NMBU

Norwegian University
of Life Sciences

Raw Data Inspection

DAT200 - Applied Machine Learning

Department of Data Science, Faculty of Science of Technology

Lecture Agenda

- Data inspection/Data exploration/Exploratory Data Analysis (EDA)
  - Visualization with `matplotlib` and `seaborn`
- Two basic ML classification algorithms
  - Perceptron
  - Adeline
- Learning as an optimization problem
  - Gradient Descent
  - Feature scaling to improve Gradient Descent

Exploratory Data Analysis

- Understand your data if you want to obtain best possible results with ML models
- Data Visualization -> the most effective way to learn more about your data
- **Absolutely necessary** to do this before training your machine learning models
- NOTE: your compulsory assignment submissions <u>will not be accepted</u> without EDA
- Examples using diabetes-dataset included in `scikit-learn`.

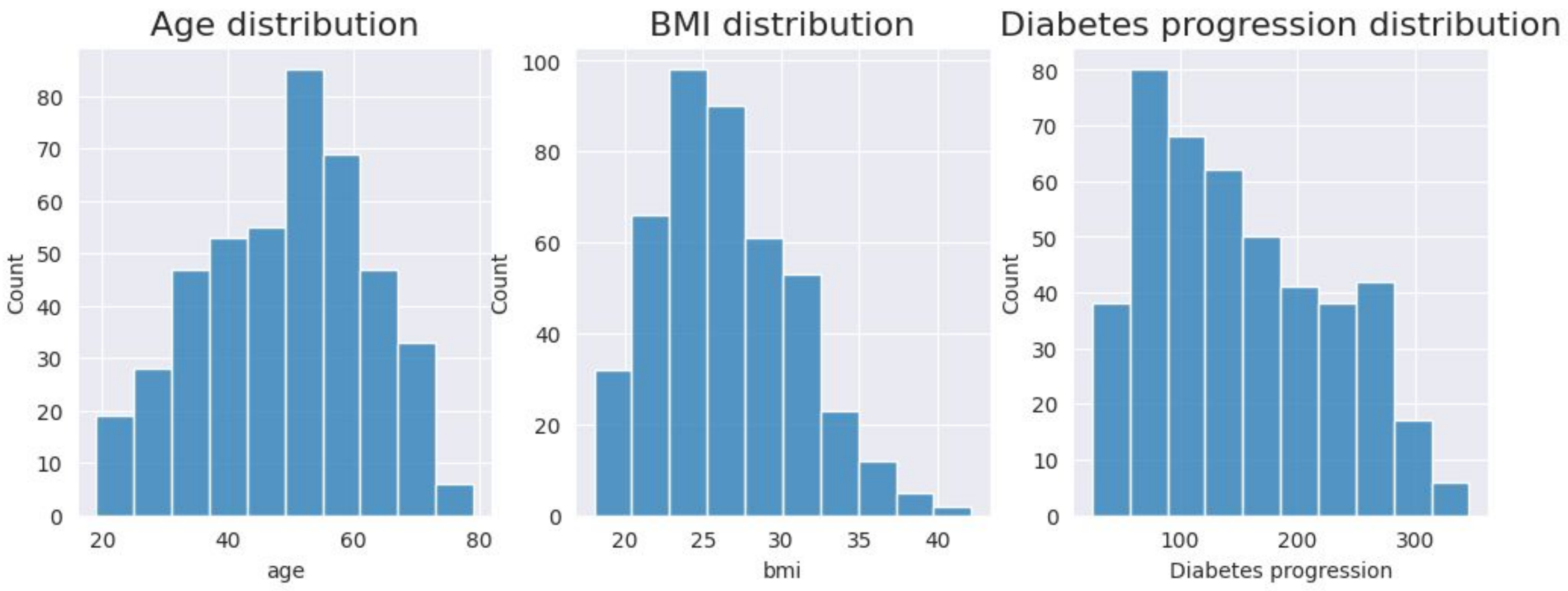
Exploratory Data Analysis: The data

- Dataset has 10 input columns and one target column (Diabetes progression score)
- Input:
  - Age
  - Sex
  - BMI
  - Blood pressure
  - Six different measures of blood cholesterol

	age	sex	bmi	bp	tc	ldl	hdl	tch	ltg	glu	Diabetes progression
0	59	male	32.1	101	157	93.2	38	4.00	4.8598	87	151
1	48	female	21.6	87	183	103.2	70	3.00	3.8918	69	75
2	72	male	30.5	93	156	93.6	41	4.00	4.6728	85	141
3	24	female	25.3	84	198	131.4	40	5.00	4.8903	89	206

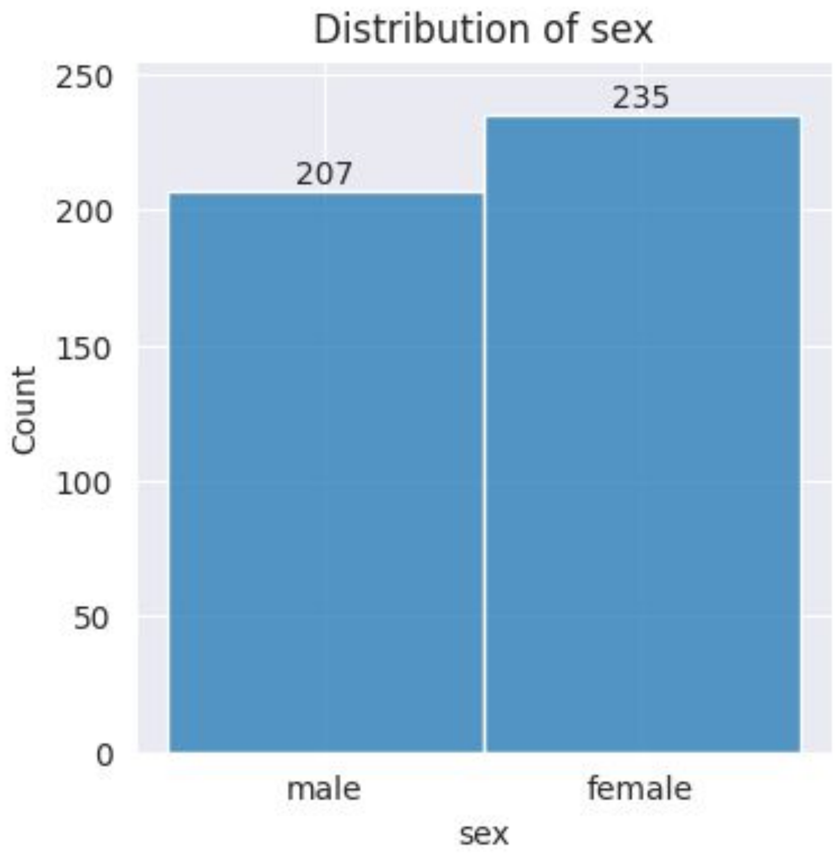
Exploratory Data Analysis: Univariate plots

- Compute descriptive statistics
- Histograms
  - Inspect distribution of each attribute
  - Groups data into bins
  - Count number of observations in each bin



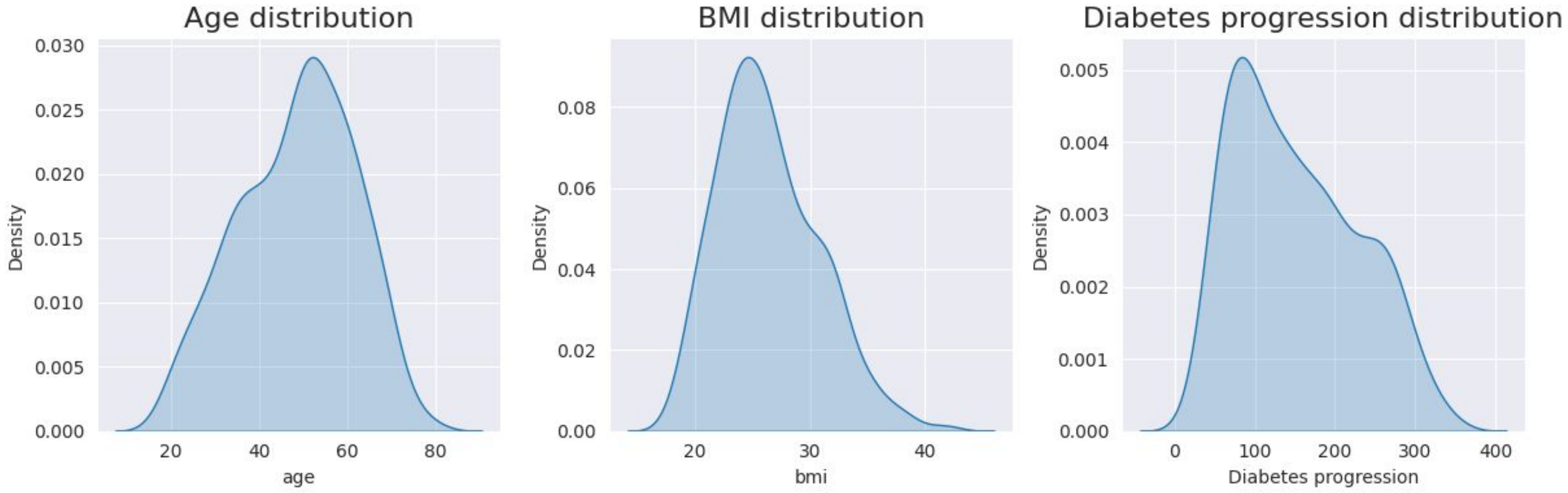
Exploratory Data Analysis: Univariate plots

- Can also be used for categorical data
- Can be helpful to add numbers on the bars

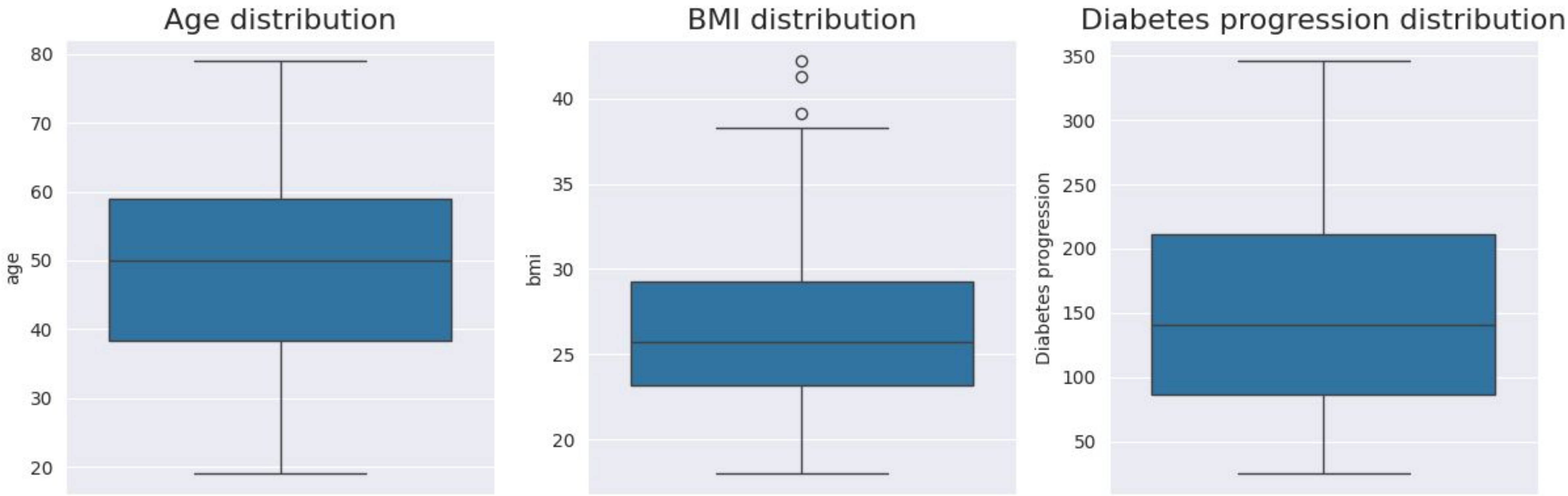


Exploratory Data Analysis: Univariate plots

- Density plots
  - Another way of inspecting the distribution
  - Uses a smooth continuous curve



