Machine Learning I

Chapter 05 - Linear Models

Prof. Dr. Sandra Eisenreich

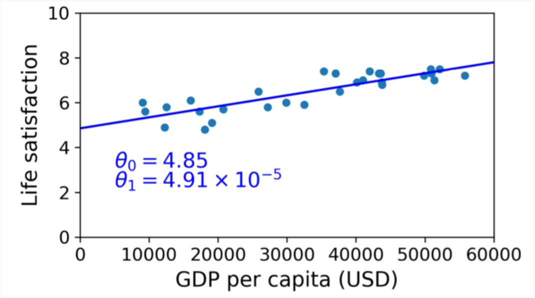
WS 2024/25

Hochschule Landshut

Linear Regression

Overview

- task: **regression**
- data: labeled, **supervised**; Note: Scaling of data speeds up the optimization process
- model: **parametric**
- complexity: simple model, quick for not too many features (see later)
- focus: **explainable, not very flexible**



When to use:

- Only works if data have a linear structure
- good first thing to try for simple predictions, works with only a few instances

Notation

- $n =$ number of features
- $m =$ number of instances in the training set
- $x^{(1)}, \dots x^{(m)} \in \mathbb{R}^m$ are the training instances
- $y^{(1)}, \dots y^{(m)}$ the corresponding labels ($\in \mathbb{R}$ for univariate regression, $\in \mathbb{R}^d$ for multivariate regression)
- $\hat{y}^{(1)}, \dots \hat{y}^{(m)}$ denote our model's predictions ($\in \mathbb{R}$ or $\in \mathbb{R}^d$)
- $\text{res}^{(i)} = \hat{y}^{(i)} - y^{(i)}$ the prediction error for input $x^{(i)}$. ($\in \mathbb{R}$ or $\in \mathbb{R}^d$).
- One combines all input instances $x^{(i)}$ as rows in a data matrix denoted $\mathbf{X}$ for more efficient training.

Recall

1. Prediction function: Pick a "suitable" function h with variable parameters.

2. Loss/gain/objective function: a measure describing how well model h fits data.

3. Training: Pick the "correct" values for the parameters (optimization) (such that predictions from 1. generate the minimal loss from 2.)

ML model $\simeq$ 1. + 2.

Univariate Linear Regression

Univariate Linear Regression is the model defined by:

- Prediction function:

$$\begin{aligned} h_\theta(\mathbf{x}) &= \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots \theta_n x_n \\ &= b + \mathbf{w}^T\mathbf{x}, \text{ for } b = \theta_0 \text{ and } \mathbf{w} = (\theta_1, \dots, \theta_n)^T \\ &= \boldsymbol{\theta}^T\tilde{\mathbf{x}}, \text{ for } \tilde{\mathbf{x}} = (1, x_1, \dots, x_n)^T \text{ and } \boldsymbol{\theta} = (\theta_0, \theta_1, \dots, \theta_n)^T. \end{aligned}$$

  b is called bias and $\mathbf{w}$ the vector of weights.
- Loss function: $\mathrm{MSE}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \mathrm{res}_{\boldsymbol{\theta}}^{(i)2}$

Note:

- In our example, b is the y-intercept.
- The w_i are called weights because they determine the influence of the individual features; if the data is standardized and $b_i >> b_j$ for all other j, then feature i has the biggest impact.

Multivariate Regression

Multivariate Linear Regression is the model defined by:

- Prediction functions (one for each output dimension):

$$\begin{array}{rcl} h_{\theta,1}(\mathbf{x}) & = & b_0 + w_{11}x_1 + w_{12}x_2 + \ldots w_{1n}x_n \\ h_{\theta,2}(\mathbf{x}) & = & b_1 + w_{21}x_1 + w_{22}x_2 + \ldots w_{2n}x_n \\ \vdots & \vdots & \vdots \\ h_{\theta,d}(\mathbf{x}) & = & b_d + w_{1d}x_1 + w_{1d}x_2 + \ldots w_{1d}x_n \\ \text{or in other words } h_\theta(\mathbf{x}) & = & \mathbf{b} + \mathbf{W} \cdot \mathbf{x}, \end{array}$$

$$\text{where } \mathbf{b} := \begin{pmatrix} b_1 \\ \vdots \\ b_d \end{pmatrix} \text{ and } \mathbf{W} := \begin{pmatrix} w_{11} & w_{12} & \ldots & w_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d1} & w_{d2} & \ldots & w_{dn} \end{pmatrix}$$

  are the bias vector and the weight matrix.

- Loss function: $\mathrm{MSE}(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\|\mathrm{res}_{\boldsymbol{\theta}}^{(i)}\|_2$

Matrix formulation for univariate regression

If we process the data matrix $\mathbf{X}$ all at once to obtain a vector of predictions $\hat{\mathbf{y}} = (\widehat{\mathbf{y}}^{(1)}, \dots, \widehat{\mathbf{y}}^{(m)})$ of the true label vector $\mathbf{y} = (y^{(1)}, \dots, y^{(m)})$, we can rewrite the formulae for linear regression in a vectorized way: First, extend the matrix $\mathbf{X}$ by a column of 1s on the left to obtain the matrix $\tilde{\mathbf{X}}$ which has the vectors $\tilde{x}^{(i)}$ as rows, and $\boldsymbol{\theta} = (\theta_0, \dots, \theta_n)$.

