STEVENS
INSTITUTE OF TECHNOLOGY
1870

PEP 559
Machine Learning in Quantum Physics

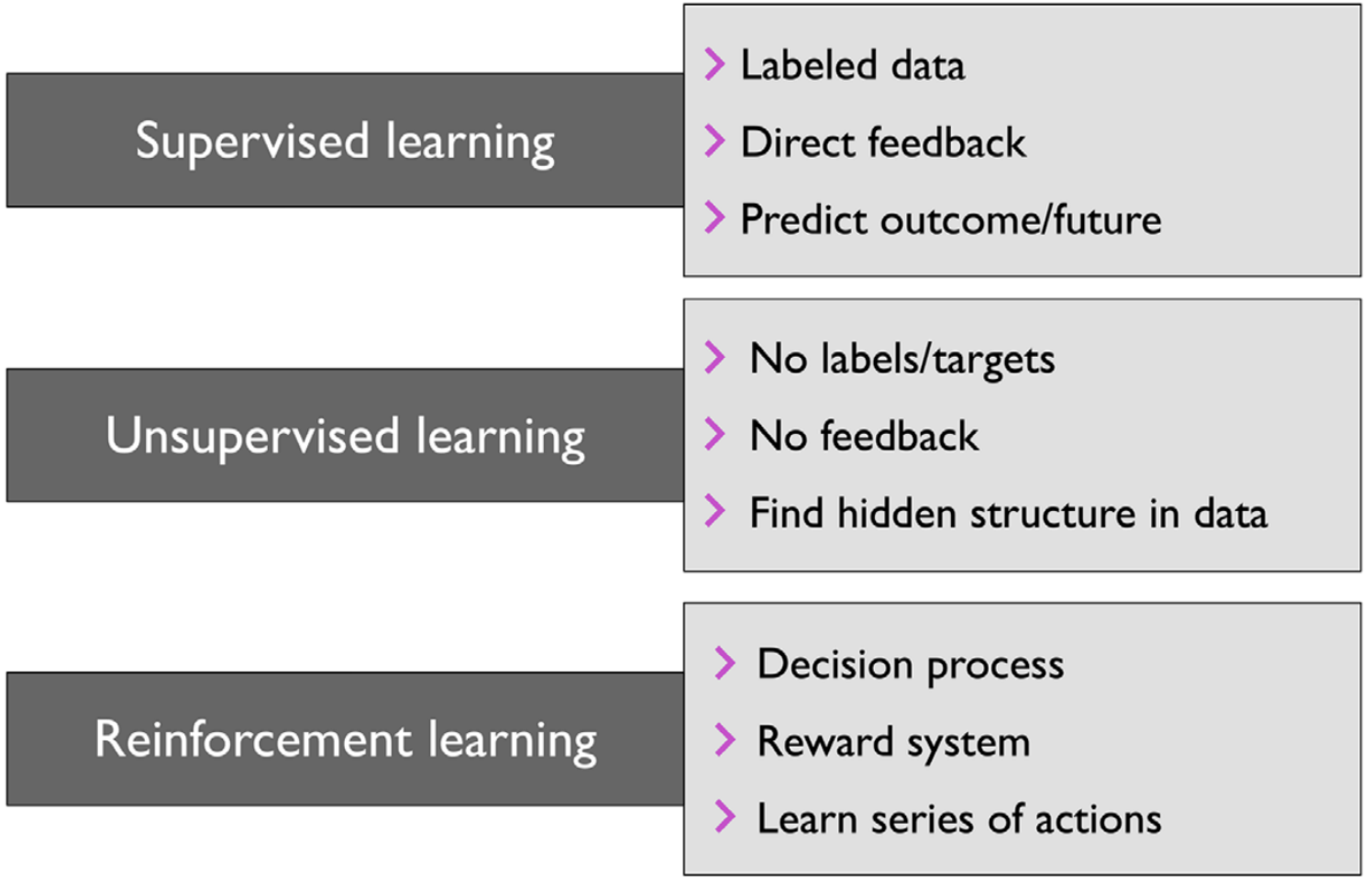
Dr. Chunlei Qu

Spring 2025

Four Modules

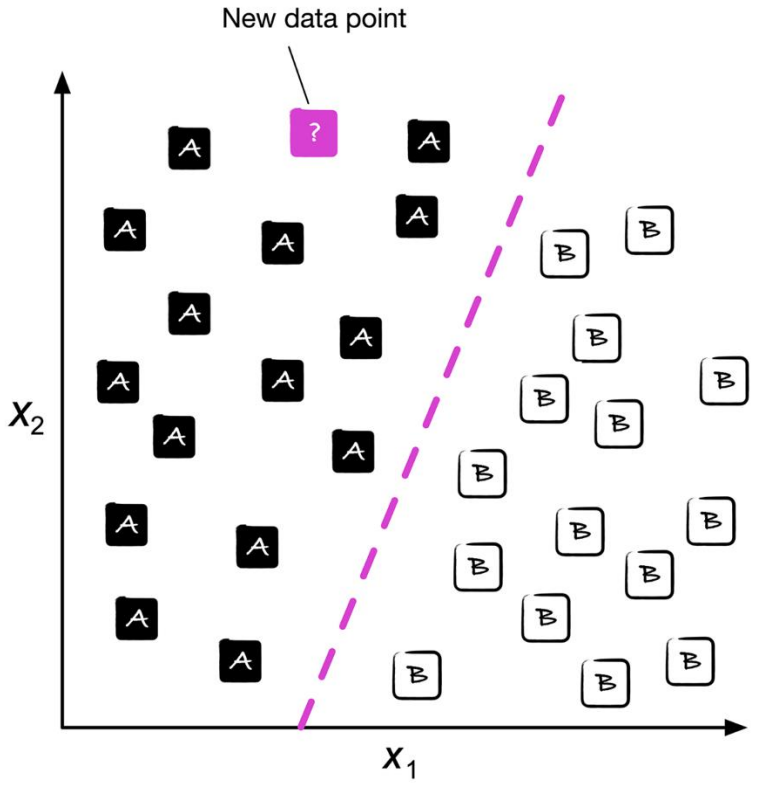
- Module A: Machine Learning
- Module B: Deep Learning
- Module C: Quantum Information
- Module D: Machine Learning for Quantum Physics

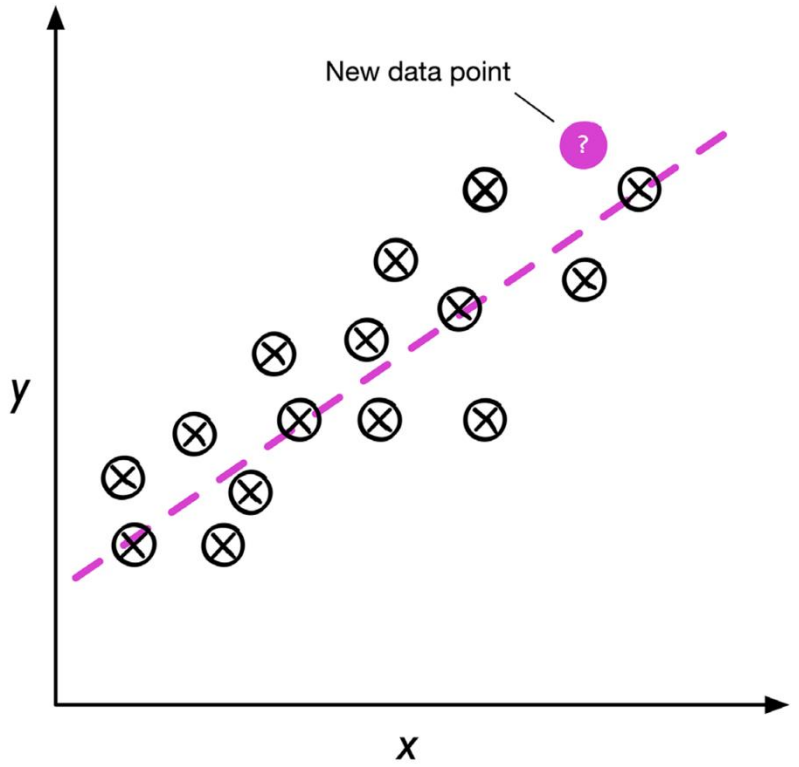
Three types of machine learning



Supervised learning

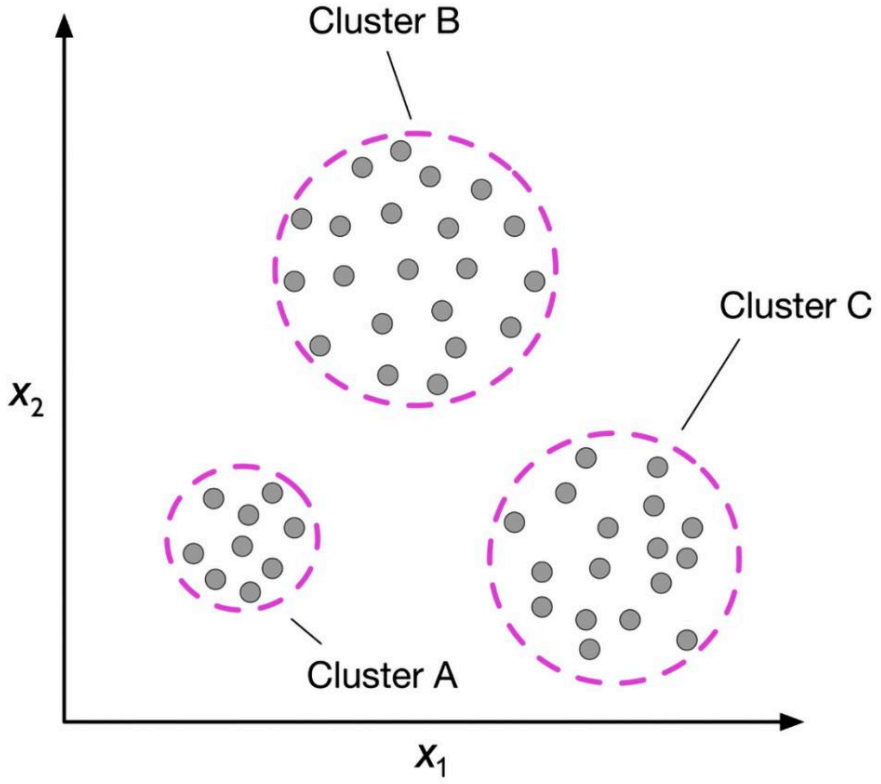
- **Classification:** labels are **discrete**, e.g., email spam detection is a binary classification task
- **Regression:** labels are **continuous**, e.g., house price vs. size