Exploratory Data Analysis: Univariate plots

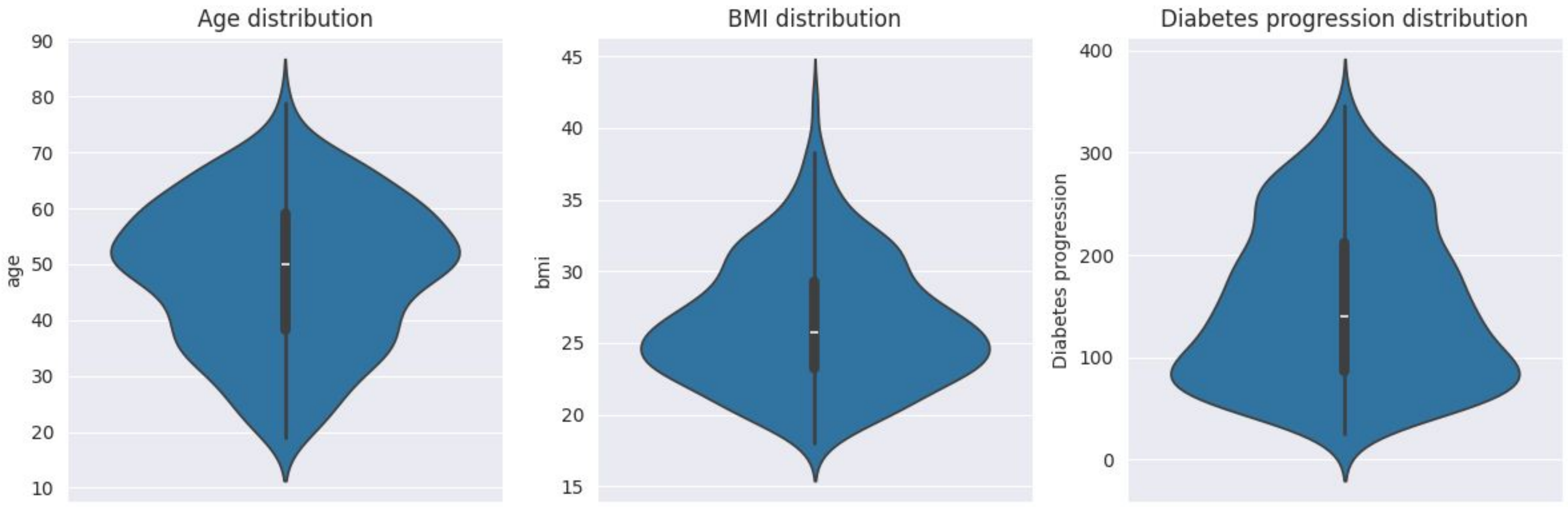


Exploratory Data Analysis: Univariate plots

- Box and Whisker plots
  - Another way of inspecting the univariate distribution
  - Instead of showing a complete distribution they give a short summary
    - Use a line to represent the median sample/entry
    - The box encompasses the middle 50% of the data. From the 25th to the 75th percentile
    - The “whiskers” are 1.5 times greater than the size of the spread of the middle 50% of the data
    - The dots are potential candidates for outlier values

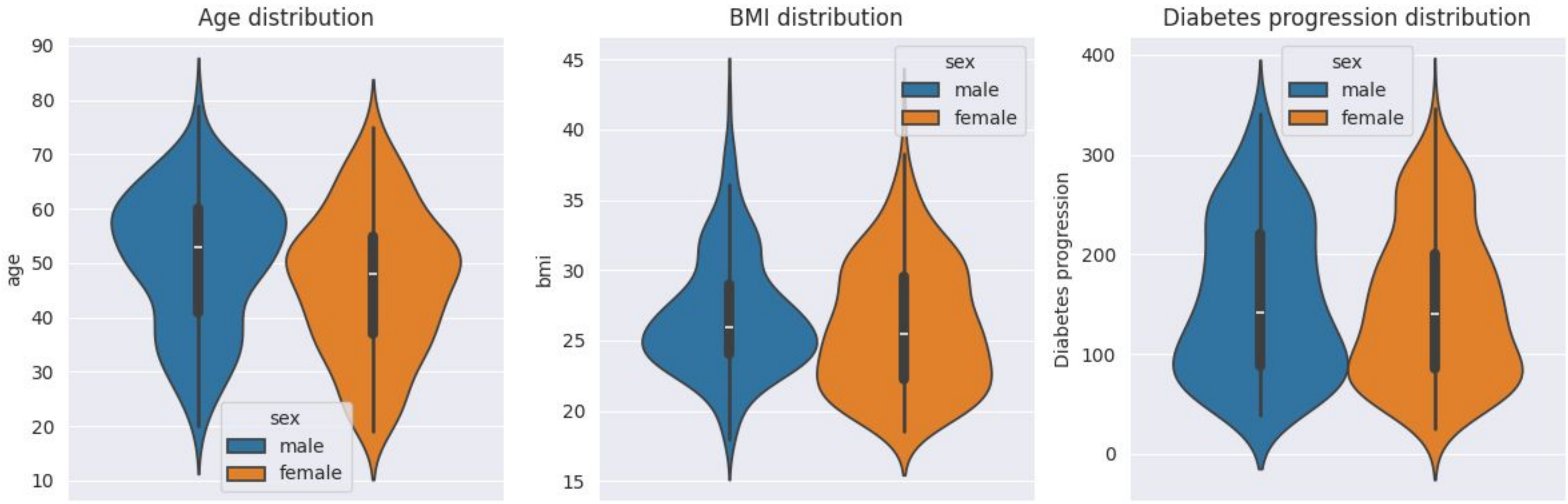
Exploratory Data Analysis: Univariate plots

- Violin plots: A better alternative to Box and whisker plots
  - Another way of inspecting the univariate distribution
  - Give a more complete/precise description of the data



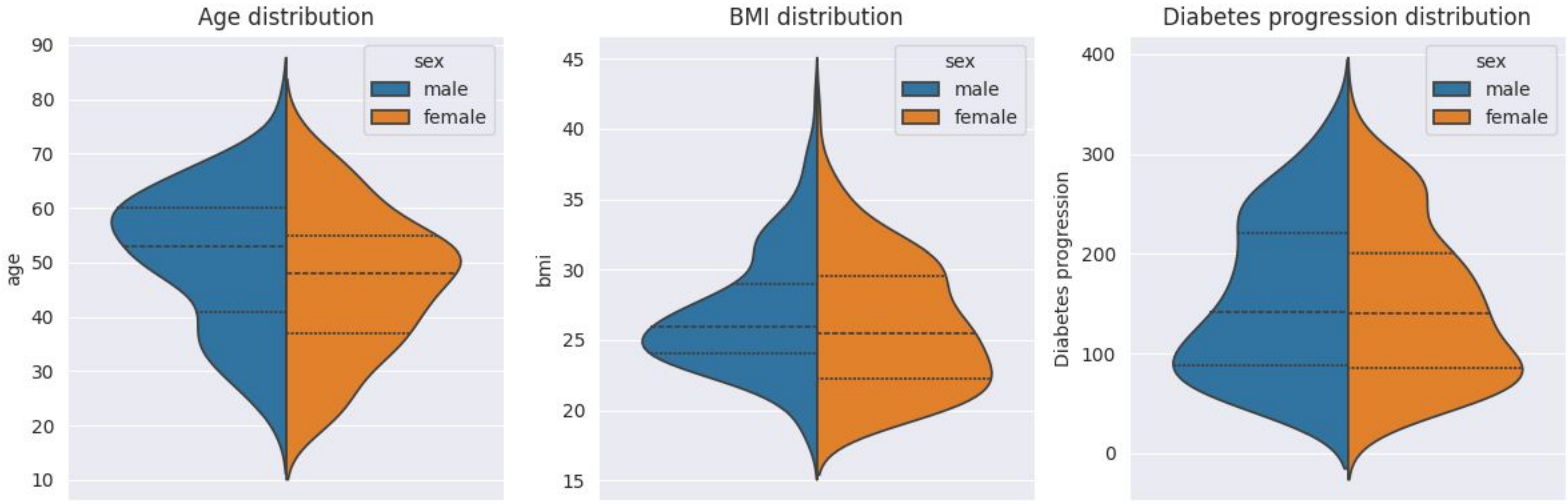
Exploratory Data Analysis: Univariate plots

- `Seaborn` allows for *splitting* using the parameter `hue`.
- Note that the *violins* are symmetrical, so the information can be compressed further



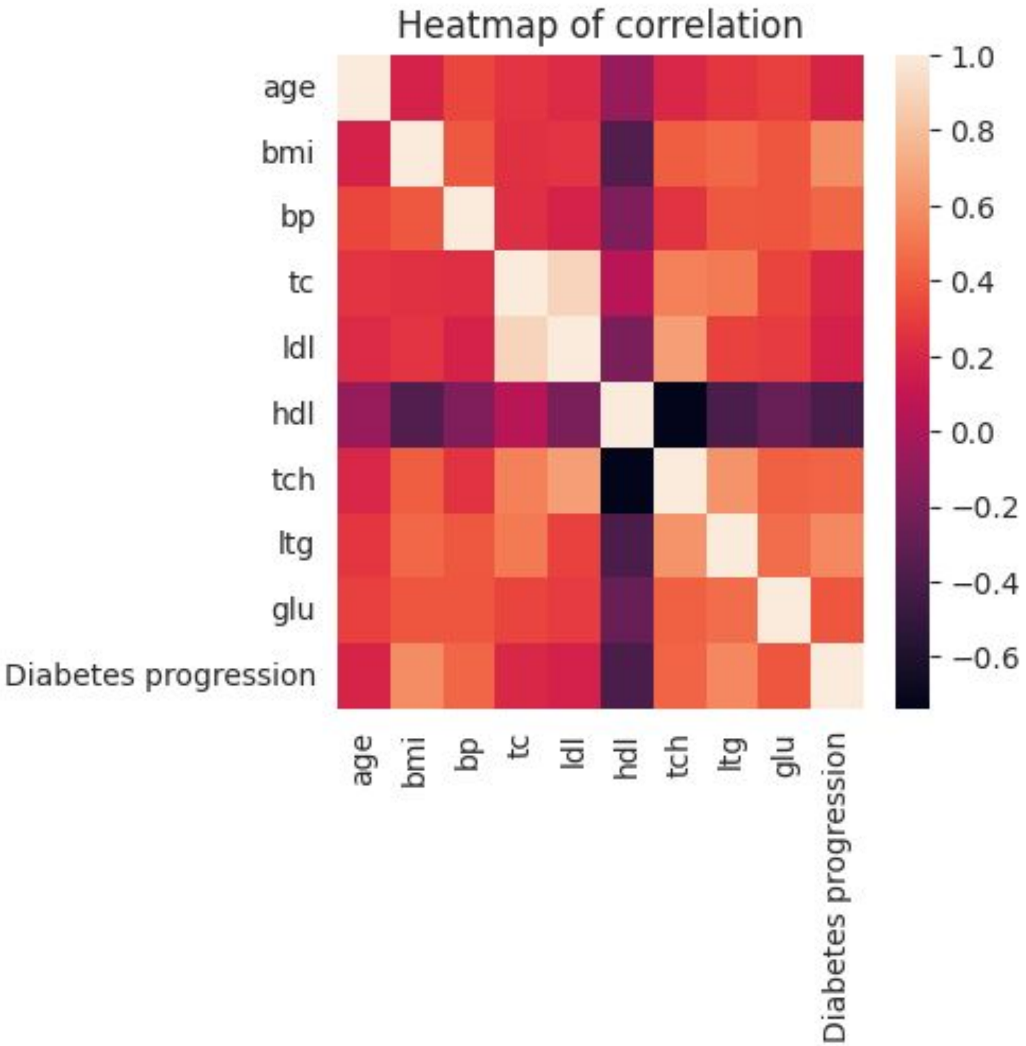
Exploratory Data Analysis: Univariate plots

- `Seaborn` allows for *splitting* using the parameter `hue`.
- Note that the *violins* are symmetrical, so the information can be compressed further