Question: How can you formulate the predictions and MSE-loss in terms of these vectors and matrices?

Result:

- vector of predictions: $\hat{\mathbf{y}} = \tilde{\mathbf{X}} \cdot \theta$
- $\mathrm{MSE}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{m} \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2 = \frac{1}{m} (\hat{\mathbf{y}} - \mathbf{y})^T (\hat{\mathbf{y}} - \mathbf{y}) = \frac{1}{m} (\tilde{\mathbf{X}}\theta - \mathbf{y})^T (\tilde{\mathbf{X}}\theta - \mathbf{y})$

Optimizing the loss function

Recall from “Optimierung”:

A function is called convex if the line joining any two points on the curve never crosses the curve. A convex function only has one global minimum.

Fortunately, one can show:

Theorem: The MSE cost function for Linear Regression is convex, so there is a unique solution for the parameters in Linear Regression.

Finding the correct solution

For convex functions, there are two ways of finding the optimum:

- compute a closed solution = an actual mathematical formula for the solution, derived from "Gradient = 0" (i.e. all partial derivatives = 0, see below).
- use an optimization algorithm (see "Optimierung") like gradient descent (will be repeated later on)

Closed solution: If $f(\theta_0, \dots, \theta_n)$ is a convex real-valued function, it has a unique minimum at point $(\hat{\theta}_0, \dots, \hat{\theta}_n)$ if

$$\nabla_{\boldsymbol{\theta}} f(\hat{\theta}_0, \dots, \hat{\theta}_n) = 0.$$

Problem: Closed solutions can only be mathematically derived for very simple loss functions! Fortunately, it works for Linear Regression.

Closed solution for Univariate Linear Regression

In our case, we have the convex real-valued function

$$\mathrm{MSE}(\boldsymbol{\theta}) = \frac{1}{m}(\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y})^T(\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y}).$$

Minimizing this does not depend on constant factors, i.e. we can drop the $\frac{1}{m}$ and multiply by $\frac{1}{2}$.

$$\mathrm{minimize}_{\boldsymbol{\theta}} \frac{1}{2}(\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y})^T(\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y})$$

One can compute (see next slide): $\nabla SS_{\text{res}}(\boldsymbol{\theta}) = \tilde{\mathbf{X}}^T\tilde{\mathbf{X}}\boldsymbol{\theta} - \tilde{\mathbf{X}}^T\mathbf{y}$

So if we set $\nabla SS_{\text{res}}(\widehat{\boldsymbol{\theta}}) = 0$, we get:

Normal Equation

The unique parameter vector $\widehat{\boldsymbol{\theta}} = \text{argmin}_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta})$ minimizing MSE satisfies

$$\tilde{\mathbf{X}}^T \tilde{\mathbf{X}} \widehat{\boldsymbol{\theta}} := \tilde{\mathbf{X}}^T \mathbf{y} \quad (*)$$

and if the matrix $\tilde{\mathbf{X}}^T \tilde{\mathbf{X}}$ is invertible:

$$\widehat{\boldsymbol{\theta}} := \left(\tilde{\mathbf{X}}^T \tilde{\mathbf{X}}\right)^{-1} \tilde{\mathbf{X}}^T \mathbf{y}$$

The formula $*$ is called the Normal Equation and the solution $\hat{\boldsymbol{\theta}}$ is called the ordinary least squares solution (OLS solution).

Computation

Proof.

$$\begin{aligned}
\nabla_\theta SS_{\text{res}}(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \left((\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y})^T(\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y})\right) \\
&= \nabla_{\boldsymbol{\theta}} \left((\tilde{\mathbf{X}}\boldsymbol{\theta})^T\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y}^T\tilde{\mathbf{X}}\boldsymbol{\theta} - (\tilde{\mathbf{X}}\boldsymbol{\theta})^T\mathbf{y} - \mathbf{y}^T\mathbf{y}\right) \\
&= \nabla_{\boldsymbol{\theta}} \left(\boldsymbol{\theta}^T\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}\boldsymbol{\theta} - 2\cdot\mathbf{y}^T\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y}^T\mathbf{y}\right),
\end{aligned}$$

where for the last equation we used the fact that $(AB)^T = B^TA^T$.

We compute the gradient of the indivdiual terms by using two mathematical facts:

(i) If $\mathbf{a}, \mathbf{x}$ are a vectors, then $\nabla_x(\mathbf{a}^T\mathbf{x}) = \mathbf{a}$

(ii) If A is a matrix and $\mathbf{x}$ a vector, then $\nabla_{\mathbf{x}}(\mathbf{x}^TA\mathbf{x}) = (A + A^T)\mathbf{x}$

Task: Compute the gradient this way!