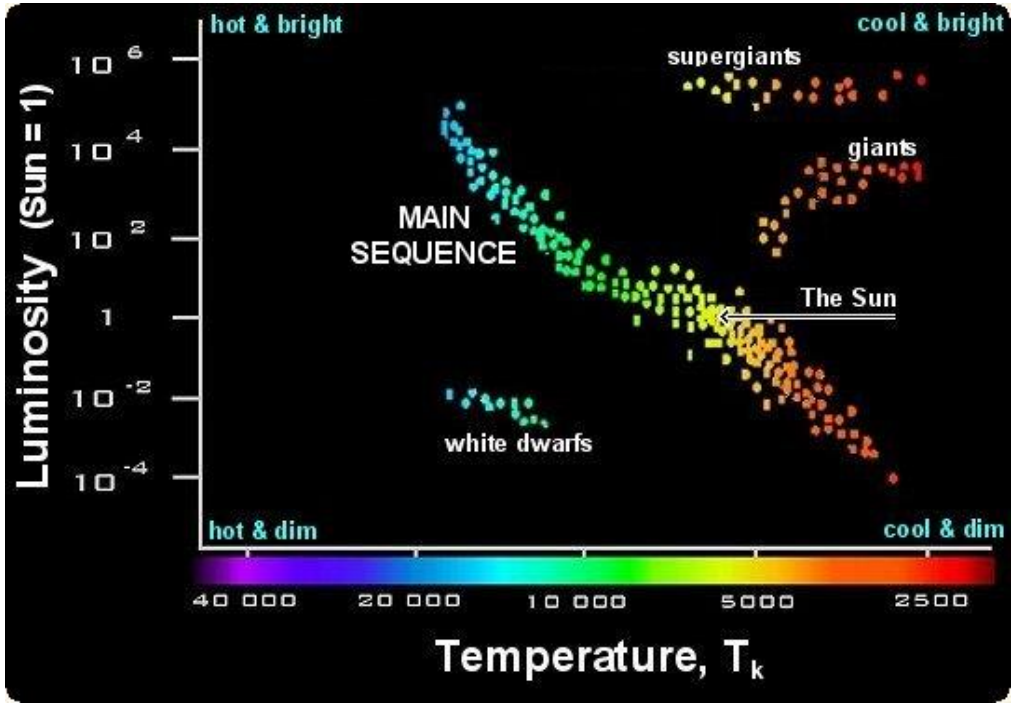




Unsupervised learning

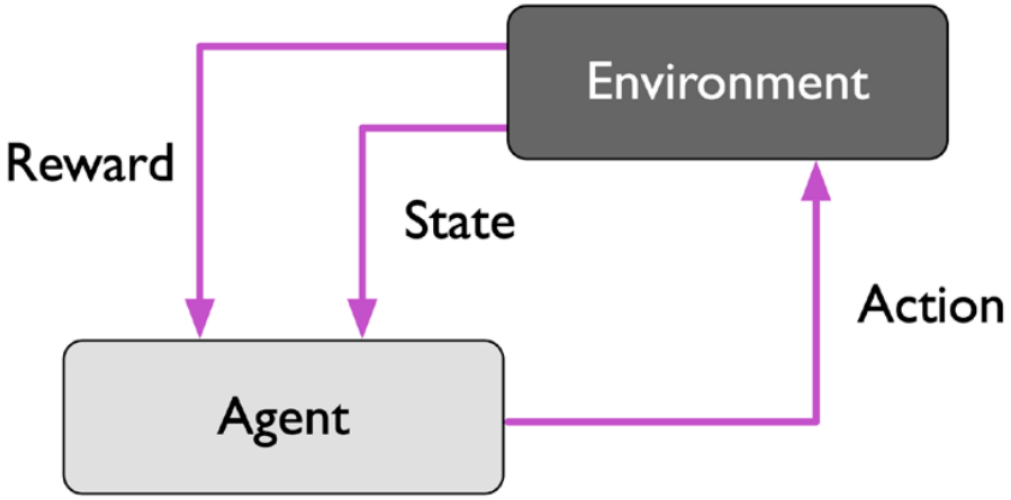
- **Clustering**: Discovering hidden structure of **unlabeled** data
- For example, the **Hertzsprung-Russell diagram** groups stars by temperature and luminosity

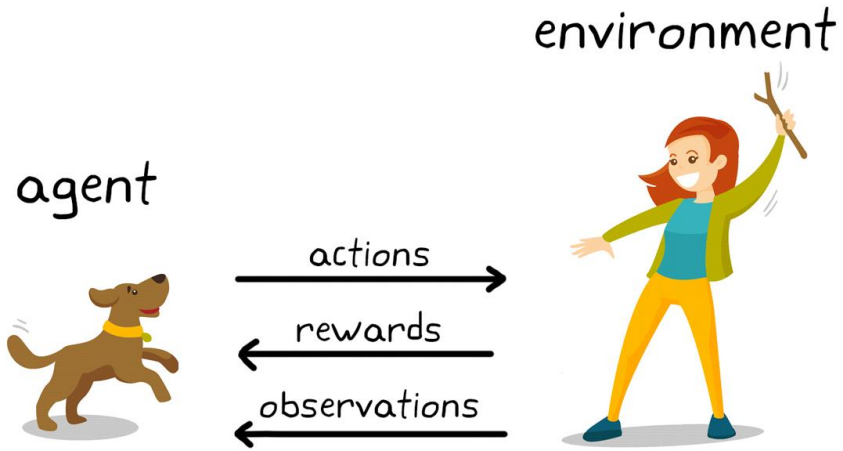




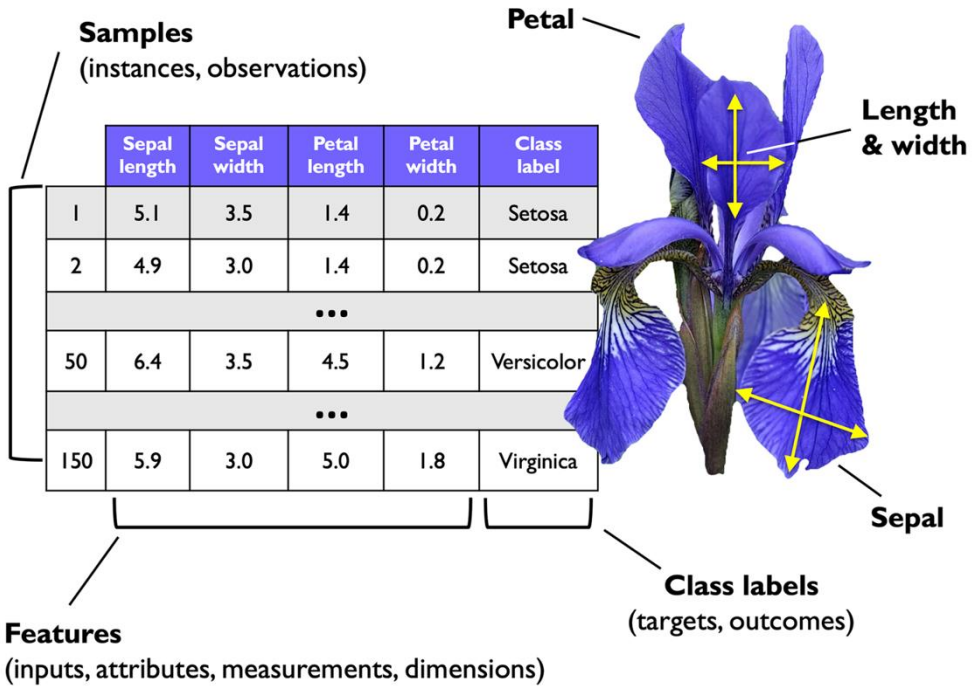
Reinforcement learning

- To develop a system (**agent**) that improves its performance based on interactions with the environment





Notation and Terminology



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

The Iris DataSet

- **4 Features**: Sepal length, Sepal width, Petal length, Petal width
- **150 Samples** or instances or observations, etc.
- **Class labels**: Setosa, Versicolor, Virginica.

Data Matrix

- Superscript = **sample** index = row index
- Subscript = **feature** index = column index