Exploratory Data Analysis: Multivariate plots

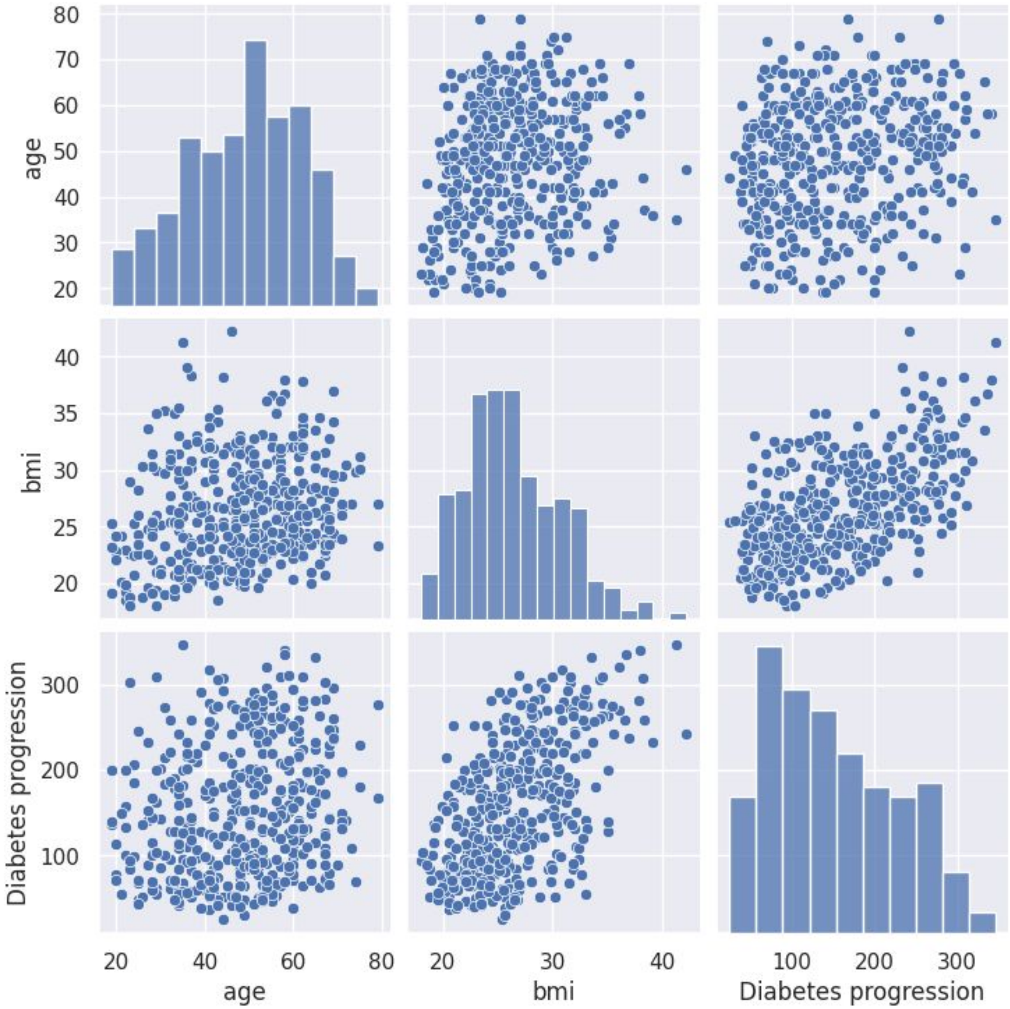
- Heatmap of correlations
  - Indicates how related two variables are
  - **Positive** correlation: Two variables change in **the same** direction
  - **Negative** correlation: Two variables change in **opposite** directions

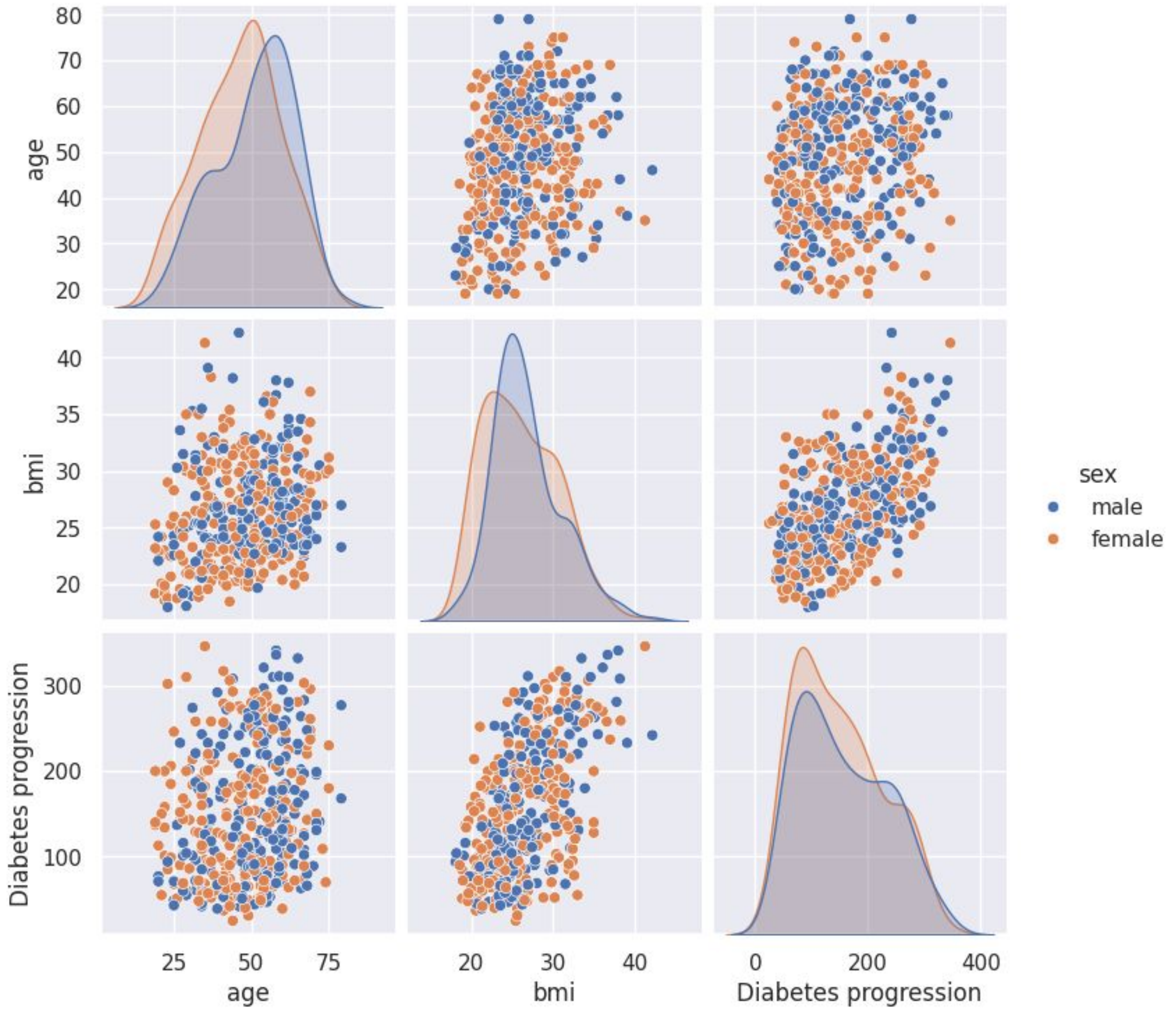


Exploratory Data Analysis: Multivariate plots

- Scatter Plot Matrix
  - Shows relationship between two variables as dots in two dimensions
  - Useful for spotting structured relationships between variables
  - Structural relationships may also be correlalated and good candidats for removal from the dataset

Exploratory Data Analysis: Multivariate plots





Exploratory Data Analysis: Multivariate plots

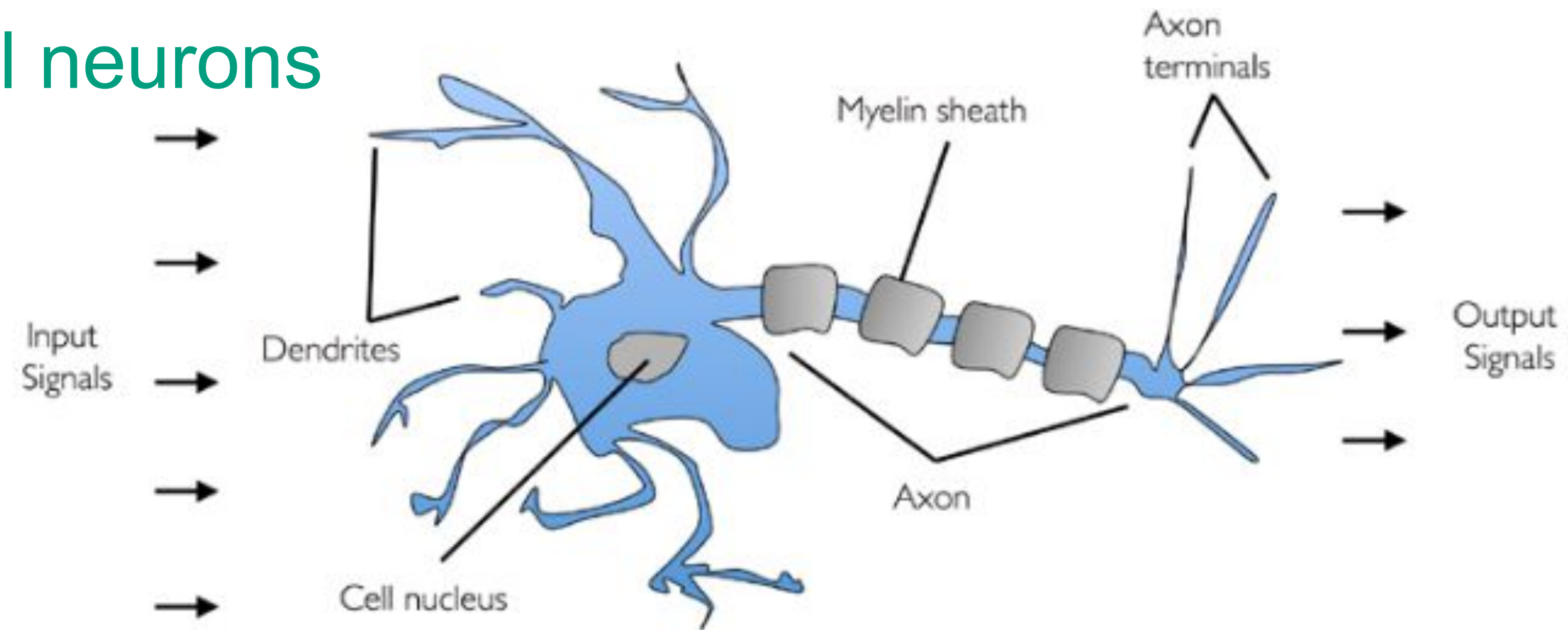
- To see more about why data visualization is important, and summary statistics alone are not enough check out this article
- The code used to generate the plots can be found in the file:
  - `01_raw_data_inspection.ipynb` under the folder for this lecture on Canvas
  - Check it out after the lecture

Learning as a concept (Ch. 2 in Raschka)

DAT200 - Applied Machine Learning

Department of Data Science, Faculty of Science of Technology

Artificial neurons

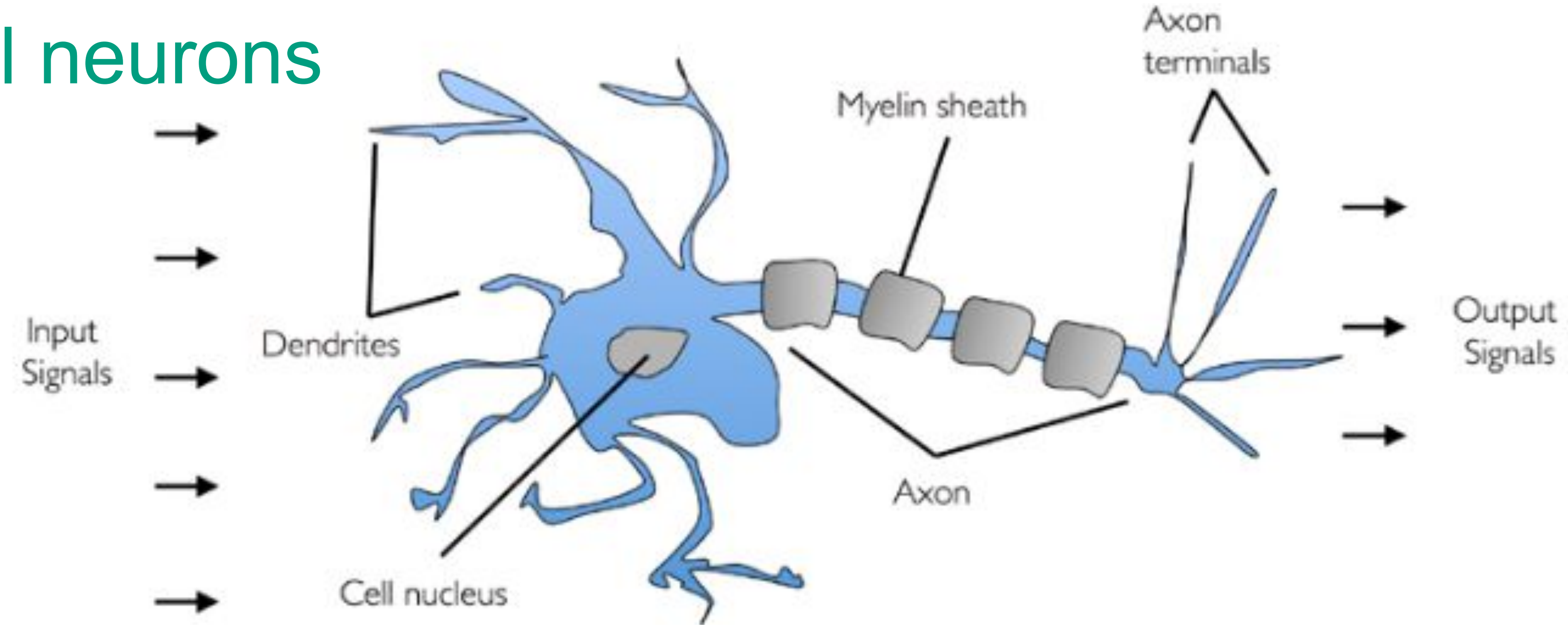


[1]