Solution

- First, we can drop the term $\mathbf{y}^T\mathbf{y}$ that does not depend on $\boldsymbol{\theta}$,
- Second, the term $\mathbf{y}^T\tilde{\mathbf{X}}\boldsymbol{\theta}$ is of form (i), so its gradient is $(\mathbf{y}^T\tilde{\mathbf{X}})^T = \tilde{\mathbf{X}}^T\mathbf{y}$.
- Last but not least, we use (ii) to get:

$$\nabla_{\boldsymbol{\theta}}\boldsymbol{\theta}^T\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}\boldsymbol{\theta} \;\; = \;\; \left(\tilde{\mathbf{X}}^T\tilde{\mathbf{X}} + (\tilde{\mathbf{X}}^T\tilde{\mathbf{X}})^T\right)\cdot\boldsymbol{\theta} = 2\cdot\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}\boldsymbol{\theta}.$$

Combining all of this, we really get:

$$\nabla_{\boldsymbol{\theta}} SS_{\text{res}}(\boldsymbol{\theta}) \;\; = \;\; \tilde{\mathbf{X}}^T\tilde{\mathbf{X}}\boldsymbol{\theta} + \tilde{\mathbf{X}}^T\mathbf{y}$$

Problem with the closed solution

Note that in the closed solution we have to compute and invert the matrix $A = \tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$.

- $\tilde{\mathbf{X}}$ is an $m \times (n+1)$-matrix, $\tilde{\mathbf{X}}^T$ is an $(n+1) \times m$-matrix,so computing the $(n+1) \times (n+1)$-matrix A is $O((n+1)^2 m) = O(n^2 m)$
- A could be (close to) non-invertible, in which case inverting it would be a numerically very bad idea (wrong result). Even if it is invertible, it is computationally expensive: $O(n^3)$.

There are numerical "work-arounds" (Cholesky) for inverting the matrix of order $O(n^2)$... but still it is much faster just to use methods of optimization.

Complexity

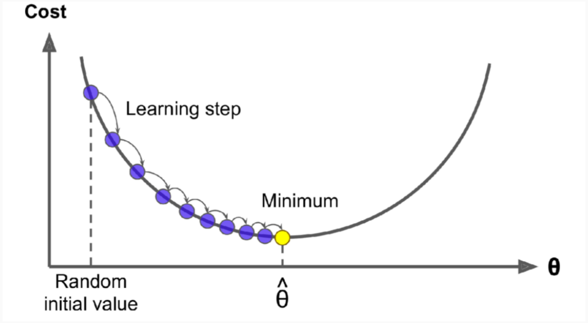
The complexity of...

- computing the OLS solution, i.e. training Linear Regression, with the closed solution, is $O(n^2 \cdot m + n^3)$, with Cholesky it's $O(n^2 \cdot m)$.
- of making the prediciton $\hat{\mathbf{y}} = \boldsymbol{\theta}^T \cdot \tilde{\mathbf{x}}$ for a new n-dimensional instance $\mathbf{x}$ is $n + 1$, i.e. inference ($=$ making a prediction on one instance) is $O(n)$.

Gradient Descent

Gradient Descent is an optimization algorithm capable of minimizing cost functions (at least locally) by changing parameters iteratively in small steps in the direction of the steepest slope.

Idea of Gradient Descent: Look at where the function goes downhill with the steepest slope (this is the negative gradient), and then make a step in that direction. This step is called learning step. Repeat making learning steps until you can't go downhill any more. Then you are at the minimum.



For an animation, look at the following link: Gradient Descent Video

Gradient descent algorithm

- initialize the parameters as some random value θ^0.
- compute the gradient of the cost function $\nabla \text{MSE}(\theta^0)$ at that point, and go one step in the opposite (negative) direction of this gradient

  $$\theta^1 = \theta^0 - \eta \nabla \text{MSE}(\theta^0),$$

  where η is called the step size/learning rate which controls how far you go.
- compute the gradient $\nabla \text{MSE}(\theta^1)$ at the new parameters, and once again go a step in the direction of the minimum

  $$\theta^2 = \theta^1 - \eta \nabla \text{MSE}(\theta^1)$$

- and so on until you get to the minimum, where the gradient is zero.

Problems with Gradient Descent

- If the loss function is not convex (it usually isn't), we might get stuck in a local minimum.
- quite slow: at each step, the algorithm has to compute the full gradient

$$\nabla_{\boldsymbol{\theta}} MSE \simeq \tilde{\mathbf{X}}^T \tilde{\mathbf{X}} \boldsymbol{\theta} + \tilde{\mathbf{X}}^T \mathbf{y}$$

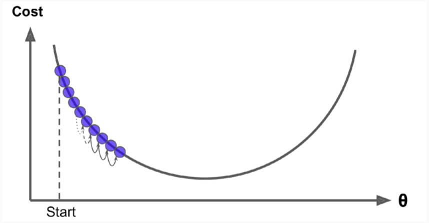
  where $\tilde{\mathbf{X}}$ is the $m \times (n+1)$-matrix with ALL m training instances. Multiplying an $(n+1) \times m$-matrix with an $m \times (n+1)$-matrix is in $O(n^2 \cdot m)$, which is the dominating factor.

The complexity of finding the OLS solution, i.e. training Linear Regression, with full-batch Gradient Descent, is $O(n^2 \cdot m \cdot T)$, where T is the number of descent steps.