$$\boldsymbol{X} \in \mathbb{R}^{150\times 4}$$

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

sample vector

feature vector

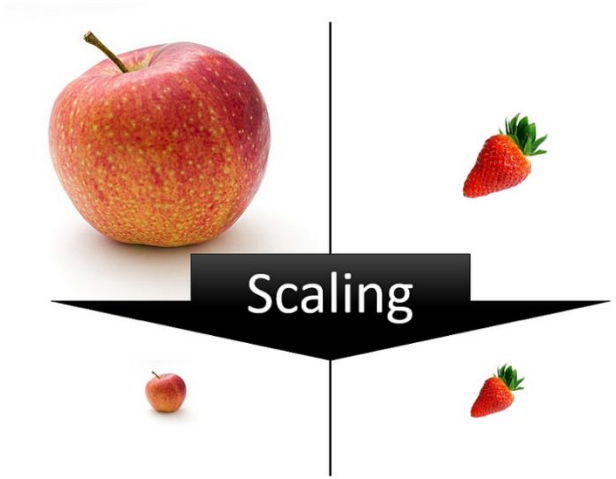
Terminology

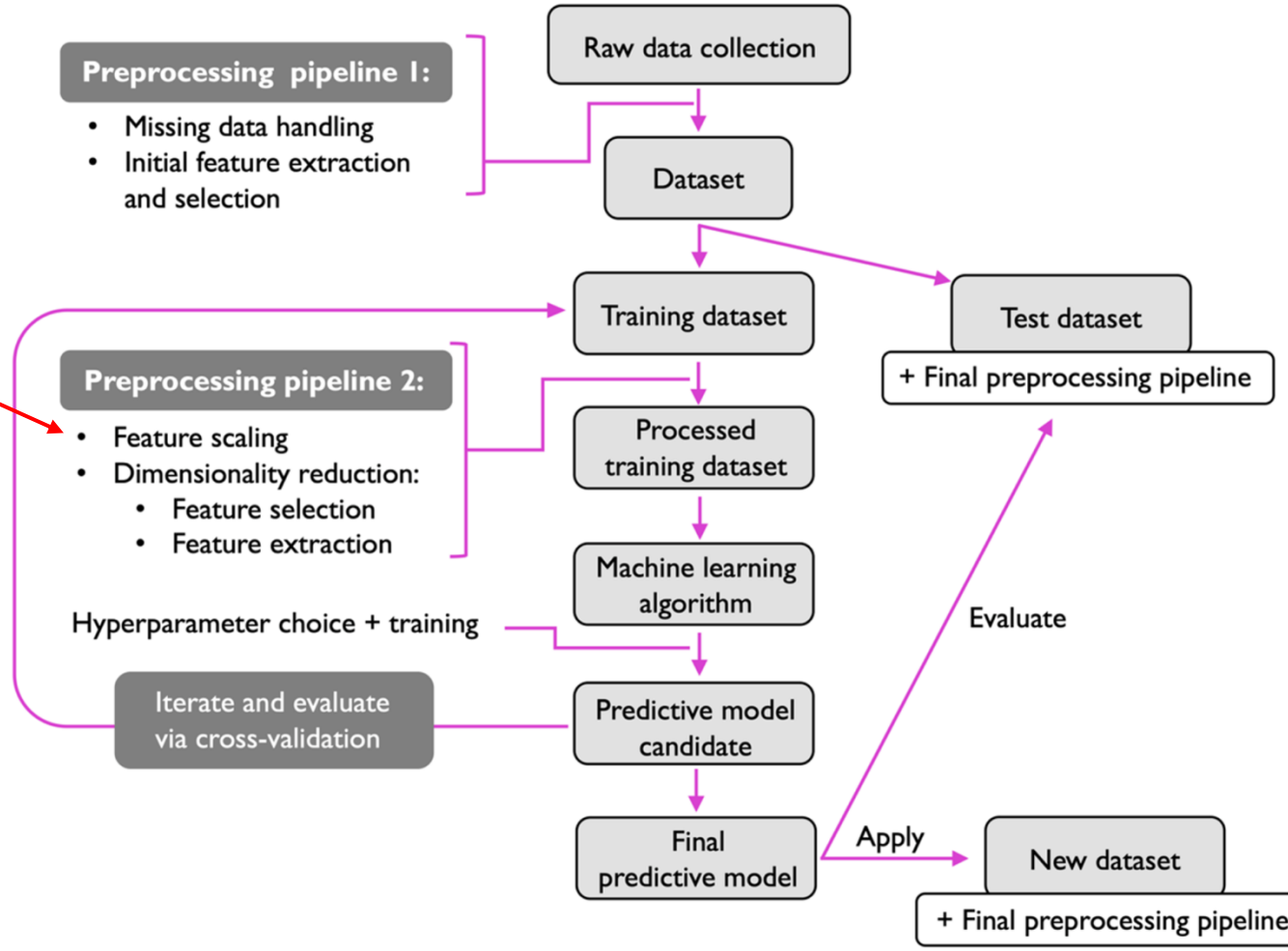
- **Training example**: a row in the data matrix, also known as an observation, record, instance, or sample
- **Feature**: a column in the data matrix, also known as predictor, variable, input, attribute
- **Target**: also known as class label, ground truth, outcome, output, etc.
- **Loss function**: also known as cost function or error function

ML typical workflow

- **Feature scaling**

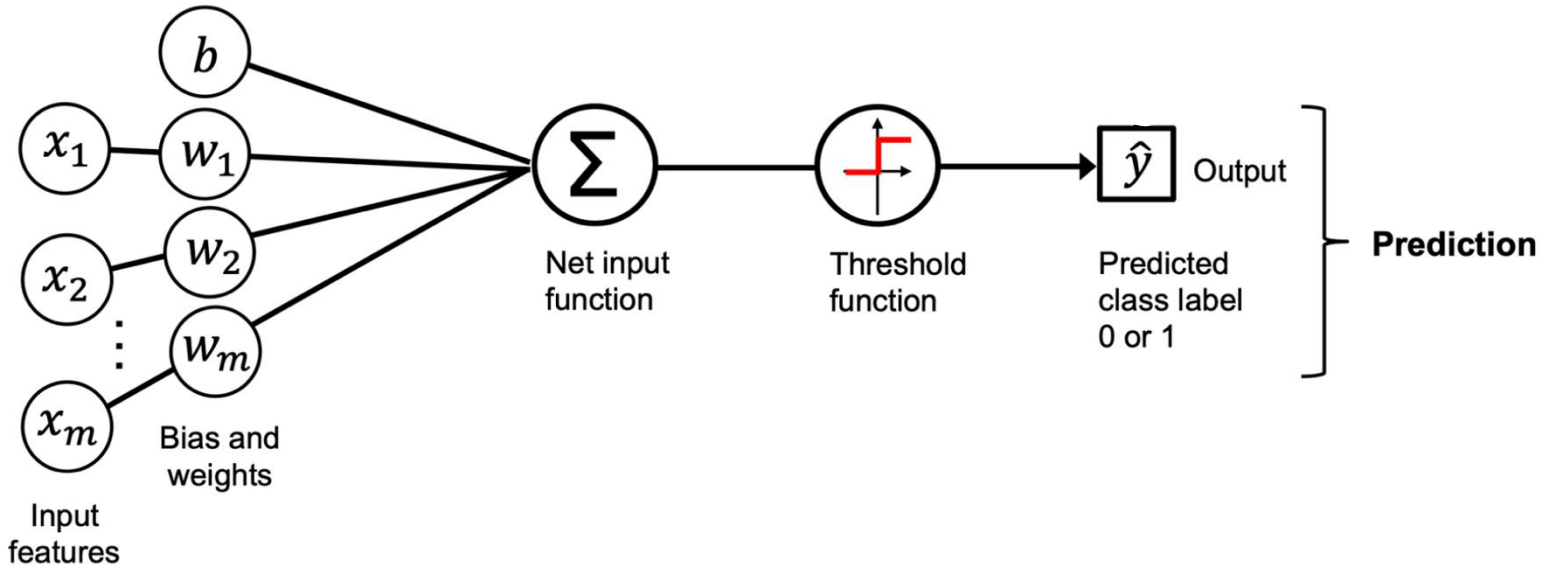
The features should be on the same scale for optimal performance. Normally, we transform it to a standard distribution with zero mean and unit variance.