- The first ML models where modeled after the neurons in the human brain
- Neurons are interconnected nerve cells in the brain
- Neurons are involved in processing and transmitting:
  - Chemical signals
  - Electrical signals
- Neuron acts as a simple logic gate with binary output

[1] Figure from S. Rascka, V. Mirjalili. *Python Machine learning - 3rd Ed.* (2019), page 20

Artificial neurons

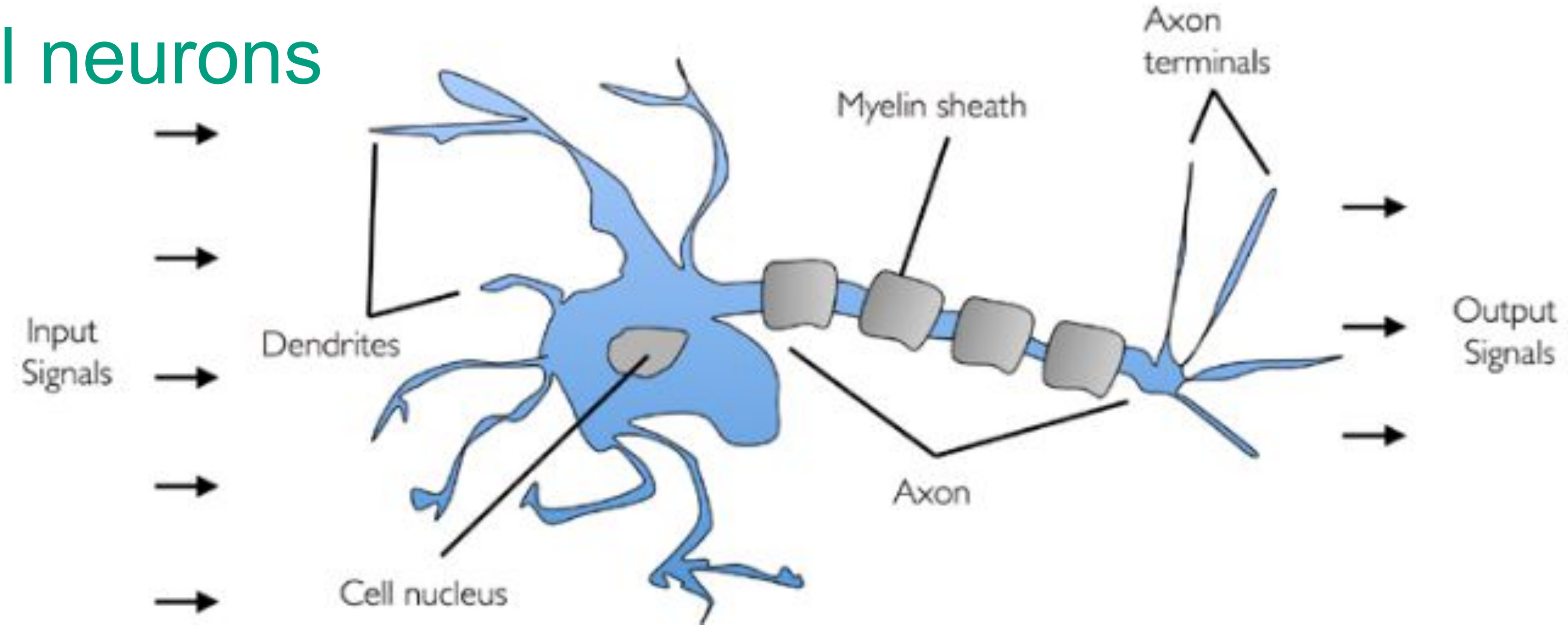


[1]

- Multiple signals arrive at dendrites
- Signals are integrated into cell body
- If accumulated signal exceeds a specific threshold → output signal is generated and passed on to axon
- McCullock-Pitts (MCP) neuron: first simplified concept of brain cell (1943)

[1] Figure from S. Rascka, V. Mirjalili. *Python Machine learning - 3rd Ed.* (2019), page 20

Artificial neurons

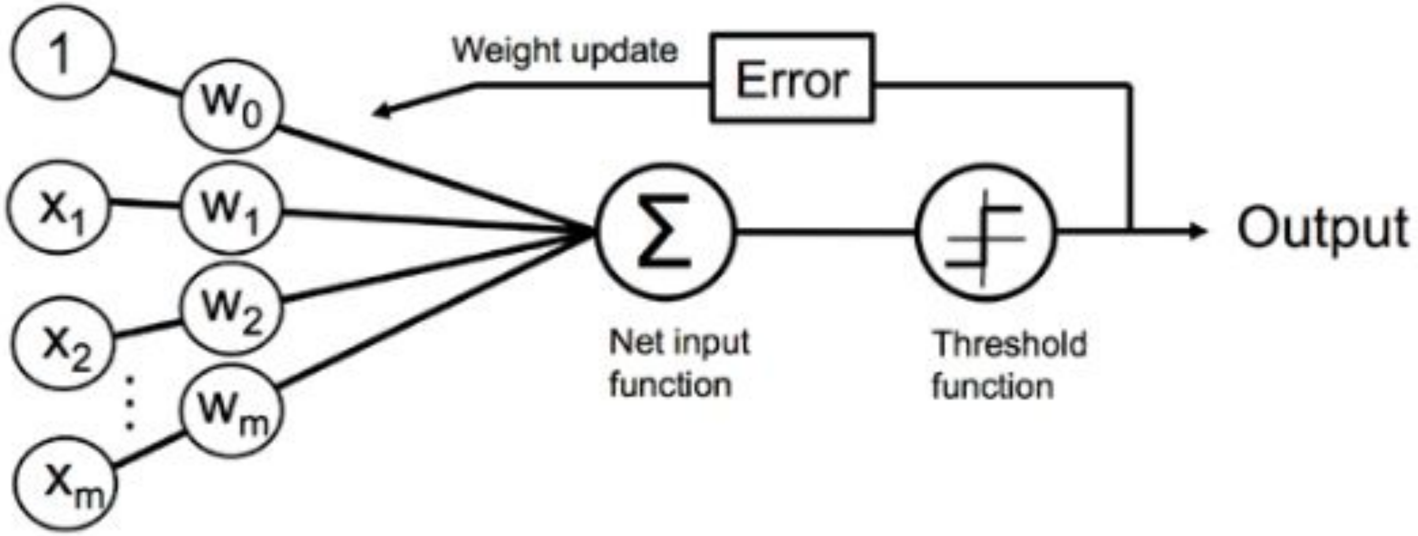


[1]

- Perceptron (first model of artificial neuron) learning rule based on MCP neuron model published in 1957
- Algorithm learns automatically optimal weight coefficients for input features
- Product of optimal weight and input feature decides whether neuron fires or not
- Can be used to predict whether an instance belongs to one class or another

[1] Figure from S. Rascka, V. Mirjalili. *Python Machine learning - 3rd Ed.* (2019), page 20

Perceptron Concept



- Each perceptron is a set of weights which are multiplied with the set of inputs and summed
- The sum is passed through a threshold function to yield a binary output 1 or -1.
- This is then called a *binary classification function*.
- The training algorithm computes a decision function based on weights and features.
- The goal is to find the optimal values for weights for best possible classification performance

[2] Figure from S. Rascka, V. Mirjalili. *Python Machine learning - 3rd Ed.* (2019), page 26

Formal Definition of An Artificial Neuron

- Binary classification task (two class problem)
  - First class coded as 1 (positive class)
  - Other class: -1 (negative class)
- Decision function ($\phi(z)$):
  - takes linear combinations of
  - values in vector **x**
  - corresponding weights in weight vector **w**
- z: net input

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

$m \times 1$ $\quad$ $m \times 1$

$$\boxed{z = w_1 x_1 + \ldots + w_m x_m}$$

Formal Definition of An Artificial Neuron

- Binary classification task (two class problem)
  - First class coded as 1 (positive class)
  - Other class: -1 (negative class)
- Decision function ($\phi(z)$):
  - takes linear combinations of
  - values in vector **x**
  - corresponding weights in weight vector **w**
- z: dot product of **x** and **w**

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

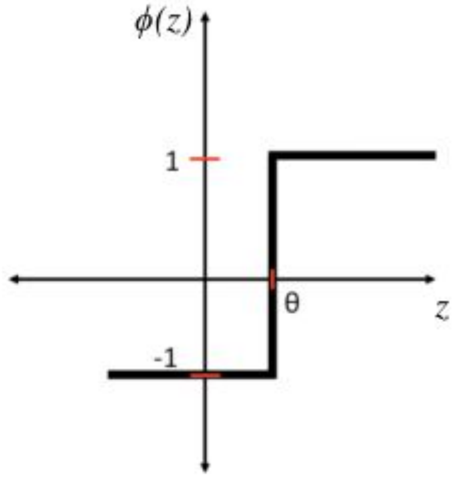
$m \times 1$ $\qquad$ $m \times 1$

$$\boxed{z = w_1 x_1 + \ldots + w_m x_m}$$

Formal Definition of An Artificial Neuron

- Given a specific sample $\boldsymbol{x}^{(i)}$
  - If net input z ≥ θ → predict class 1 for $\boldsymbol{x}^{(i)}$
  - If net input z < θ → predict class -1 for $\boldsymbol{x}^{(i)}$
  - θ represents threshold: at which level/value should sum of all weighted input signals be to predict class 1?
- For perceptron algorithm ϕ(·) is a variant of unit step function

$$\phi(z) = \begin{cases} 1 & if\ z \geq \theta \\ -1 & otherwise \end{cases}$$



Formal Definition of An Artificial Neuron

- For simplicity bring θ to the left side of the equation

$$z \geq \theta$$

$$w_1 x_1 + w_2 x_2 + \cdots + w_m x_m \geq \theta$$

$$-\theta + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m \geq 0$$

$$-\theta \cdot 1 + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m \geq 0$$

$$w_0 x_0 + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m \geq 0$$

where $w_0 = -\theta$

$x_0 = 1$

Formal Definition of An Artificial Neuron

- z now can be written in a more compact form

$$\boxed{z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} \boldsymbol{x}_j \boldsymbol{w}_j = \boldsymbol{w}^T \boldsymbol{x}}$$

$1 \; x \; (m+1) \qquad (m+1) \; x \; 1$

- and $\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$
- In ML literature $w_0 = -\theta$ is often called the bias unit
- **Summarised:**
  - **from input values x and weights w → compute net input z**
  - **from net input z and decision function ϕ(z) → to classification outputs -1 and 1**

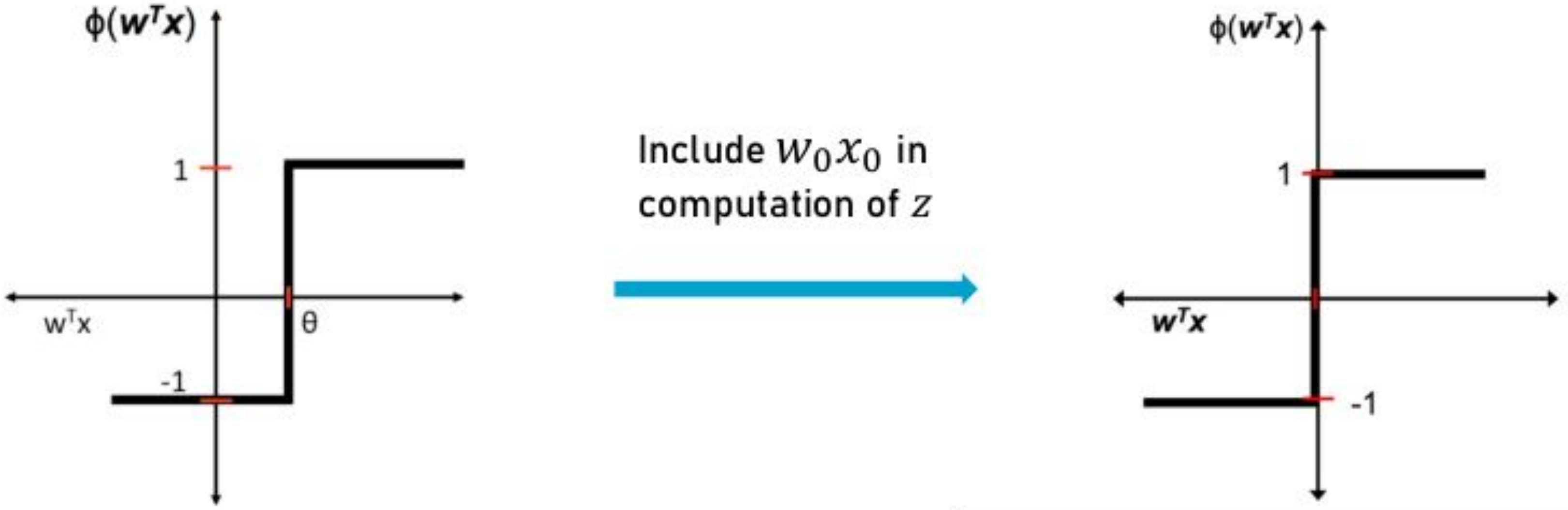
Formal Definition of An Artificial Neuron

- The value of net input $z = \mathbf{w}^T \mathbf{x}$ decides whether decision function of the perceptron produces output 1 og -1.