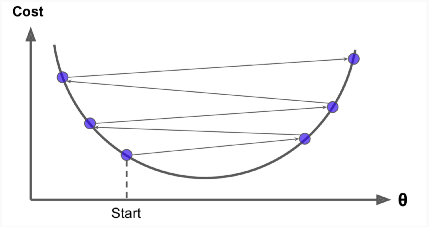
How to set the learning rate

For the optimization algorithm, it is very important how you choose the size of the steps = learning rate. The learning rate is a hyperparameter of the optimization you need to set.

If the learning rate is too small, then the algorithm will converge very slowly:



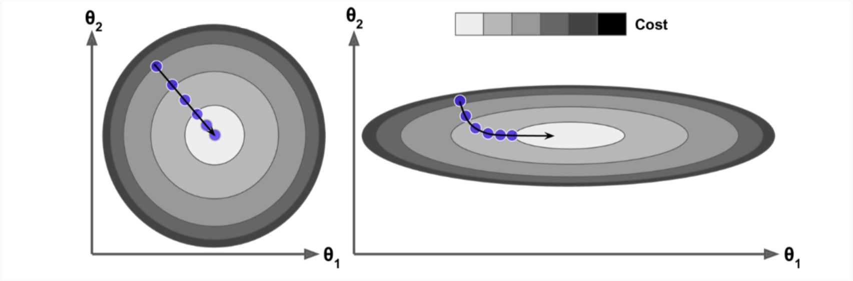
However, if the learning rate is too big, it might zig-zag around the minimum.



To find a good learning rate, you can use grid search with limited number of iterations.

Feature scaling

In order to make it converge faster, it is important to ensure that all features have a similar scale.



Stochastic Gradient Descent (SGD)

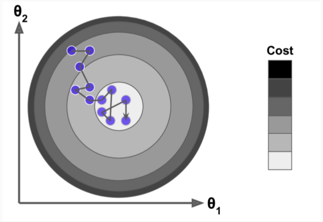
There is a different version of Gradient Descent called Stochastic Gradient Descent (SGD): At each Descent step, do not compute the gradient of the entire

$$MSE = \frac{1}{m}\sum_{i=1}^{m} res^{(j)2},$$

but only compute the gradient for one single random instance $\mathbf{x}$ (with label y), or a so-called mini-batch of a few (K) random instances (this is called mini-batch Gradient Descent): $g_t = 2 \cdot (\theta_t^T \cdot \mathbf{x} - y) \cdot \mathbf{x}$. This way, you save the factor of m in each learning step, i.e. the complexity of a step is $O(n^2)$.

Disadvantage: due to its stochastic nature, "bounces around". Does not reach the optimum, but gets close.

Solution: start with a big learning rate, make it smaller over time, so the descent "rolls in softly".



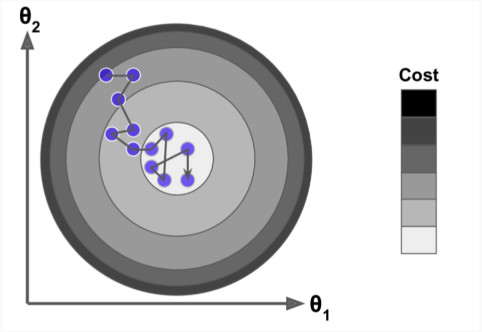
A varying learning rate over the course of training is called a **learning rate schedule**.

Pros and Cons of SGD

Disadvantage: “bouncy”, does not reach optimum.

Advantages:

- can “bounce” out of local minima and find better ones.
- much faster,
- only one random instance in memory at a time (online learning= learning from one instance at a time)



Complexity of Linear Regression

The complexity of training a linear regression model (i.e. computing the OLS solution)...

- by computing the closed solution is $O(n^3 + n^2 \cdot m)$ when inverting the matrix, $O(n^2 \cdot m)$ with Cholesky.
- with gradient descent is $O(n^2 \cdot m \cdot T)$ for T steps (often only a few so same as Cholesky)
- with stochastic gradient descent is $O(n^2 \cdot T)$ for T steps

For large number of features, this gets computationally expensive.

The inference (=prediction) complexity is $O(n)$.

Performing Linear Regression with Scikit-Learn

To train the linear regression model with Full-batch Gradient Descent, you can use Scikit-Learn's `Linear_Model`:

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
```

To train the linear regression model with Stochastic Gradient Descent, you can use Scikit-Learn's `SGDRegressor` class (`ravel` makes a column vector a row vector):

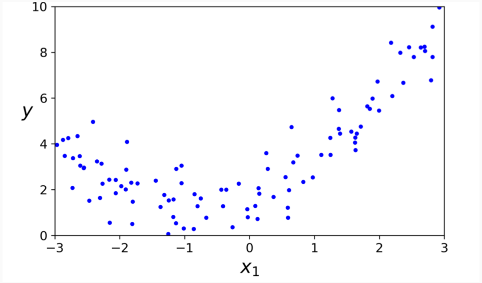
```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

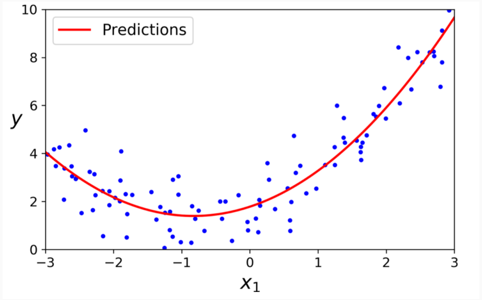
For both methods, to get a prediction for new data `X_new`, use `.predict(X_new)`. To get the bias term b, use `.intercept_`, to get the weights w_i, use `.coef_`.