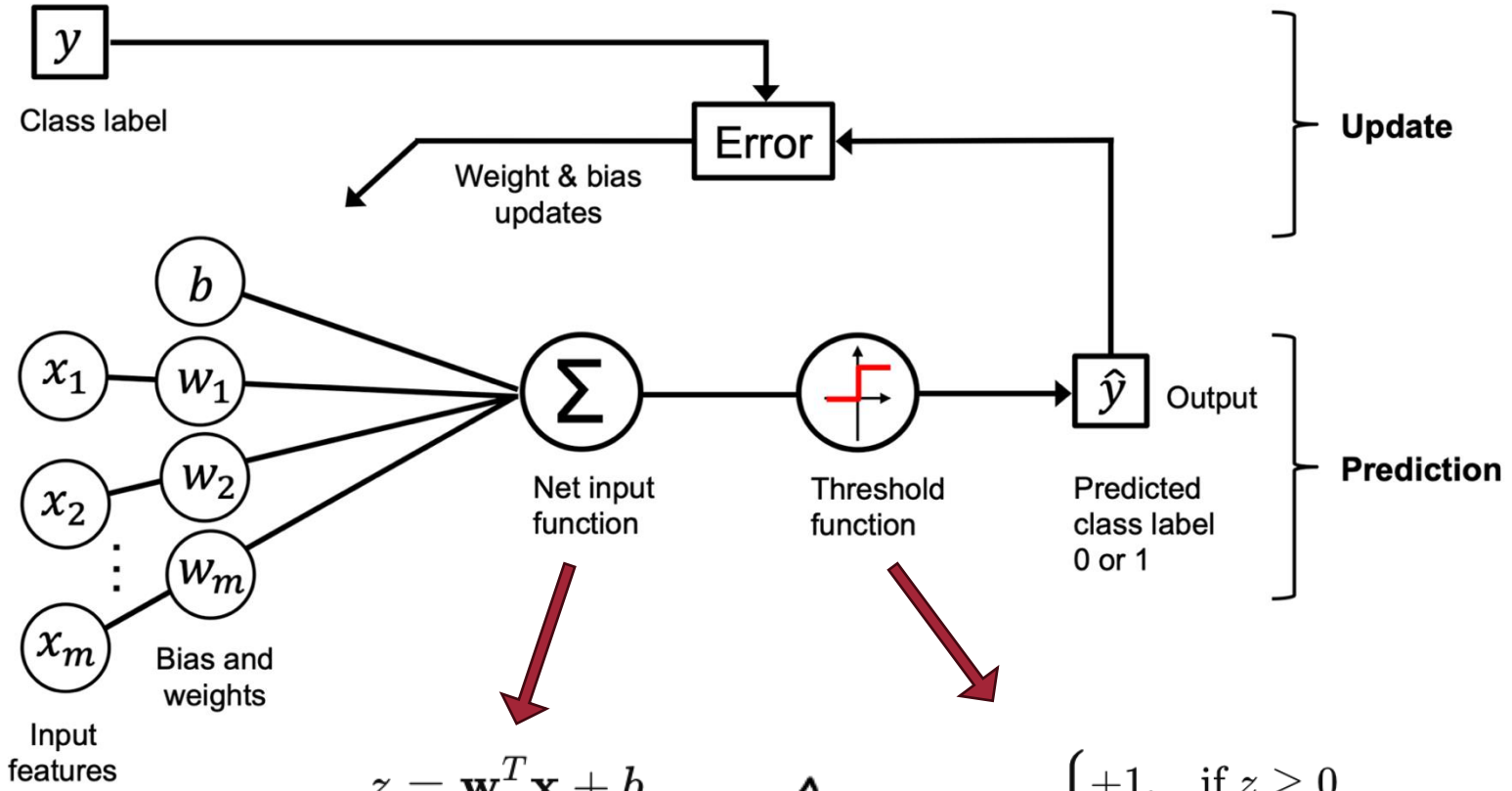
McCulloch-Pitts (MCP) neuron model

- **Pre-determined weights, no learning capability**



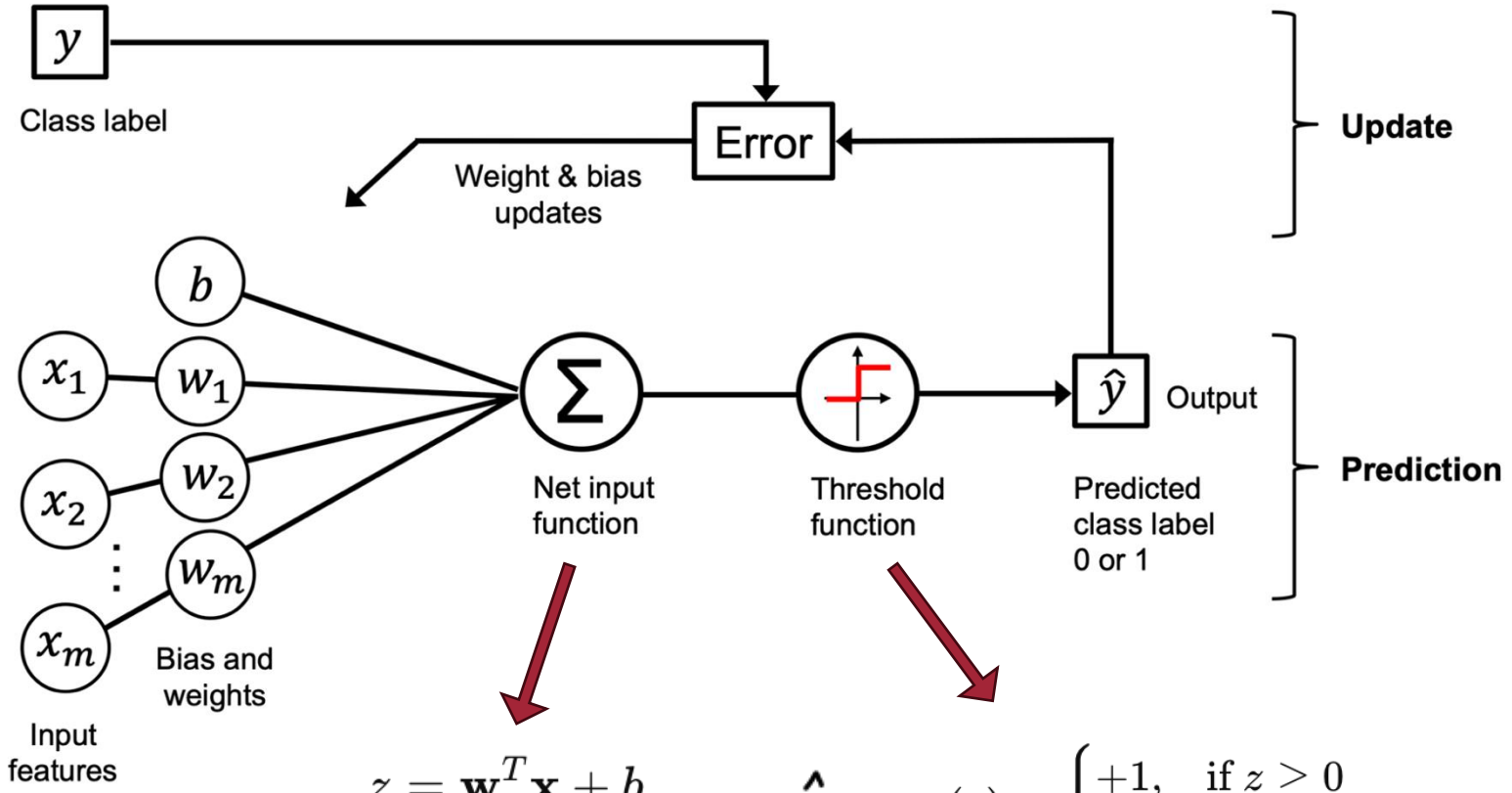
Rosenblatt's perceptron model

- Proposed an algorithm that would automatically **learn the optimal weight** coefficients



Key idea to adjust the weight (and bias)

- If predicted label is 1, but the actual label is 0, we want to reduce the weight
- If predicted label is 0, but the actual label is 1, we want to enhance the weight



The perceptron learning rule

1. Initialize the weights and bias unit to 0 or small random numbers
2. For each training example, $\boldsymbol{x}^{(i)}$:
    a. Compute the output value, $\hat{y}^{(i)}$
    b. Update the weights and bias unit

$$w_j := w_j + \Delta w_j$$
$$\text{and} \quad b := b + \Delta b$$

The update values ("deltas") are computed as follows:

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$
$$\text{and} \quad \Delta b = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)$$

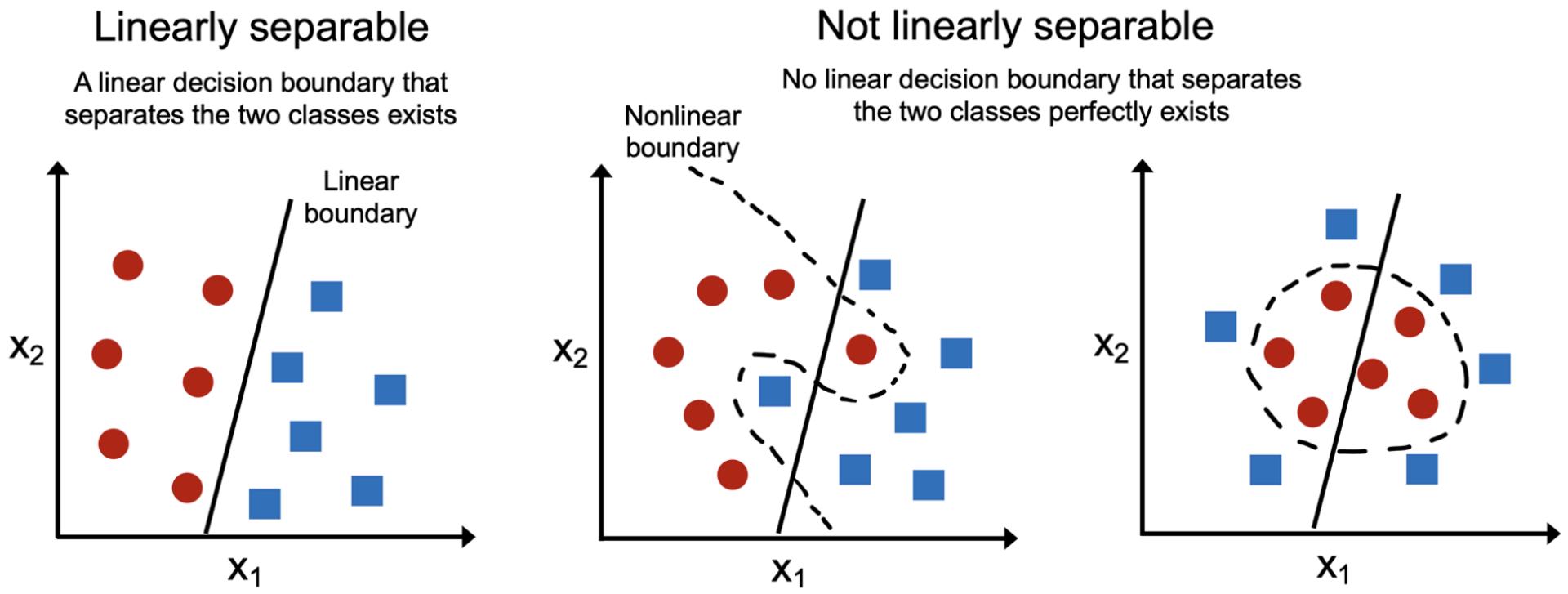
learning rate

actual class label

predicted class label

Applicable to linearly separable data only

- The algorithm finds the linear decision boundary after certain number of iterations (**epochs**)



Python basics

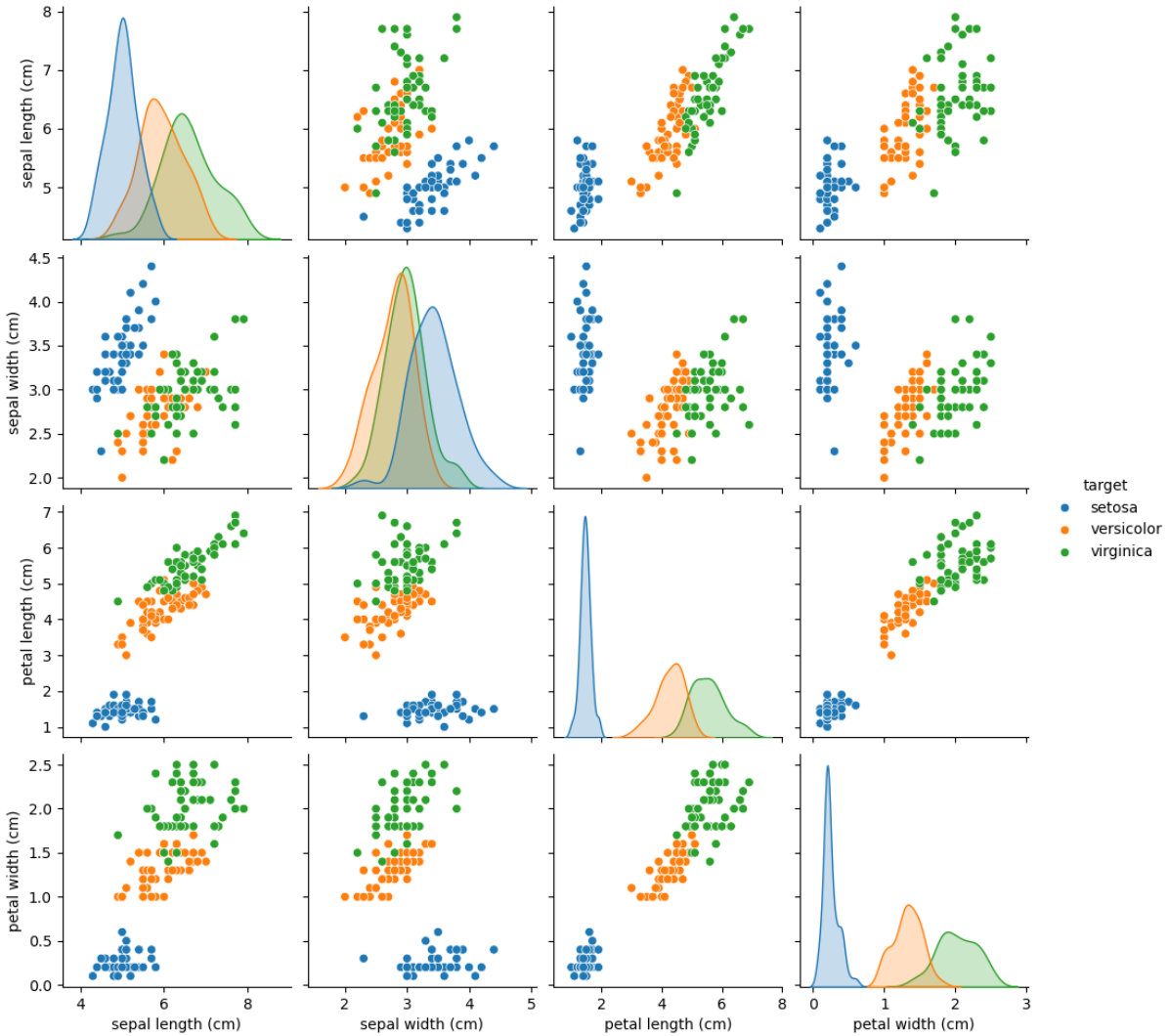
- See Jupyter Notebook: **Python_CheatSheet.ipyb**

