right] x_j \cdot \psi(w^T x) \quad \text{chain rule}
\]

\[
= - \frac{y - \psi(w^T x)}{\psi(w^T x)^y} x_j \frac{\partial}{\partial w_j} \psi(w^T x) \quad \text{chain rule}
\]

\[
= \frac{\hat{f}(x) - y}{\hat{f}(x)} x_j
\]
And since differentiation and finite summation are interchangeable, i.e.,
\[
\frac{d}{dx} \sum y(x) = \sum \frac{d}{dx} y(x)
\]
we get the gradient for the total loss, \( L(f, D) \), as:
\[
\frac{\partial L(f, D)}{\partial w_j} = \frac{1}{n} \sum_{(x, y) \in D} \left[ f(x) - y \right] \cdot x_j
\]

Lemma 5.35 (Softmax Regression Loss Function). Assuming the \( y \)'s in \( Y \) are independent and identically distributed categorical random variables with parameters \( p_i = \sigma(w^T_i x) \) for \( i = 1, \ldots, k \), then maximizing the likelihood \( C(w) \) is equivalent to minimizing the following loss function:
\[
L(f, D) = -\frac{1}{n} \sum_{x \in D} \sum_{i=1}^k \left[ \mathbb{1}[y = i] \log(f(x)_i) \right]
\]
where \( w \) is the set of all weights \( \{w_i\}_{i=1}^k \) and \( \mathbb{1} \) 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

Definition 5.34 (Softmax Regression). Softmax regression with \( k \geq 2 \) classes chooses the following functional form for \( f \):
\[
f(x) = \sigma(w^T_1 x, w^T_2 x, \ldots, w^T_k x), \quad \sigma = \frac{\exp(w^T_i x)}{\sum_{j=1}^k \exp(w^T_j x)} \quad \text{for } i = 1, \ldots, k
\]
where \( \sigma \) is called the softmax function.

Remarks:
- \( f(x) \) is a vector of length \( k \), with elements summing up to 1. It can be seen as a vector of probabilities, with \( f(x)_i = \sigma(x) \).
- A different set of linear weights, \( w^T_i \), 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 \( w = w_2 - w_1 \).
- We also get the same form for the loss function:

def predict(self, x):
    # self is current node, initially root
    if self.is_leaf:
        return self.value
    if x[feature] <= self.threshold:
        return self.left.predict(x)
    else:
        return 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.
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 \( D_v \subseteq D \).

Remarks:
- The weights ensure that all training samples have equal contribution.
- 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.
- 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 \( \text{NP}-\text{hard} \).
- One big advantage of decision trees is their simple interpretation. They suffer from high variance and overfit easily. A common technique to improve decision trees is to use many trees in an ensemble.

Definition 5.37 (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:

\[
D_v = \{ x \mid x_i \leq t \}, \quad D_{v,l} = \{ x \mid x_i > t \}
\]

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:

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

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:
- In expectation, every \( D_v \) 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 $f_b$ 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.

<table>
<thead>
<tr>
<th>True label</th>
<th>Predicted label</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p'$</td>
<td>False Positive</td>
</tr>
<tr>
<td>True Positive</td>
<td>True Negative</td>
</tr>
<tr>
<td>$n'$</td>
<td>False Negative</td>
</tr>
<tr>
<td>True Negative</td>
<td>True Positive</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Metric</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>$\frac{TP + TN}{P + N}$</td>
</tr>
<tr>
<td>Positive Predictive Value (Precision)</td>
<td>$\frac{TP}{TP + FP}$</td>
</tr>
<tr>
<td>True Positive Rate (Recall)</td>
<td>$\frac{TP}{TP + FN}$</td>
</tr>
<tr>
<td>False Positive Rate</td>
<td>$\frac{FP}{FP + TN}$</td>
</tr>
<tr>
<td>F1 score</td>
<td>$\frac{2 \cdot PPV \cdot TPR}{PPV + TPR}$</td>
</tr>
</tbody>
</table>

**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 those 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 Characteristic (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 $\hat{f}(x)$, 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:**
- $\tau_{pr}$ is the threshold for which if $\hat{f}(x) \geq \tau_{pr}$ 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 $x_N$ and a random positive sample $x_P$, what is the probability that $\hat{p}(x_P) > \hat{p}(x_N)$, i.e., the classifier will assign higher probability to the random positive sample than to the negative sample.
Chapter Notes

Ordinary least squares (OLS) was one of the first statistical methods to be developed, circa 1800 [5]. There is still controversy over who first applied it, Gauss or Legendre. Subsequently, weighted least squares, minimization of other norms (e.g., L1), multivariate minimization, regularization (e.g., Ridge, Lasso) and many other tools were developed. The term “ordinary” was added to least squares only after many alternative methods were suggested.

Gradient descent is generally attributed to Cauchy, who first suggested it in 1847 [4]. Hadamard independently proposed a similar method in 1907 [3]. Its convergence properties for non-linear optimization problems were first studied by Haskell Curry in 1944 [2], with the method becoming increasingly well-studied and used in the following decades, also often called steepest descent.

The logistic function was developed as a model of population growth and named “logistic” by Pierre François Verhulst in the 1830s and 1840s [1]. The logistic model was likely first used as an alternative to the probit model by Edwin Bidwell Wilson and Jane Worcester in 1943 [6]. The probit model is similar to logistic regression, but models the output as the cumulative distribution of a normal distribution centered around $\omega^T x$. The logit model was initially dismissed as inferior to the probit model, but gradually achieved an equal footing before surpassing it. Its popularity is credited to its computational simplicity, mathematical properties, and generality, allowing its use in varied fields.

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

Bibliography