Polynomial Regression

Motivation

What if your data is more complex than a straight line? Surprisingly, you can still use a linear model to fit nonlinear data by adding powers of each feature as a new feature, and training a model on this extended set of features. This is called Polynomial Regression.





When to use: when your data is not linear, but you want a simple model; take care: can grow really slow for high degree polynomials!

From Linear to Polynomial Regression

The idea of polynomial regression is to transform the n features $x = (x_1, \dots, x_n)^T$ into

$$(n+d)!/d!n!$$

features of all combinations of degree d

$$x = (x_1, \dots, x_n, x_1^2, x_1x_2, x_1x_3, \dots, x_n^2, x_1^3, x_1^2x_2, \dots, x_n^d)$$

Attention: Beware of the combinatorial explosion of the number of features!

Question: What is the complexity of one step of training a Polynomial regression model with

- full-batch GD
- SGD?

Answer:

- $O(((n+d)!/d!n!)^2 m)$ which grows exponentially fast with $d =$ degree
- $O(((n+d)!/d!n!)^2)$

Polynomial Regression with Scikit-Learn

To do Polynomial Regression with Scikit-Learn, you can use PolynomialFeatures:

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
```

`X_poly` now contains the original feature of X plus the square of this feature. Now you can fit a LinearRegression model to this extended training data to perform quadratic regression.

Regularized Linear Regression

Regularization as a method to reduce overfitting

One way to reduce overfitting is to constrain the weights of the model by adding a term to the cost function that penalizes their size. We will introduce three different ways to do that:

- Ridge Regression,
- Lasso Regression and
- Elastic Net.

Note: It is important to scale the data (eg with StandardScaler) before training regularized models, because most are sensitive to the scale of input features.

Ridge Regression

Ridge Regression is a ℓ_2-regularized Linear Regression model, i.e. the loss function is

$$L_{\text{ridge}}(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=0}^{n} \theta_i^2,$$

Ridge Regression yields smaller values for the parameters.

Ridge Regression has higher bias and lower variance than Ordinary Linear Regression. The bigger the regularization constant α, the bigger the bias and lower the variance.

Closed solution for ridge regression

Due to the added regularization term, there is a different closed solution for ridge regression:

Theorem: The optimal parameters for ridge regression are given by the closed-form solution

$$\left(\mathbf{X}^T\mathbf{X} + \alpha A\right)^{-1}\mathbf{X}^T\mathbf{y},$$

where A is the matrix

$$A = \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & \mathbb{1}_n \end{array}\right)$$

But again, in most cases using optimization for finding the parameters that minimize the cost function is quicker.

Ridge Regression with Scikit-Learn

Training a ridge regression model with the closed solution:

```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1, solver="cholesky")
ridge_reg.fit(X, y)
```

or with SGD:

```
sgd_reg = SGDRegressor(penalty="l2")
sgd_reg.fit(X, y.ravel())
```

As always, you can predict the label of new data X_{new} with `.predict(X_new)`, and get the bias term b with `.intercept_`, and the weights w_i with `.coef_`.

Lasso Regression

Lasso Regression ("Least Absolute Shrinkage and Selection Operator Regression") is ℓ_1-regularized Linear Regression, i.e. the loss function is:

$$L_{\text{lasso}}(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=0}^{n} |\theta_i|.$$

The ℓ_1-regularization sets weights θ_i of irrelevant features to zero, i.e. it works as a feature selector and the trained model is sparse.

Lasso Regression also has higher bias and lower variance than Ordinary Linear Regression. The bigger the regularization constant α, the bigger the bias and lower the variance.

Pros and Cons

Advantages:

- Allows dropping irrelevant features and retraining a smaller model $\rightarrow$ faster
- Can be used as Data Preprocessing step for feature selection.

Disadvantages:

- The Lasso cost function is not differentiable, so there is no closed solution, and you can't use gradient descent, but only a variant.
- During training, Lasso bounces around the minimum even more than SGD.

Lasso Regression and Collinearity

Multicollinearity (also collinearity) is a phenomenon in which one feature in a regression model can be perfectly predicted from the others.

Question: What will a Lasso Regression model do when you have collinearity in the features

- during training, or
- if you train it several times?

Answer:

- during training, it can be indecisive as to which features it should pick, and jump around erratically.
- It might pick a different feature each time for the prediction and set the others to 0. This behaviour is not robust behaviour.

Lasso Regression with Scikit-Learn

To train a Lasso model with Gradient Descent:

```
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X, y)
```

To use Lasso with Stochastic Gradient Descent: use the SGDRegressor class with `SGDRegressor(penalty="l1")` (since the sum of absolute values of the parameters is the ℓ_1-norm of the parameter vector).

Again, use `.predict(X_new)` for inference.

Elastic Net

If you are not sure which regularization is better, or think ℓ_2 is too weak and ℓ_1 is too strong, you can use a combination:

Elastic Net Elastic Net is s the ML model with the same prediction function as Linear Regression, but the cost function changed to

$$L_{\text{elastic}}(\boldsymbol{\theta}) \quad = \quad \text{MSE}(\boldsymbol{\theta}) + \alpha \left(r \sum_{i=0}^{n} |\theta_i| + \frac{1-r}{2} \sum_{i=0}^{n} \theta_i^2 \right)$$

for constants $\alpha > 0, r \in [0,1]$. (A combination of ℓ_1 and ℓ_2-regularization). The hyper-parameter r determines which regularization has more weight; $r = 0$: Ridge Regression, $r = 1$: Lasso Regression.

Scikit-Learn and When to use which variant of Linear Regression?

Elastic Net with Scikit-Learn:

```
from sklearn.linear_model import ElasticNet
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_net.fit(X, y)
```

When to use what?

- almost always regularization is better than without.
- Ridge is a good default,
- If you suspect only a few features are useful, Lasso or ElasticNet are better.
- If several features are strongly correlated/multicollinear, Lasso might behave erratically.
- ElasticNet is better trainable than Lasso and less erratic with collinearity.

Logistic Regression for binary classification

Overview

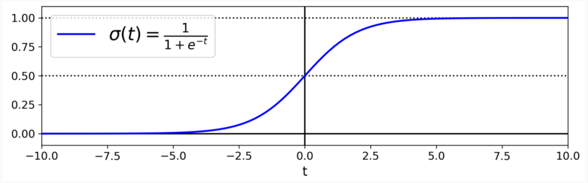
- task: binary classification
- rest: same as linear regression

One can translate any classification task into a regression task by predicting the probability vector (= a real-valued vector), or in case of binary classification, the probability of the positive class (the prob. of the negative class is simply 1- this number).

Using univariate linear regression for binary classification is called Logistic Regression.

Using Regression for Classification

Regression predicts values in $\mathbb{R}$, but we want a probability between 0 and 1 $\rightarrow$ need a function mapping $\mathbb{R}$ to $[0, 1]$: the sigmoid function or logistic function



$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

Note: You apply σ to the output $\boldsymbol{\theta}^T\tilde{\mathbf{x}}$ (called the logit or pre-activation in this case) of the Linear Regression Model, so it

- 1 if the Linear Regression model output $\boldsymbol{\theta}^T\tilde{\mathbf{x}}$ is ≥ 0, and
- 0 if $\boldsymbol{\theta}^T\tilde{\mathbf{x}} < 0$.

Logistic Regression

Logistic Regression is the ML model with

- prediction function (predicting probability of the positive class)

$$\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma(\boldsymbol{\theta}^T\tilde{\mathbf{x}}) = \sigma(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots \theta_n x_n),$$

  where σ is the sigmoid function and $\tilde{\mathbf{x}} = (1, \mathbf{x})$. The predicted class is then given by

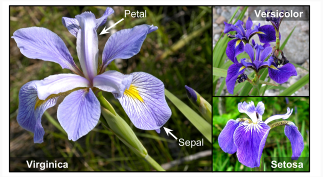
$$\widehat{\mathbf{y}} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

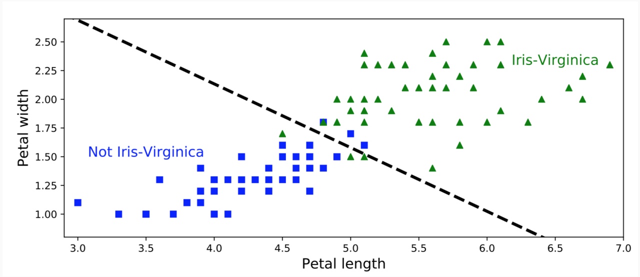
- loss function: the cross entropy loss (log loss)

$$L_{\mathbb{H}}(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\hat{p}^{(i)} + \left(1 - y^{(i)}\right)\log\left(1 - \hat{p}^{(i)}\right)\right)$$

Decision boundaries

Since the Logistic Regression model predicts 1 if $\boldsymbol{\theta}^T\tilde{\mathbf{x}}$ is positive and 0 if it is negative, we get a decision boundary where $\boldsymbol{\theta}^T\tilde{\mathbf{x}} = 0$:





The decision boundary in Logistic Regression is given by the hyperplane

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots \theta_n x_n = 0.$$

Training and Scikit-Learn

There is no known closed-form solution. $\rightarrow$ use Gradient Descent or SGD.

with Scikit-Learn:

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)
```

- Get learned bias and weights with `.intercept_` und `.coef_`
- mean accuracy: `.score()` applied to the data you want to evaluate the model on

Alternatively, you can use SGD-Optimization (faster!) with:

```
from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(loss='log')
sgd_clf.fit(X,y)
```

Motivation: Multiclass classification

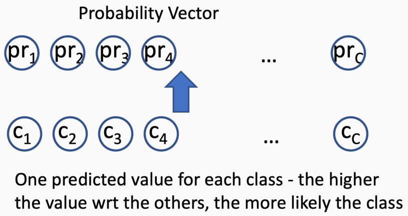
Question: Can you also use Linear Regression for multiclass classification? If so, how?