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

[3]



$$z = w_1x_1 + \ldots + w_mx_m$$

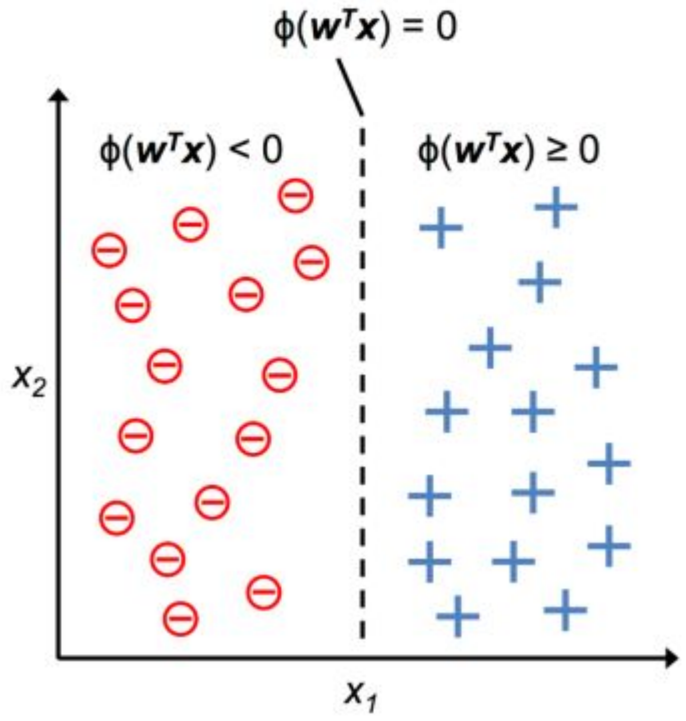
$$z = w_0x_0 + w_1x_1 + w_2x_2 + \cdots + w_mx_m$$

[3] Figure from S. Rascka, V. Mirjalili. *Python Machine learning - 3rd Ed.* (2019), page 23

Formal Definition of An Artificial Neuron

- The value of net input $z = \boldsymbol{w}^T \boldsymbol{x}$ decides whether decision function of the perceptron produces output 1 og -1.

[4]



[4] Figure from S. Rascka, V. Mirjalili. *Python Machine learning - 3rd Ed.* (2019), page 23

The Perceptron Learning Rule

- Idea behind MCP neuron and Rosenblatt’s thresholded perceptron model
  - Reductionist approach to mimic how a single neuron in the brain works
  - the neuron either fires or not
- Initial perceptron rule:
  - 1. Initialise the weights to 0 or small random numbers
  - 2. For each training sample $\boldsymbol{x}^{(i)}$
    - 2a. Compute output value $\hat{y}$ (prediction of true class label y)
    - 2b. Compare true class label y and predicted output $\hat{y}$
    - 2c. If different from one another, update weights
- **How to update the weights?**

The Perceptron Learning Rule

- Weights w_j in weight vector $\boldsymbol{w}$ are updated simultaneously
- Update of each weight w_j can be more formally written as

$$\boxed{w_j := w_j + \Delta w_j}$$

- Value of Δw_j, which is used to update the weight w_j is calculated by the **perceptron learning rule**

$$\boxed{\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}}$$

η : Learning rate (typically constant between 0.0 and 1.0)

$y^{(i)}$: true class label

$\hat{y}^{(i)}$: predicted class label

$x^{(i)}$: value of feature j in sample vector i

The Perceptron Learning Rule

- For a two-dimensional (two variables: x_1 and x_2) dataset:

$$\Delta w_0 = \eta \left(y^{(i)} - output^{(i)} \right)$$

$$\Delta w_1 = \eta \left(y^{(i)} - output^{(i)} \right) x_1^{(i)}$$

$$\Delta w_2 = \eta \left(y^{(i)} - output^{(i)} \right) x_2^{(i)}$$

The Perceptron Learning Rule

- **Correct** class label predictions:
  - **Negative class** predicted **correctly:** $$\Delta w_j = \eta\left(-1-(-1)\right)x_j^{(i)} = 0$$
  - **Positive class** predicted **correctly:** $$\Delta w_j = \eta\left(1-1\right)x_j^{(i)} = 0$$

- **Incorrect class label predictions**
  - **Positive class** predicted **incorrectly** $$\Delta w_j = \eta\left(1--1\right)x_j^{(i)} = \eta\left(2\right)x_j^{(i)}$$
  - **Negative class predicted incorrectly** $$\Delta w_j = \eta\left(-1-1\right)x_j^{(i)} = \eta\left(-2\right)x_j^{(i)}$$

The Perceptron Learning Rule

- Intuition for multiplicative factor $x^{(i)}$ in $\boxed{\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}}$

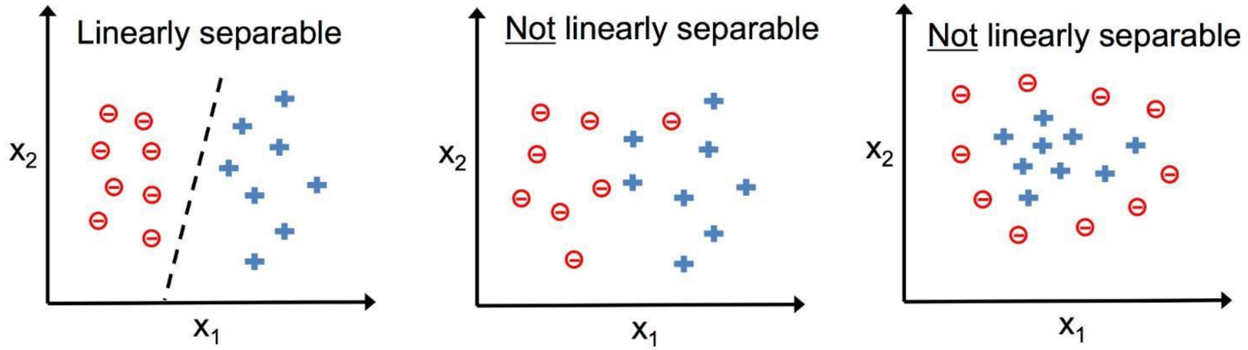
- Example: $\hat{y}^{(i)} = -1, \quad y^{(i)} = +1, \quad \eta = 1$
- Assume: $x_j^{(i)} = 0.5$ and sample j has been misclassified as -1
- Consequently, weight update will be: $\Delta w_j = (1 - -1)0.5 = (2)0.5 = 1$
- We see that weight update Δw is proportional to value of $x^{(i)}$

- Now assume: $x_j^{(i)} = 2$ sample j has been misclassified as -1
- Consequently, weight update will be $\Delta w_j = (1 - -1)2 = (2)2 = 4$

Important Notes on Perceptron Training Algorithm

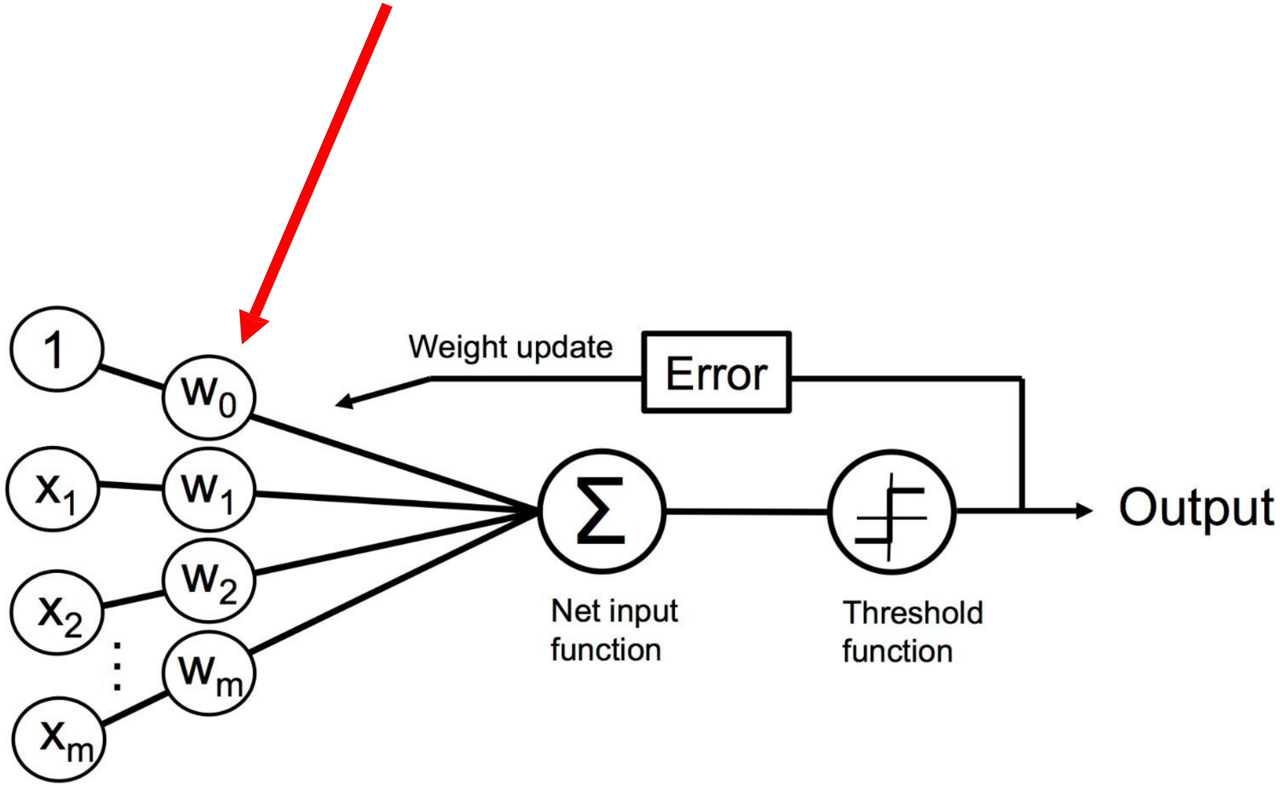
- **Convergence** of perceptron algorithm guaranteed only if
  - Two classes are **linearly** separable
  - Learning rate is **sufficiently small**
- If classes cannot be separated by a linear decision boundary take **at least one** of the following two measures
  - Set maximum number of iterations (**epochs**) over training samples
  - Set threshold for maximum number of tolerated misclassifications
- If classes **not** linearly separable **AND** none of the two measures above were taken

   perceptron **never stops** updating weights



Important Notes on Perceptron Training Algorithm

Goal: Find optimal values for weights for best possible classification performance



Important Notes on Perceptron Training Algorithm

- The perceptron class: `perceptron.py`
- Training perceptron class on iris data: `Ch02_01_iris_perceptron_alt1.py`
- Same as `Ch02_01_iris_perceptron_alt1.py`, but using package `mlxtend` for plotting:
- `Ch02_01_iris_perceptron_alt2.py`
- Understanding random seed: `Ch_02_play_with_random.py`
- Extended version of `perceptron.py` that collects values of weights across all epochs:
- `perceptron_ext.py`
- Extended version of `Ch02_01_iris_perceptron_alt1.py` visualising weight updates on iris data: `Ch02_03_iris_perceptron_weightPlotting.py`

Exercise: Understand Code in perceptron.py

- Open file: `perceptron.py`
- Identify where in the code the following is computed
  - Net input
  - Weight updates
  - Threshold function