- **Virtual environment** with conda
- **Jupyter Notebook**
- Essential packages: **numpy, matplotlib, pandas, seaborn, scipy**

Demo: Iris flowers classification

- See jupyter notebook: **demo_Iris.ipynb**

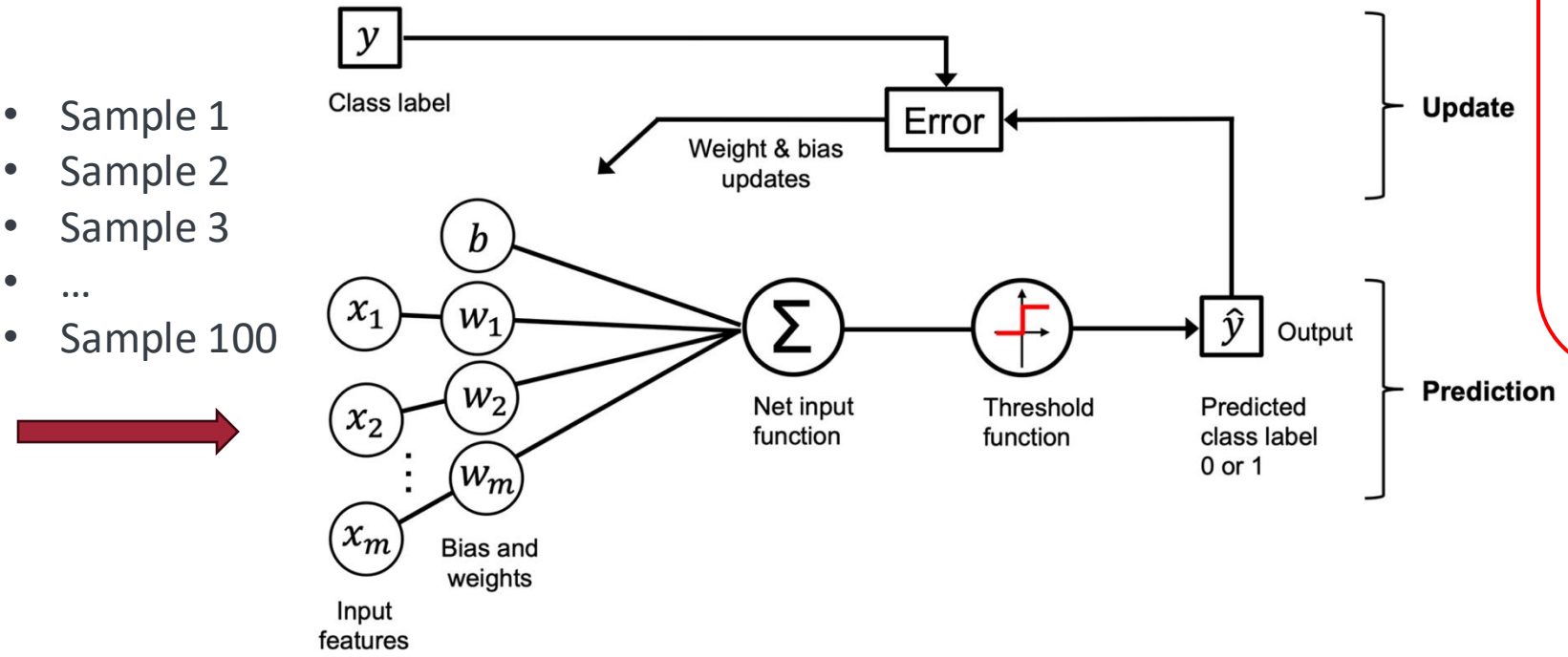
- Iris Dataset: **4 feature variables, 3 classes, 150 samples**
- Rosenblatt's model is specifically designed for **binary classification** tasks
- Need to **remove the data for one class** before we apply Rosenblatt Perceptron model



Rosenblatt perceptron

- **Single-layer** NN
- The weights are updated based on a **step function**
- The weight update is calculated incrementally after **EACH** training example

- Sample 1
- Sample 2
- Sample 3
- …
- Sample 100



$$w_j := w_j + \Delta w_j$$

and $b := b + \Delta b$

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

and $$\Delta b = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)$$

learning rate

predicted class label

actual class label

j: feature index

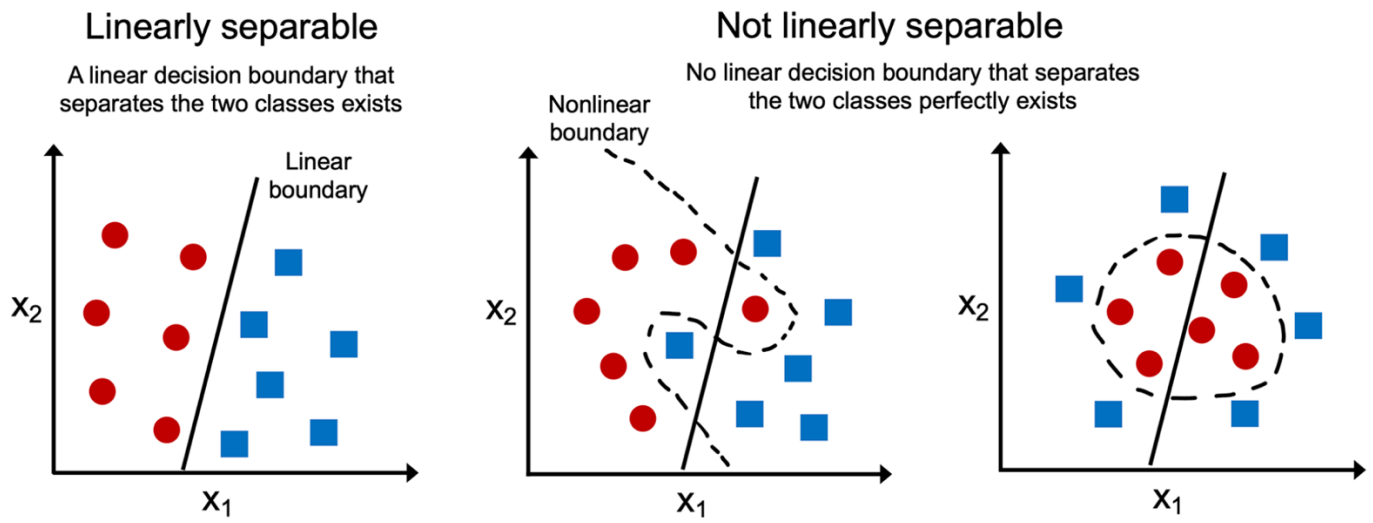
i: sample index

m=4 (features)

Perceptron convergence theorem

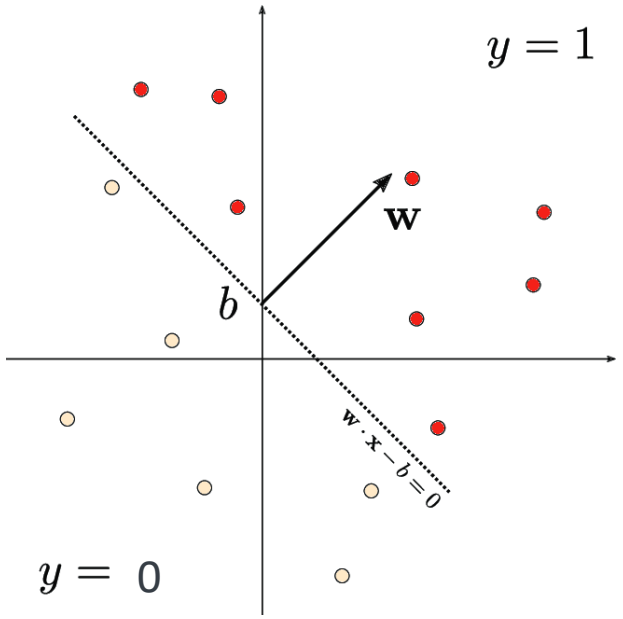
- Rosenblatt proved mathematically that the perceptron learning rule **converges** if the two classes can be **separated by a linear hyperplane.**

- If two classes cannot be separated by a linear hyperplane, the weights will never stop updating unless we set a maximum number of iterations (or epochs)



Geometric intuition

The weight vector is perpendicular to the decision boundary.