Answer: Predict the probability vector with multivariate linear regression. BUT: Regression predicts real-valued vectors, but we need the results to be an actuall probability vector, i.e. the values between 0 and 1 and the sum =1.

How? Use a function like σ such that the resulting values

- are between 0 and 1
- and sum up to 1.

$\rightarrow$ Softmax function


Probability Vector
pr_1 pr_2 pr_3 pr_4 ... pr_C
c_1 c_2 c_3 c_4 ... c_C
One predicted value for each class - the higher the value wrt the others, the more likely the class

Softmax function and scores

The Softmax function with C classes is defined as

$$S(s_1(\mathbf{x}), \ldots, s_C(\mathbf{x})) := (S_1, \ldots, S_C)^T = \left(\frac{e^{s_1(\mathbf{x})}}{\sum_{j=1}^C e^{s_j(\mathbf{x})}}, \frac{e^{s_2(\mathbf{x})}}{\sum_{j=1}^C e^{s_j(\mathbf{x})}}, \ldots, \frac{e^{s_C(\mathbf{x})}}{\sum_{j=1}^C e^{s_j(\mathbf{x})}} \right).$$

This function transforms any real-valued vector $(s_1, \ldots, s_C) \in \mathbb{R}^d$ to a vector $(S_1, \ldots, S_C)$ with $S_i \in [0, 1]$ and $\sum_{i=1}^C S_c = 1$.

Task: Verify the two properties above.

Softmax Regression

- The output of the regression model is called the Softmax score

$$s = (s_1, \ldots, s_K)^T, \text{ where } s_c(\mathbf{x}) = \boldsymbol{\theta}^{(c)T}\tilde{\mathbf{x}}$$

- Then a Softmax Regression classifier/Multinomial Logistic Regression model is defined by
  - prediction function (predicting the probability vector of the C classes)

$$\hat{p} = h_\theta(\mathbf{x}) = S(s_1(\mathbf{x}), \ldots, s_C(\mathbf{x})) = S(\boldsymbol{\theta}^{(1)T}\tilde{\mathbf{x}}, \ldots, \boldsymbol{\theta}^{(C)T}\tilde{\mathbf{x}}),$$

    The predicted is then given by most probable class.
  - cost function: the cross entropy loss

$$L_{\mathbb{H}}(\boldsymbol{\Theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{c=1}^{C}\mathbf{y}_c^{(i)}\log\hat{p}_c^{(i)},$$

    where $\mathbf{y}^{(i)}$ is the one-hot encoding of the ground-truth class (only one entry 1).

Softmax Regression with Scikit-Learn

```
from sklearn.linear_model import LogisticRegression
softmax_reg = LogisticRegression(multi_class="multinomial", C=10)
softmax_reg.fit(X, y)
```

- get class predictions for new data via `.predict()`.
- get probability vector predicitons with `.predict_proba()`.

Chapter 05 Summary

Chapter 05 Summary - Overview

Attention: From now on the content is too complex to be able to summarize everything important here. So the summaries from now on are just a quick reminder of some of the most important facts/terminology.

- task Linear Regression/Logistic Regression: **regression/classification**
- **supervised** learning; Note: Scaling of data speeds up the optimization process
- model: **parametric**
- focus: **explainable, not very flexible**
- When to use: when the underlying data have a linear structure

The complexity of training a linear/Logistic regression model (i.e. computing the OLS solution)...

- by computing the closed solution is $O(n^3 + n^2 \cdot m)$ when inverting the matrix, $O(n^2 \cdot m)$ with Cholesky.
- with gradient descent is $O(n^2 \cdot m \cdot T)$ for T steps (often only a few so same as Cholesky)
- with stochastic gradient descent is $O(n^2 \cdot T)$ for T steps

For large number of features, this gets computationally expensive.

Chapter 05 Summary - Linear Regression

- Univariate Linear Regression has
  - prediction function $h_\theta(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots \theta_n x_n = b + \mathbf{w}^T\mathbf{x}$, where $b = \theta_0$ =bias and $\mathbf{w} = (\theta_1, \ldots, \theta_n)^T$ = weights.
  - loss function: $\mathrm{MSE}(\boldsymbol{\theta}) := \mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)}\right)^2 = \frac{1}{m}(\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y})^T(\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y})$, where $\tilde{\mathbf{X}}$ is the data matrix with an additional first column of 1's.
- Training: Since $\mathrm{MSE}(\boldsymbol{\theta})$ is a convex function, it has a unique minimum. This closed solution is given by the vector $\widehat{\boldsymbol{\theta}}$ satisfying the normal equation

$$\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}\widehat{\boldsymbol{\theta}} := \tilde{\mathbf{X}}^T\mathbf{y}.$$

  One could invert the matrix $\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$ to get $\widehat{\boldsymbol{\theta}}$, but that is computationally expensive (takes long) and could lead to numerical issues. Better: Cholesky, or Gradient Descent/SGD
- Training with optimization: gradient descent or (faster but more erratic) stochastic gradient descent (SGD) with the learning rate hyperparameter α.