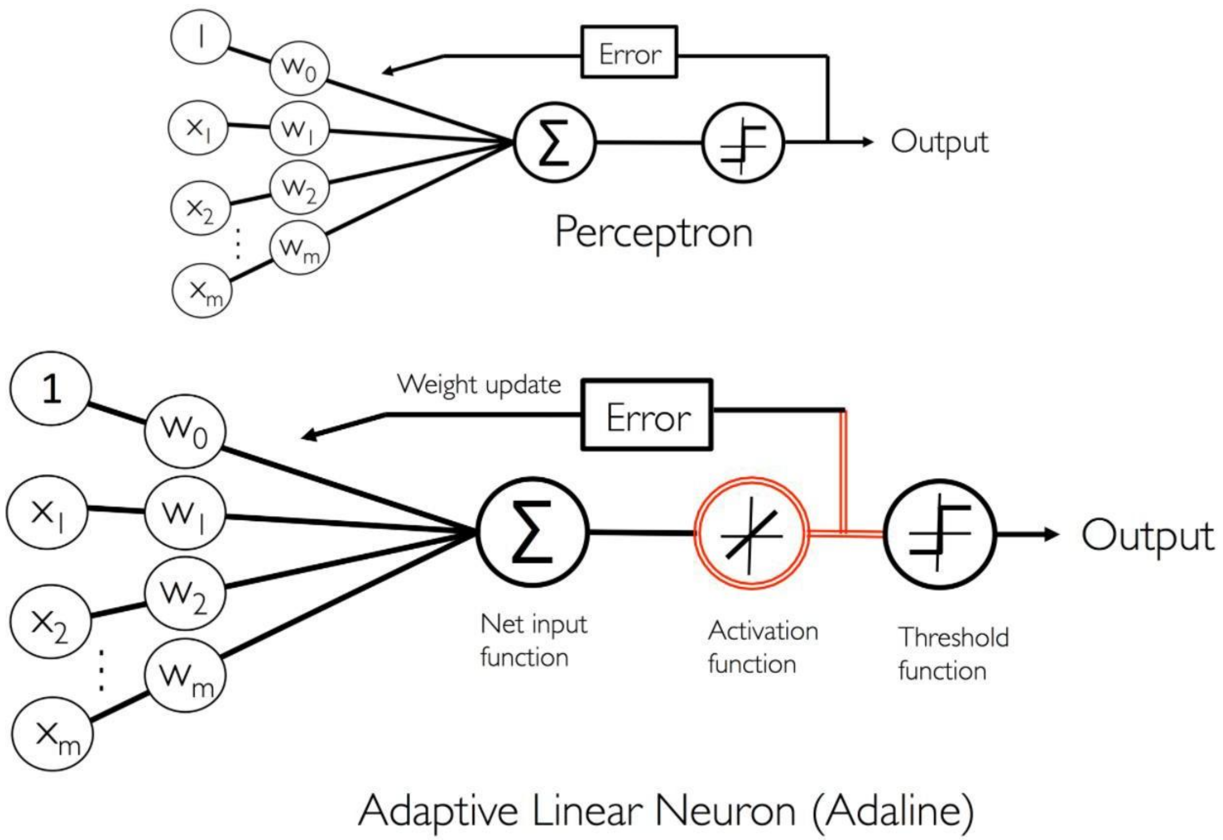
- Where are the weights initialised?
- What are the dimensions of the prediction?
- How often are the weights updated?
- How often is the net input computed?
- How is the error checked for?

Adaptive Linear Neurons - Learning Convergence

- **ADA**ptive **LI**near **NE**uron (Adaline)
- Published in 1960
- Considered to be an improvement over Perceptron algorithm
- **Key concept** of Adaline algorithm – **defining** and **minimising** cost functions
- The concept lays **foundation** for **more advanced machine learning algorithms**
    - Classification models (e.g. logistic regression, Support vector machines (SVM))
    - Regression models
- Introduces concept of activation functions
    - Various activation functions used in artificial neural networks
    - Logistic, hyperbolic tangent (tanh), rectified linear unit (RELU), etc.

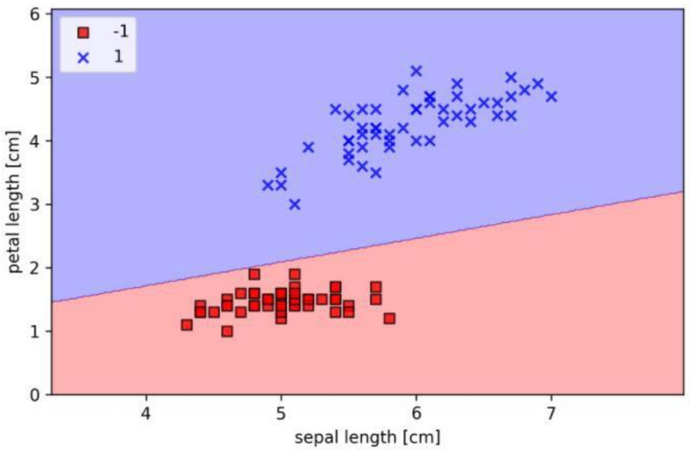
Adaptive Linear Neurons - Learning Convergence

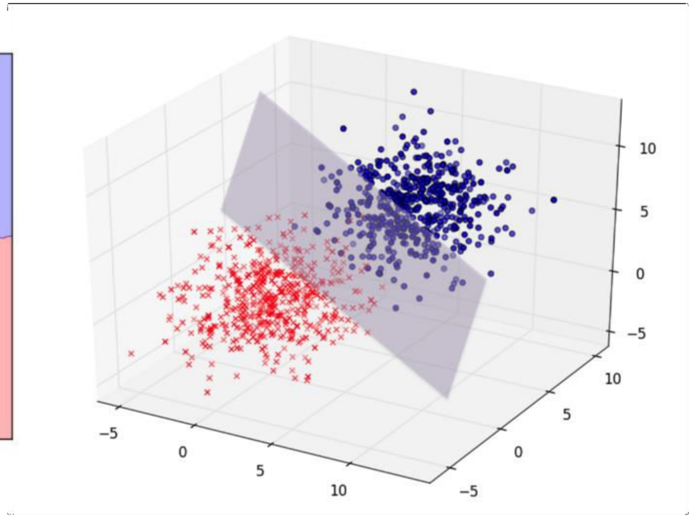
- Illustration of the Perceptron and Adeline algorithms



Adaptive Linear Neurons - Learning Convergence

- **Common properties** of Perceptron and Adaline algorithms
  - Both are classifiers for **binary classification** (two-class problems)
  - Both have a **linear decision boundary**
  - Both use **threshold function**





More than three
input variables

Linear decision boundary
for two input variables
 a straight line

Linear decision boundary
for three input variables
 a plane

Linear decision boundary
for more than three variables
 a hyperplane

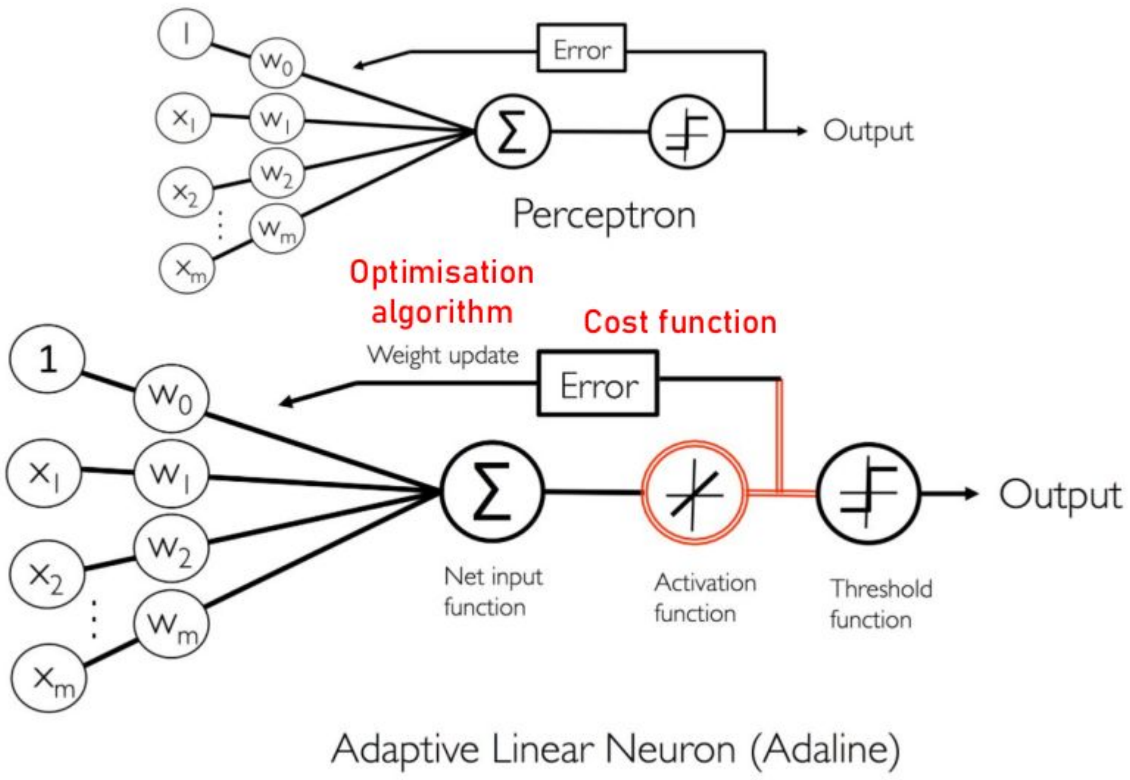
Adaptive Linear Neurons - Learning Convergence

- **Key differences** between the Adaline and Perceptron algorithms
- **Perceptron**:
  - Weights are updated using a **step function** $\phi(z)$ (serves as an activation function)
  - Computation of error: compares **true class labels** to **predicted class labels**
  - Updates weights **immediately** after a misclassification. Updates happen **multiple times** during a full set of iterations (epoch)
- **Adaline**:
  - Weights are updated based on a **linear activation function** $\phi(z)$
  - The linear activation function is simply an **identity function** of the net input
  - Computation of error: compares **true class labels** to the **continuous output** from the activation function
  - Updates weights only **at the end** of each epoch (batch-wise updates)

Linear activation function $\boxed{\phi(z) = \phi(\mathbf{w}^T \mathbf{x}) = \mathbf{w}^T \mathbf{x}}$

Adaptive Linear Neurons - Learning Convergence

- Linear activation function is used for learning of the weights
- However, a threshold function is still used to make final prediction
- Figure below illustrates main differences between the Perceptron and Adaline algorithms



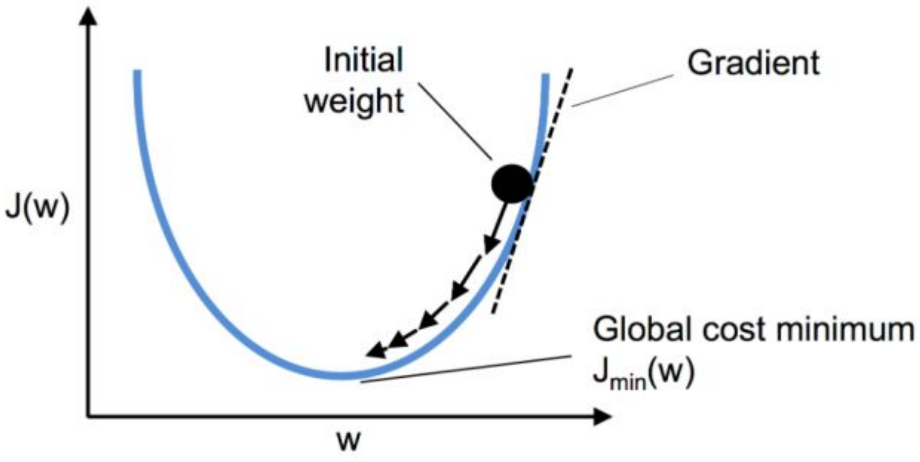
Minimizing The Cost Function

- One key ingredient in many machine learning algorithms: a defined objective function (loss function; cost function)
- Goal of using objective/loss/cost function:
  - Capture the performance of the model, how well the model is predicting classes (quality of the output)
  - Doing so by computing error or distance score between the true class label and the output of activation function
  - Compute weights such that loss function is minimised (find global cost minimum where error is as small as possible)

- *Learning* means finding a combination of weights that minimizes a loss function for a given set of training samples and their corresponding targets (true class labels)

Minimizing The Cost Function

- Adeline algorithm:
  - Cost function *J* learns weights as the Sum of Squared Errors (SSE) between outputs (from activation function) and true class labels



$$\boxed{J(\boldsymbol{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2}$$

Cost function (SSE)

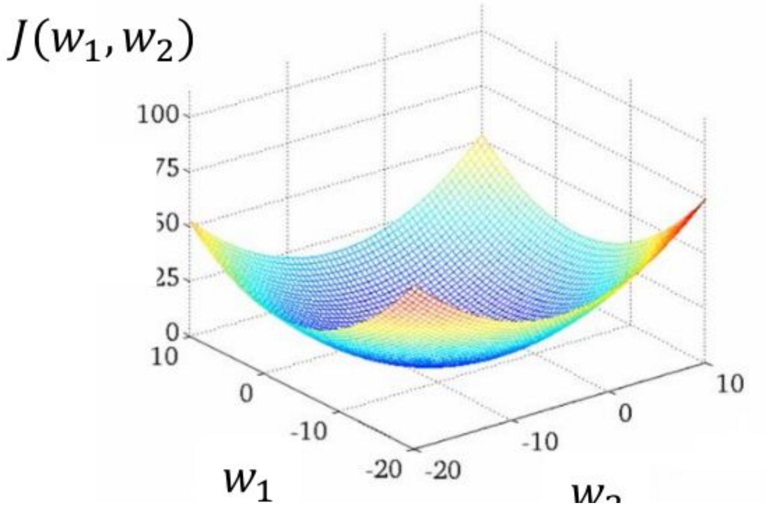
$$\boxed{\phi\,(z) = \phi(\boldsymbol{w}^T\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}}$$

Linear activation function

Use ***optimisation*** algorithm named **gradient descent** to find global minimum of cost function $J(w)$.