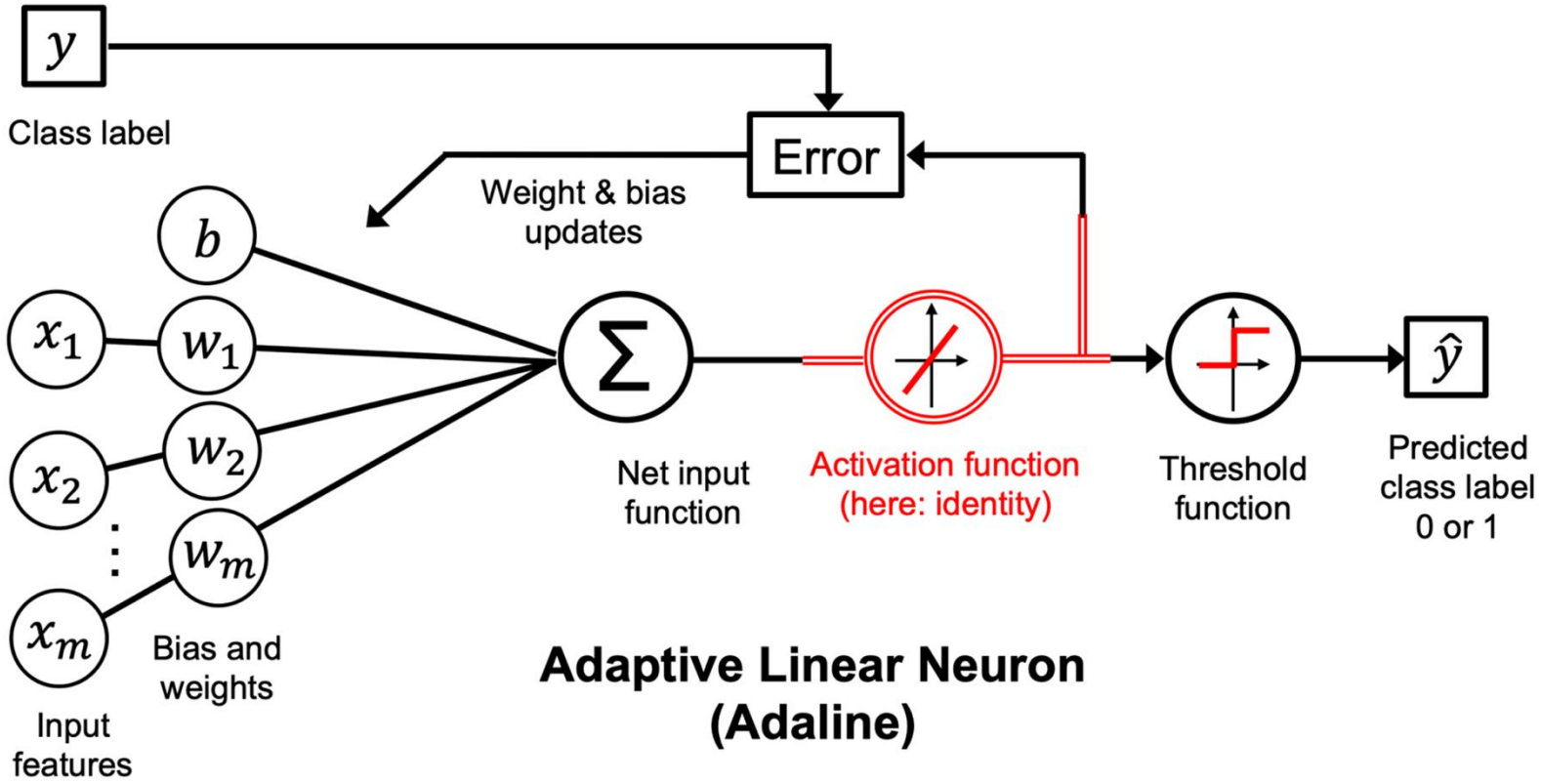
$$\hat{y} = \begin{cases} 0, & \mathbf{w}^T\mathbf{x} \leq 0 \\ 1, & \mathbf{w}^T\mathbf{x} > 0 \end{cases}$$

$$\mathbf{w}^T\mathbf{x} = ||\mathbf{w}|| \cdot ||\mathbf{x}|| \cdot \underbrace{\cos(\theta)}$$

So this needs to be 0 at the boundary, and it is zero at 90°

Adaptive linear neuron (Adaline)

- A generalized Rosenblatt's neuron model by Bernard Widrow and Tedd Hoff (**1960**)



Key difference

Learning

$$\sigma(z) = z$$

- In the Adaline rule, the weights are updated based on a **linear activation function** rather than a step function
- The weight update is calculated based on **all samples** in the training dataset (instead of updating the parameters incrementally after each training sample)
- It is referred to as **full batch gradient descent**

Prediction

- While the linear activation function is used for learning the weights, we still use a **step function** to make the final prediction

Adaline learning: Gradient descent

- To minimize the **objective function**, or loss or cost function

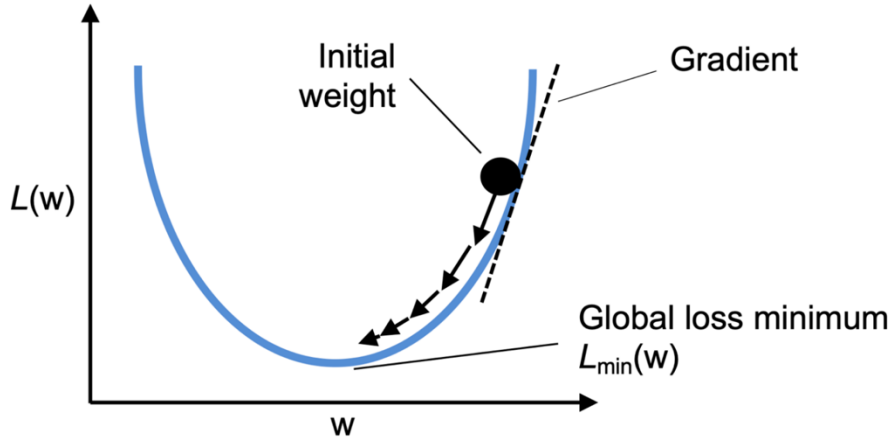
- **Mean squared errors (MSE)**

weighted sum of i-th feature vector

$$L(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)^2$$

Sum over all features

actual class label



- **Adaline learning or updating rule**

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}, \quad b := b + \Delta b$$

$$\Delta w_j = -\eta\frac{\partial L}{\partial w_j} \text{ and } \Delta b = -\eta\frac{\partial L}{\partial b}$$

$$\frac{\partial L}{\partial w_j} = -\frac{2}{n}\sum_{i}\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)x_j^{(i)}$$

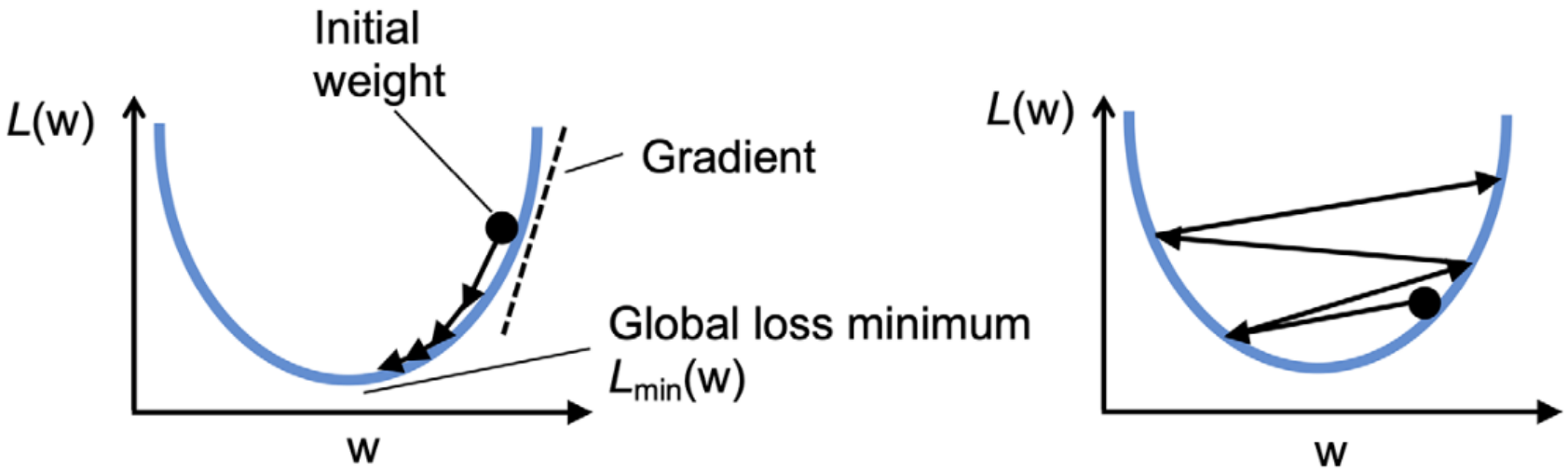
$$\frac{\partial L}{\partial b} = -\frac{2}{n}\sum_{i}\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)$$

Advantages of this MSE loss function

i) Differentiable

ii) It is convex; thus a local or global minimum can be reached by climbing down the hill (along the negative direction of the gradient)

Learning rate



- If we choose a learning rate that is too large --- we **overshoot** the global minimum
- If it is too small --- training will be slow and might get stuck in local minima (for complex loss function). However, MSE loss function is convex and there are no local minima.

Python implementation of Adaline

- See Jupyter notebook: **demo_iris_Adaline.ipynb**

Feature scaling

- Many ML algorithms require feature scaling for optimal performance
- Gradient descent is one of the them that benefit from feature scaling. Other algorithms, such as regularization and k-means, also strongly depend on feature scaling. While the decision trees and random forests don't need to worry about feature scaling.
- **Standardization**

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

$$x_{norm}^{(i)} = \frac{x^{(i)} - \mathbf{x}_{min}}{\mathbf{x}_{max} - \mathbf{x}_{min}}$$

standardization

min-max scaling ("normalization")

	input	standardized	normalized
0	0	-1.46385	0.0
1	1	-0.87831	0.2
2	2	-0.29277	0.4
3	3	0.29277	0.6
4	4	0.87831	0.8
5	5	1.46385	1.0

- **Normalization**
- After feature scaling, it is **easier to find a learning rate that works well for all weights** (and bias).