Chapter 05 Summary - Polynomial Regression

- If the data are not linear, but more complex, one could use Polynomial Regression: transform the n features $x = (x_1, \ldots, x_n)^T$ into

$$(n + d)!/d!n!$$

features of all combinations of degree d:

$$x = (x_1, \ldots, x_n, x_1^2, x_1x_2, x_1x_3, \ldots, x_n^2, x_1^3, x_1^2x_2, \ldots, x_n^d)$$

and apply linear regression model to this feature space. **Attention**: Beware of the combinatorial explosion of the number of features!

Chapter 05 Summary - Ridge Regression

- Ridge Regression is a ℓ_2-regularized Linear Regression model, i.e. the loss function is

$$L_{\text{ridge}}(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=0}^{n} \theta_i^2,$$

- Ridge Regression yields smaller values for the parameters.
- Ridge Regression has higher bias and lower variance than Ordinary Linear Regression. The bigger the regularization constant α, the bigger the bias and lower the variance.

Chapter 05 Summary - Lasso Regression

- Lasso Regression ("Least Absolute Shrinkage and Selection Operator Regression") is ℓ_1-regularized Linear Regression, i.e. the loss function is:

$$L_{\text{lasso}}(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=0}^{n} |\theta_i|.$$

- In Lasso Regression, weights θ_i of irrelevant features are set to zero, i.e. it works as a feature selector and the trained model is sparse.
- It doesn't have a closed solution, so the parameters have to be determine via optimization.
- Pros and Cons:
    - Pros: Can be used as data preprocessing step for feature selection
    - Cons: More instable training, does not work well for multi-collinearity in features (Multicollinearity (also collinearity) is a phenomenon in which one feature in a regression model can be perfectly predicted from the others.)

Chapter 05 Summary - Elastic Net and Overview

- Elastic Net is a mixture of Lasso and Ridge Regression. The cost function is

$$L_{\text{elastic}}(\boldsymbol{\theta}) \quad = \quad \text{MSE}(\boldsymbol{\theta}) + \alpha \left(r \sum_{i=0}^{n} |\theta_i| + \frac{1-r}{2} \sum_{i=0}^{n} \theta_i^2 \right)$$

  for constants $\alpha > 0, r \in [0, 1]$. (A combination of ℓ_1 and ℓ_2-regularization). The hyperparameter r determines which regularization has more weight; $r = 0$: Ridge Regression, $r = 1$: Lasso Regression.

- **When to use what?**
  - almost always regularization is better than without.
  - Ridge is a good default,
  - If you suspect only a few features are useful, Lasso or ElasticNet are better.
  - If several features are strongly correlated/multicollinear, Lasso might behave erratically.
  - ElasticNet is better trainable than Lasso and less erratic with collinearity.

Chapter 05 Summary - Logistic Regression

- Logistic Regression is a linear regression model that is used for classification by translating the real-valued output into a probability vector for all classes. It has
  - prediction function (predicting probability of the positive class)

    $$\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma(\boldsymbol{\theta}^T \tilde{\mathbf{x}}) = \sigma(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots \theta_n x_n),$$

    where σ is the logistic function or sigmoid function

    $$\sigma(t) = \frac{1}{1 + \exp(-t)}.$$

  - loss function: the cross entropy loss (also called log loss)

    $$L_{ce}(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log \hat{p}^{(i)} + \left(1 - y^{(i)}\right) \log \left(1 - \hat{p}^{(i)}\right)\right)$$

  - The logistic regression model prediction is then given by

    $$\widehat{\mathbf{y}} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

  - The decision boundary between the positive and negative class is given by $\boldsymbol{\theta}^T \mathbf{x} = 0$

Chapter 05 Summary - Softmax regression

- The Softmax function with C classes is defined as

$$S(s_1(\mathbf{x}), \dots, s_C(\mathbf{x})) := (S_1, \dots, S_C)^T = \left(\frac{e^{s_1(\mathbf{x})}}{\sum_{j=1}^C e^{s_j(\mathbf{x})}}, \frac{e^{s_2(\mathbf{x})}}{\sum_{j=1}^C e^{s_j(\mathbf{x})}}, \dots, \frac{e^{s_C(\mathbf{x})}}{\sum_{j=1}^C e^{s_j(\mathbf{x})}} \right).$$

- The output of the multivatiate regression model is called the Softmax score

$$s = (s_1, \dots, s_K)^T, \text{ where } s_c(\mathbf{x}) = \boldsymbol{\theta}^{(c)T} \tilde{\mathbf{x}}$$

- Then a Softmax Regression classifier/Multinomial Logistic Regression model is defined by
  - prediction function (predicting the probability vector of the C classes)

$$\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = S(s_1(\mathbf{x}), \dots, s_C(\mathbf{x})) = S(\boldsymbol{\theta}^{(1)T} \tilde{\mathbf{x}}, \dots, \boldsymbol{\theta}^{(C)T} \tilde{\mathbf{x}}),$$

  The predicted is then given by most probable class.
  - cost function: the cross entropy loss

$$L_{\mathbb{H}}(\boldsymbol{\Theta}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{c=1}^{C} \mathbf{y}_c^{(i)} \log \hat{p}_c^{(i)},$$

  where $\mathbf{y}^{(i)}$ is the one-hot encoding of the ground-truth class (only one entry 1).