Minimizing The Cost Function

- Adeline algorithm:
    - Cost function *J* learns weights as the Sum of Squared Errors (SSE) between outputs (from activation function) and true class labels



$$J(\mathbf{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

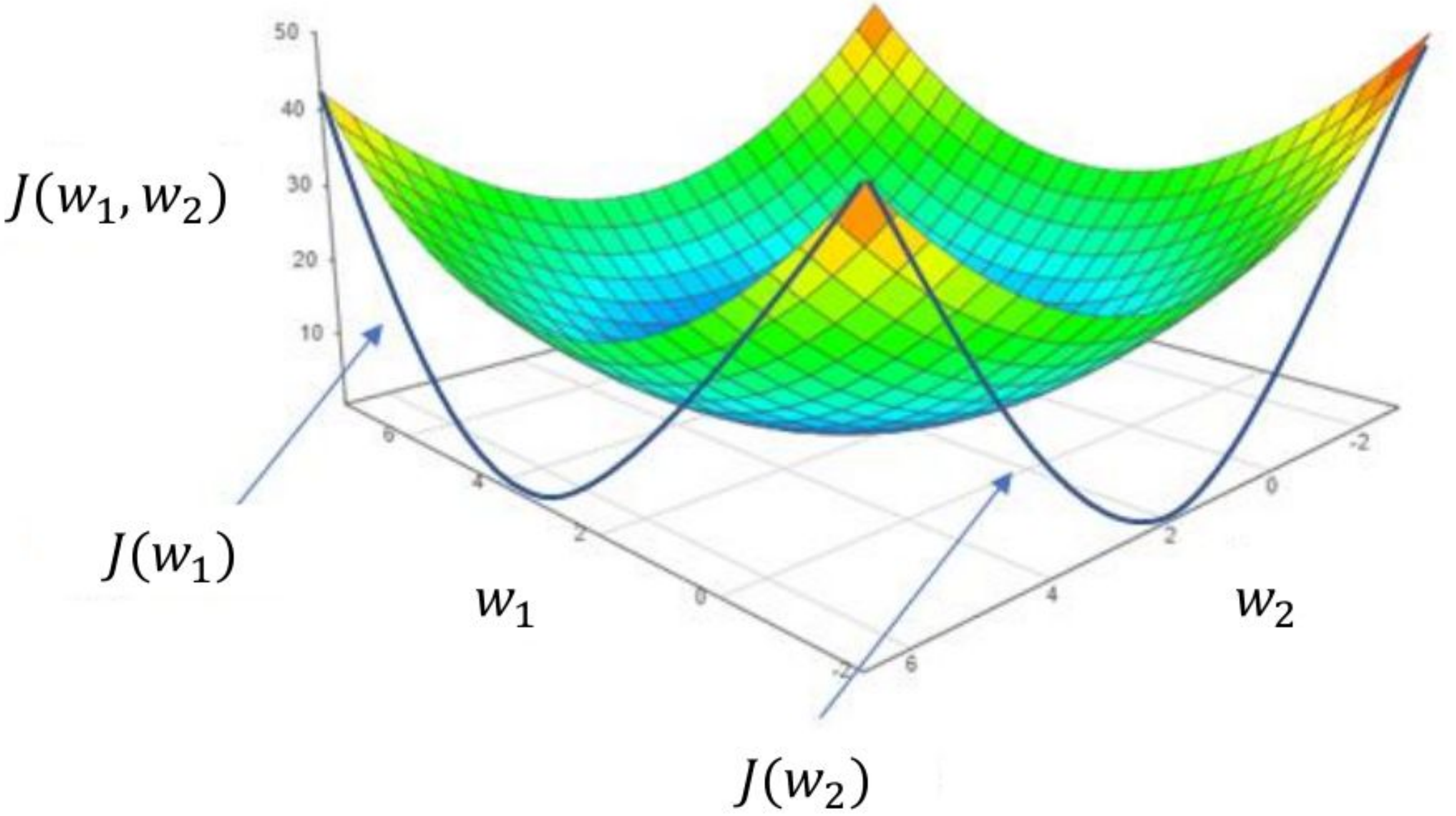
Cost function (SSE)

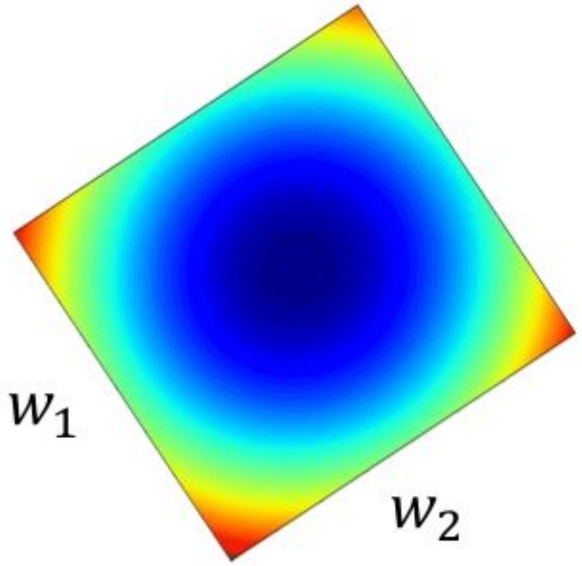
$$\phi(z) = \phi(\mathbf{w}^T\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$

Linear activation function

Use <u>*optimisation*</u> algorithm named **gradient descent** to find global minimum of cost function $J(w)$.

Minimizing The Cost Function





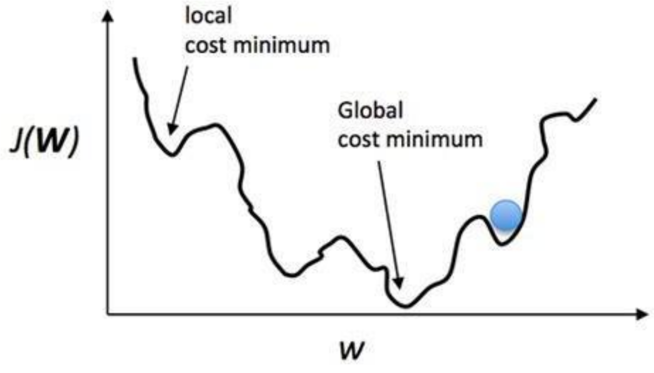
Minimizing The Cost Function

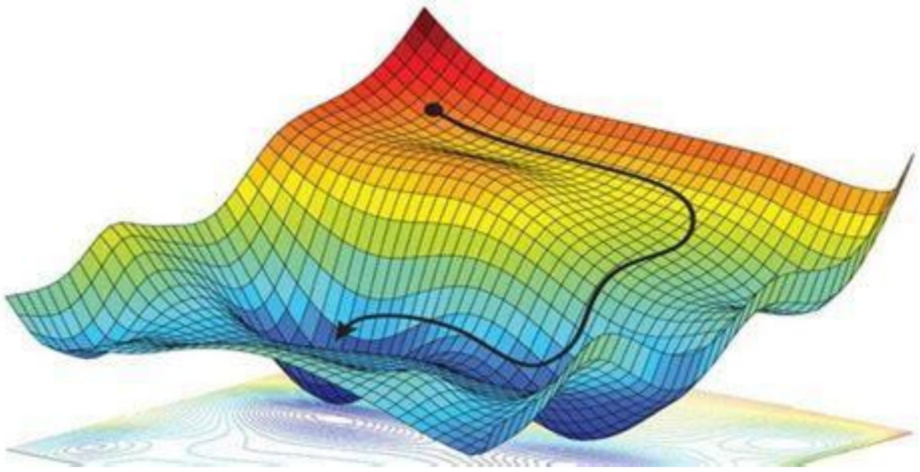
- Main advantage of linear activation function over the unit step function is that cost function becomes differentiable
  - This allows us to derive the gradient of loss function (important information for moving in right direction towards global cost minimum)

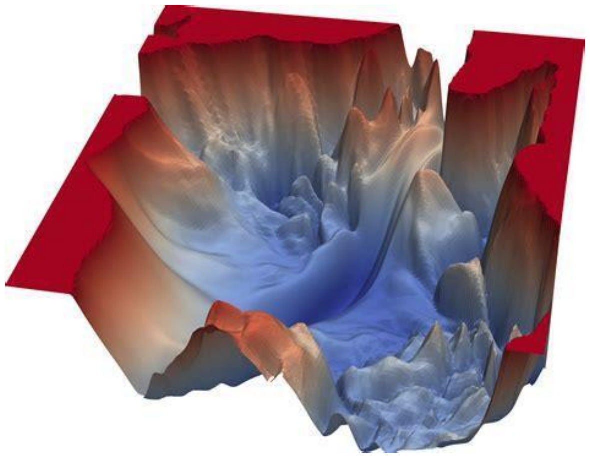
- A convenient property of SSE cost function: it is convex
  - This in turn enables the gradient descent algorithm to move towards weights result in a global cost minimum

Minimizing The Cost Function

- Examples of loss landscapes (not Adaline)







Minimizing The Cost Function

- Adeline algorithm:
  - Cost function *J* learns weights as the Sum of Squared Errors (SSE) between calculated outcome and true class label

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

- Using gradient descent update weights by taking step in opposite direction of the gradient of the cost function $\nabla J(w)$

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}$$

- The weight change Δw is defined as the negative gradient $\nabla J(w)$ multiplied by the learning rate η

$$\Delta\boldsymbol{w} = -\eta\nabla J(\boldsymbol{w})$$

- Update of the weights

$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = \eta\sum_i\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)x_j^{(i)}$$

$$\frac{\partial J}{\partial w_j} = -\sum_i\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)x_j^{(i)}$$

Minimizing The Cost Function

The squared error derivative

If you are familiar with calculus, the partial derivative of the SSE cost function with respect to the *j*th weight can be obtained as follows:

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)^2$$

$$= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)^2$$

$$= \frac{1}{2} \sum_i 2 \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \phi\left(z^{(i)}\right) \right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_i \left(w_j^{(i)} x_j^{(i)} \right) \right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \left(-x_j^{(i)} \right)$$

$$= -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$$

Minimizing The Cost Function

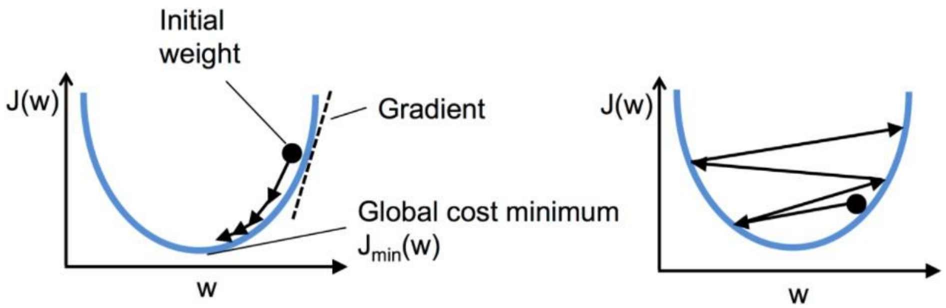
- Adaline algorithm:
  - Update of the weights

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)} \right) \right) x_j^{(i)}$$