Stochastic gradient descent

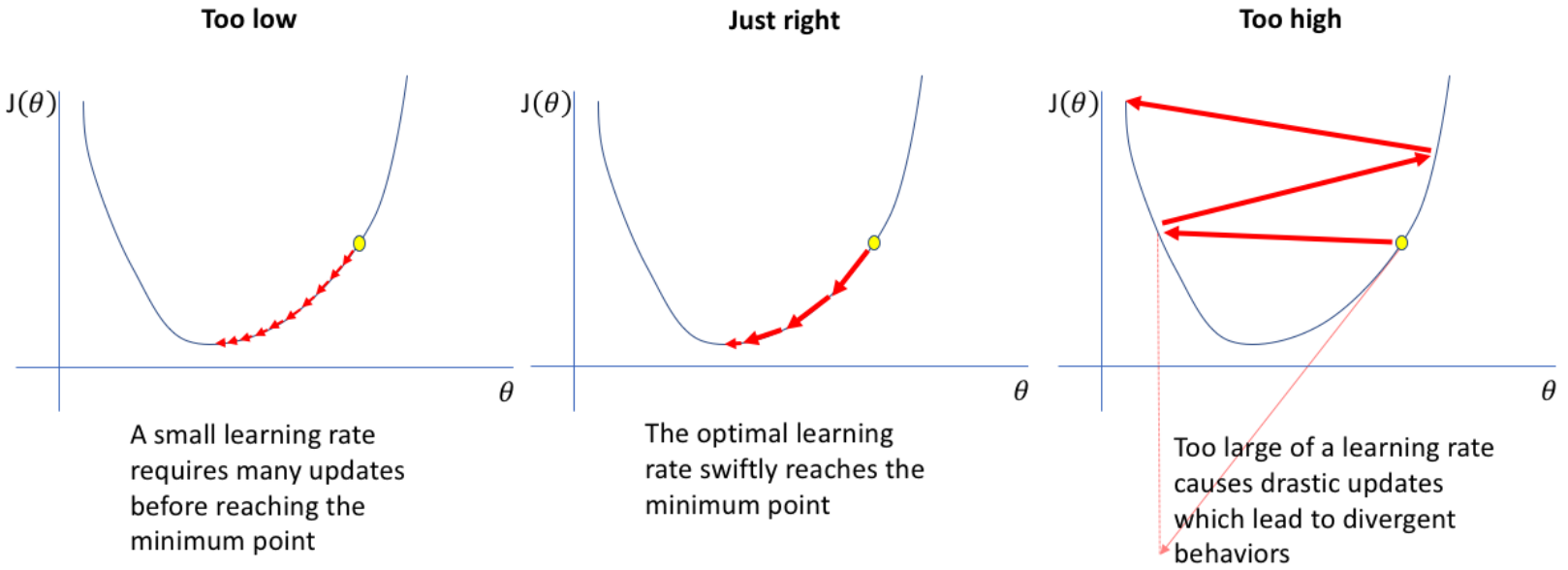
- For very large dataset with millions of data points, full batch gradient descent can be computationally expensive
- Instead of updating the weights based on the sum of the **accumulated errors over all training sample**, we update the parameters **incrementally for each training sample --- SDG**
- **Or use mini-batch gradient descent – apply full batch gradient to smaller subset of the training data.**
- Compared to SGD, we can replace the for loop over the training examples with vectorized operations, which can further improve the computational efficiency.

Adaptive learning rate

In SGD implementations, the fixed learning rate, η, is often replaced by an adaptive learning rate that decreases over time

$$\frac{c_1}{[\text{number of iterations}] + c_2}$$

where c1 and c2 are constants.
SGD does not reach the global loss minimum but an area very close to it.



Python demos

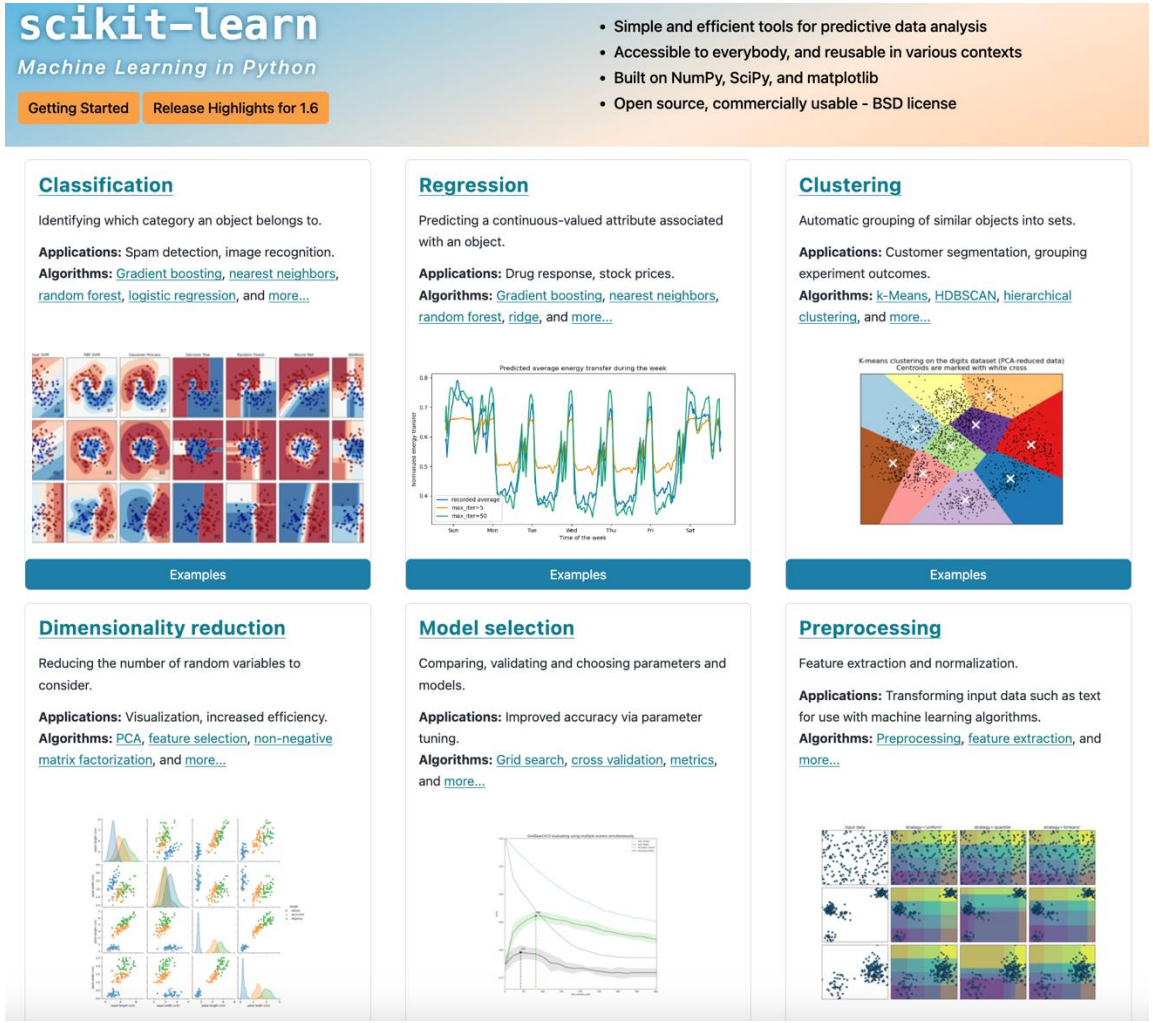
- See Jupyter notebook

Scikit-learn and other ML algorithms

- Scikit-learn API
- Logistic regression
- Support vector machine
- Slack variable
- Decision tree learning
- Ensemble method such as random forests
- K-nearest neighbors

Scikit-learn

- Offers a user-friendly and consistent interface for using popular ML algorithms efficiently and productively



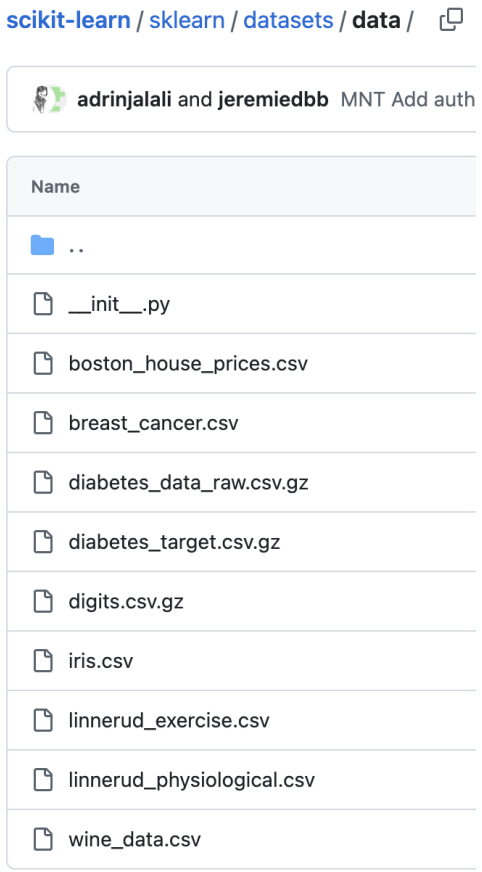
https://scikit-learn.org/stable/

- ✓ Classification
- ✓ Regression
- ✓ Clustering
- ✓ Decision trees
- ✓ Ensemble methods
- ✓ Nearest neighbors
- ✓ Support vector machines
- ✓ Dimensionality reduction
- ✓ Model selection
- ✓ Preprocessing

pip install scikit-learn

Datasets in scikit-learn

- Small Toy Datasets (CSV-based datasets)



Use load_*() to access

- Scikit-learn can also fetch larger real-world datasets from external sources (e.g., OpenML, UCI, or LIBSVM) via fetch_*() functions

- Additionally, scikit-learn provides functions to generate artificial datasets

Load and extract Iris-flower dataset

- Instead of implementing the perceptron rule and Adaline in Python and Numpy by ourselves, we use the scikit-learn library

Assign the petal length and petal width of the 150 flower examples to the feature matrix

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
>>> print('Class labels:', np.unique(y))
Class labels: [0 1 2]
```

Assign the three class labels: Iris-setosa, Iris-versicolor, and Iris-virginica, to label vector

Return the three labels as integers
(a common convention among most ML libraries)

Separate the training and test examples

- To evaluate how well a trained model performs on **unseen data**, split the dataset into separate **training and test datasets**

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=1, stratify=y
... )
```

- Import the **train_test_split** function, from the **model_selection** module, which is a part of the **scikit-learn** library
- The **train_test_split** function shuffles the dataset internally before splitting
- The **number of test dataset** is test_size*total number of dataset = 0.3*150=45
- **random_state=1**: to provide a fixed random seed for internal pseudo-random number generator (which is used to shuffle the dataset) → useful to reproduce the results in the future

Feature scaling

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

standardization

- **Feature scaling** with **StandardScalar** class from scikit-learn's **preprocessing** module

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Initialize a new StandardScalar **object** and assign it to the sc variable

- Using the **fit** method, the sample ***mean*** and ***standard deviation*** are estimated, for each feature variable
- Using the **transform** method, we standardize the data using the estimated mean and std.