- A single weight update is computed based on all samples in the training set (instead of incrementally after each sample as with Perceptron algorithm) therefore referred to as **batch gradient descent**
- Note the similarity with the weight updates of the Perceptron algorithm
  - Perceptron: $\phi(z)$ takes values -1 or 1
  - Adeline: $\phi(z)$ takes continuous values $\boldsymbol{w}^T\boldsymbol{x}$

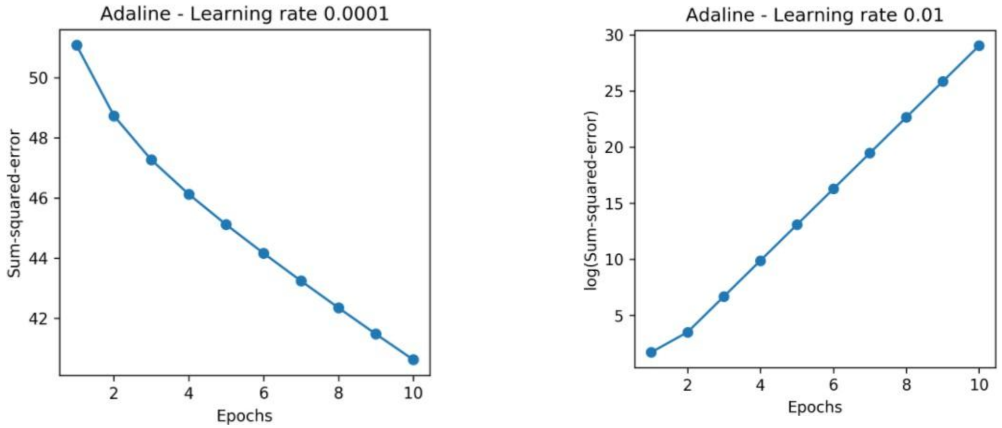
Minimizing The Cost Function

- Error over number of epochs – important to find appropriate learning rate η for weight updates Δw



adaline.py
Ch02_04_iris_adaline.py

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$



Improving Gradient Descent with Feature Scaling

- Many machine learning algorithms require some sort of feature scaling for optimal performance (also called standardisation)
- Gradient descent is one of the many algorithms that benefit from feature scaling

- Scaling (standardisation) gives the data some specific properties
  - Standard distribution helps the gradient descent learning to converge more quickly
  - Shifts the mean of each feature so that it is centered at zero
  - Each feature has a standard deviation of 1

$$\boldsymbol{x}'_j = \frac{\boldsymbol{x}_j - \mu_j}{\sigma_j}$$

$\boldsymbol{x}_j$: feature vector (variable or column in data)
μ_j: mean of feature vector
σ_j: standard deviation of feature vector
$\boldsymbol{x}'_j$: scaled feature vector

Improving Gradient Descent with Feature Scaling

Person	Height (cm)	Weight (kg)	Shoe size
Person A	174	55	46
Person B	188	92	45
Person C	158	65	42
Person D	202	110	49
Person E	171	96	44
Person F	193	79	48
Mean	181	82.833333	45.6667
STD	16.198765	20.507722	2.58199

Original
data

Person	Height (cm)	Weight (kg)	Shoe size
Person A	-7	-27.833333	0.33333
Person B	7	9.1666667	-0.66667
Person C	-23	-17.833333	-3.66667
Person D	21	27.166667	3.33333
Person E	-10	13.166667	-1.66667
Person F	12	-3.8333333	2.33333
Mean	0.00	0.00	0.00
STD	16.198765	20.507722	2.58199

Centred
data
(Zero-mean)

Person	Height (cm)	Weight (kg)	Shoe size
Person A	-0.4321317	-1.3572123	0.1291
Person B	0.4321317	0.4469861	-0.2582
Person C	-1.4198613	-0.8695911	-1.42009
Person D	1.2963951	1.3247043	1.29099
Person E	-0.617331	0.6420346	-0.6455
Person F	0.7407972	-0.1869215	0.9037
Mean	0.00	0.00	0.00
STD	1	1	1

Standardised
(scaled)
data

Improving Gradient Descent with Feature Scaling

- Number of objects (rows):
  - $i = 1 \dots N$
- Number of variables (columns):
  - $j = 1 \dots K$
- Observed value x_{ij} for
  - i'th sample
  - j'th variable

$$X = \begin{pmatrix} x_{11} & \cdots & x_{1K} \\ \vdots & \ddots & \vdots \\ x_{N1} & \cdots & x_{NK} \end{pmatrix}$$

center

$$\boxed{x_{ij,cent} = x_{ij} - \bar{x}_j}$$

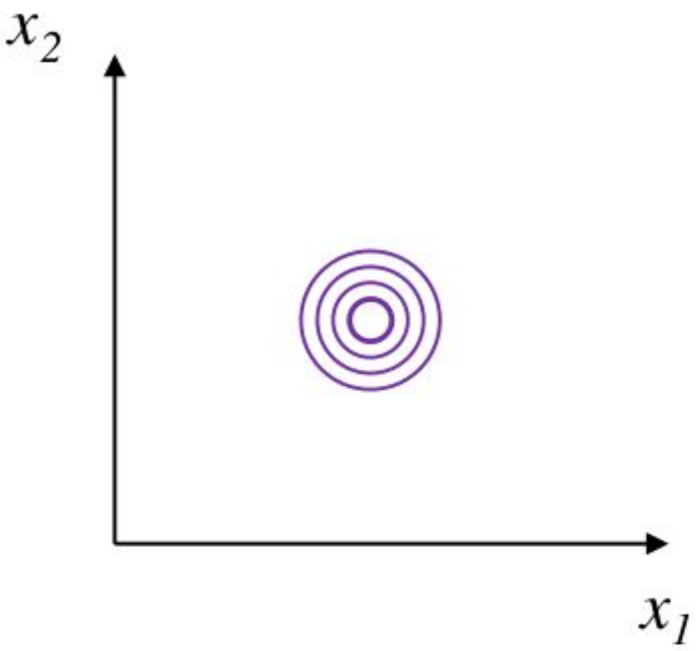
$$\bar{x}_j = \frac{1}{N}\sum_{j=1}^{N} x_{ij}$$

standardise

$$\boxed{x_{ij,stand} = \frac{x_{ij} - \bar{x}_j}{\sigma_j}}$$

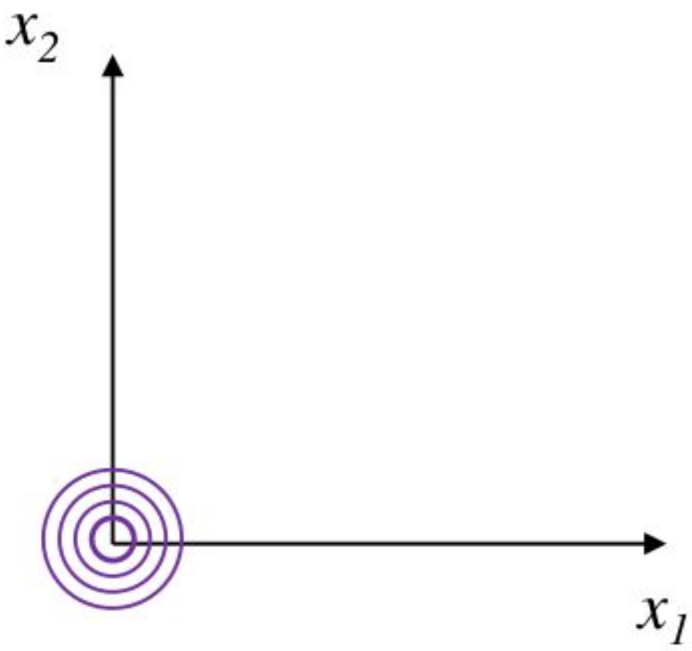
$$\sigma_j = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(x_{ij} - \bar{x}_j\right)^2}$$

Improving Gradient Descent with Feature Scaling



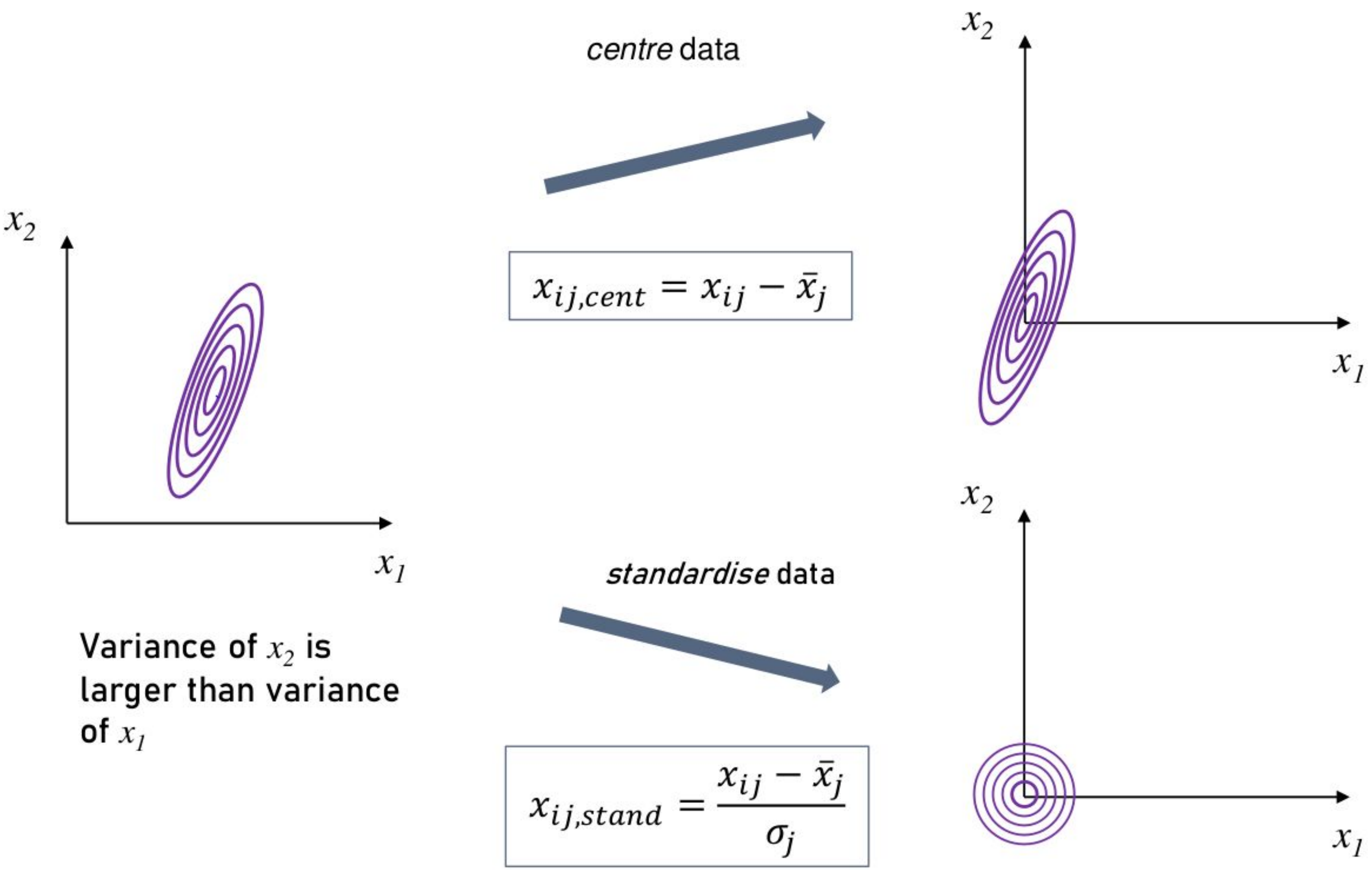
centre data

$$\boxed{x_{ij,cent} = x_{ij} - \bar{x}_j}$$



Equal variance of x_1 and x_2

Improving Gradient Descent with Feature Scaling



Improving Gradient Descent with Feature Scaling

- Feature scaling (standardisation) helps gradient descent (our optimiser) to find good or optimal solution in fewer steps
- Note that SSE (cost function) may remain non-zero even though all samples were classified correctly

Improving Gradient Descent with Feature Scaling

- Feature scaling with numpy – one feature at a time

```
# Standardise X one feature at a time (not very efficient)
X_sc = X.copy()
X_sc[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()
X_sc[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()
```

- Feature scaling with numpy – all variables at once

```
# Standardise X (all features at once)
X_sc = X.copy()
X_sc = (X_sc - np.mean(X_sc, axis=0)) / np.std(X_sc, axis=0)
```

```
Ch02_05_iris_perceptron_weightPlotting_selectClasses_scaling.py
Ch02_06_iris_adaline_weightPlotting_selectClasses_scaling.py
Ch02_07_scaling_examples.py
```

Thank you for coming!