Training a perceptron

- Now train a perceptron model
- Scikit-learn supports multiclass classification [via the one-versus-rest (OvR) method]

```
>>> from sklearn.linear_model import Perceptron
>>> ppn = Perceptron(eta0=0.1, random_state=1)
>>> ppn.fit(X_train_std, y_train)
```

- Import the **Perceptron** class from the **linear_model** module
- Initialize the new Perceptron object (and provide the learning rate 0.1 and a random seed 1)
- The **fit** method trains the model

Predictions and performance metrics

- Make predictions via the **predict** method

```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Misclassified examples: %d' % (y_test != y_pred).sum())
Misclassified examples: 1
```

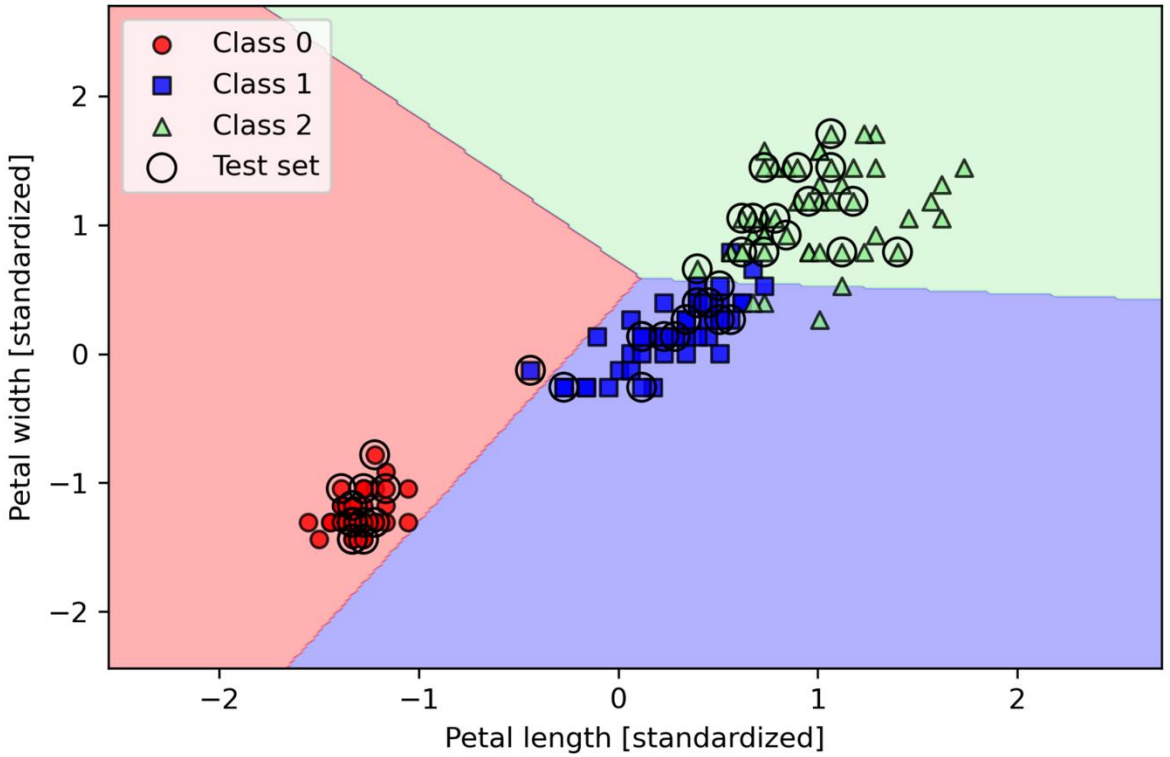
The **misclassification error** on the test data set is 1/45, approximately 2.2%
Or
The **classification accuracy** is 1 - error = 97.8%

- You can also use the **metrics** module

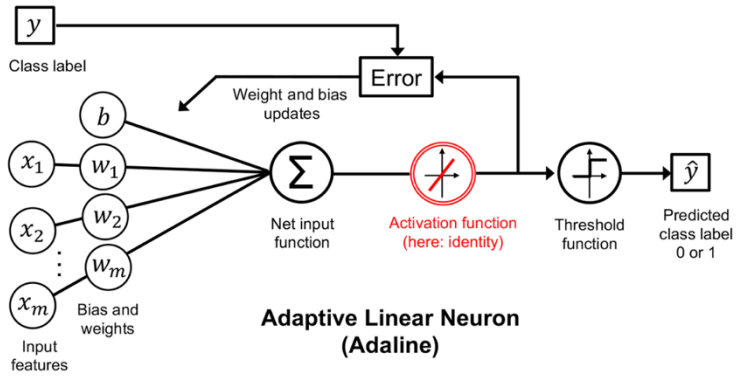
```
>>> from sklearn.metrics import accuracy_score
>>> print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
Accuracy: 0.978
```

Visualization

- Visualizing the decision boundaries of a multi-class perceptron model
- See demo code: **demo_sklearn_perceptron.ipynb**

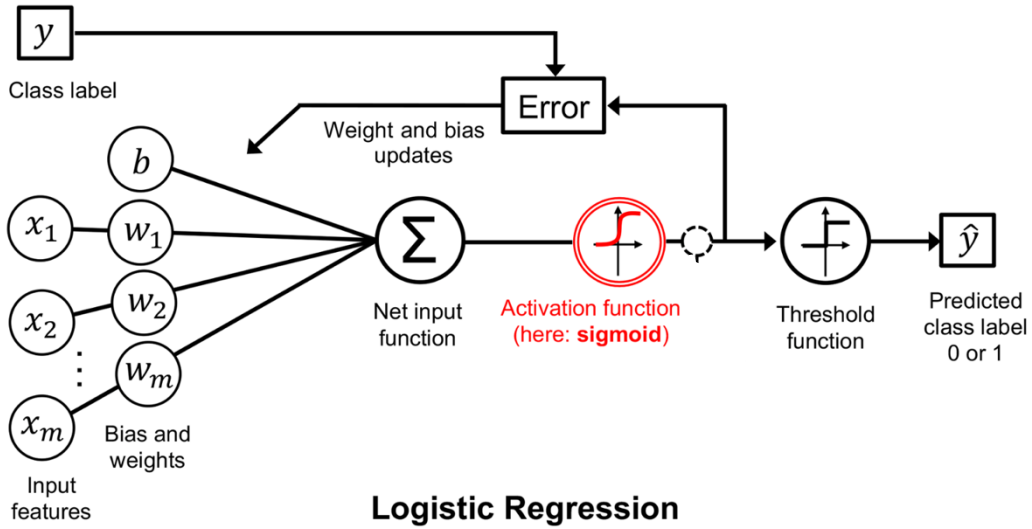


Logistic regression



Adaptive Linear Neuron (Adaline)

$$z = \mathbf{w}^T\mathbf{x} + b$$

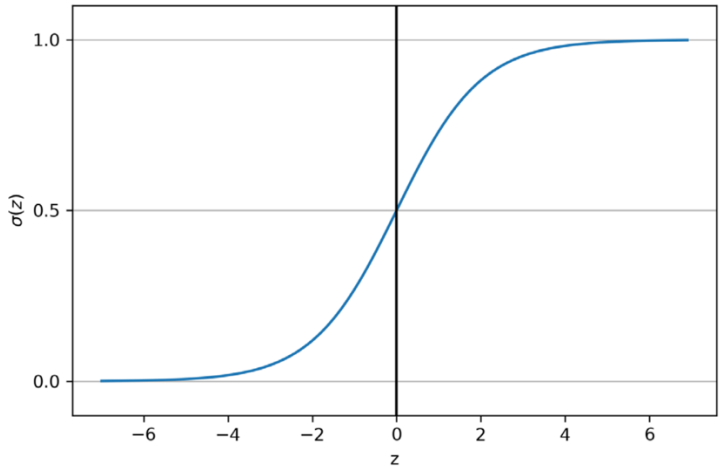


Logistic Regression

- More accurately, logistic regression should be called logistic classification as we apply it to classify datasets

- Logistic sigmoid function or sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

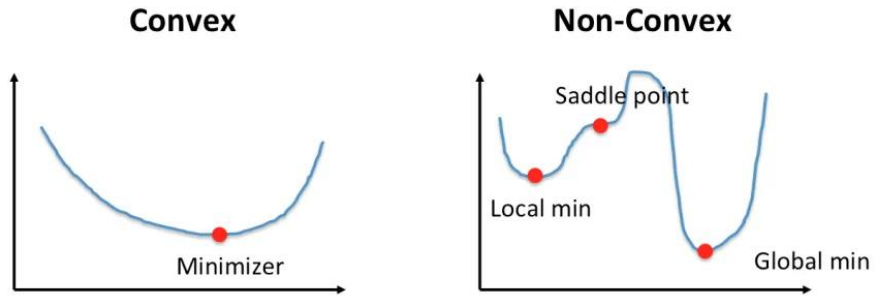


- It represents the probability of a particular example belong to class 1
- After the optimization of the weights, the threshold function further converts the probability to binary classes 1 or 0

The loss function

- The **Mean-Squared-Error (MSE)** loss function

$$L(\boldsymbol{w}, b|\boldsymbol{x}) = \sum_i \frac{1}{2}\left(\sigma\left(z^{(i)}\right) - y^{(i)}\right)^2$$



- Predicted probability (output of the sigmoid function)
- Due to the involvement of sigmoid function, the MSE is **non-convex**
  The second derivative changes sign
- The gradient descent method can get stuck in local minima
- The gradient of MSE tends to saturate, leading to slow learning

The loss function

- **The negative log-likelihood (or binary cross-entropy) loss function**

$$L(\boldsymbol{w}, b) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\sigma\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \sigma\left(z^{(i)}\right)\right) \right]$$

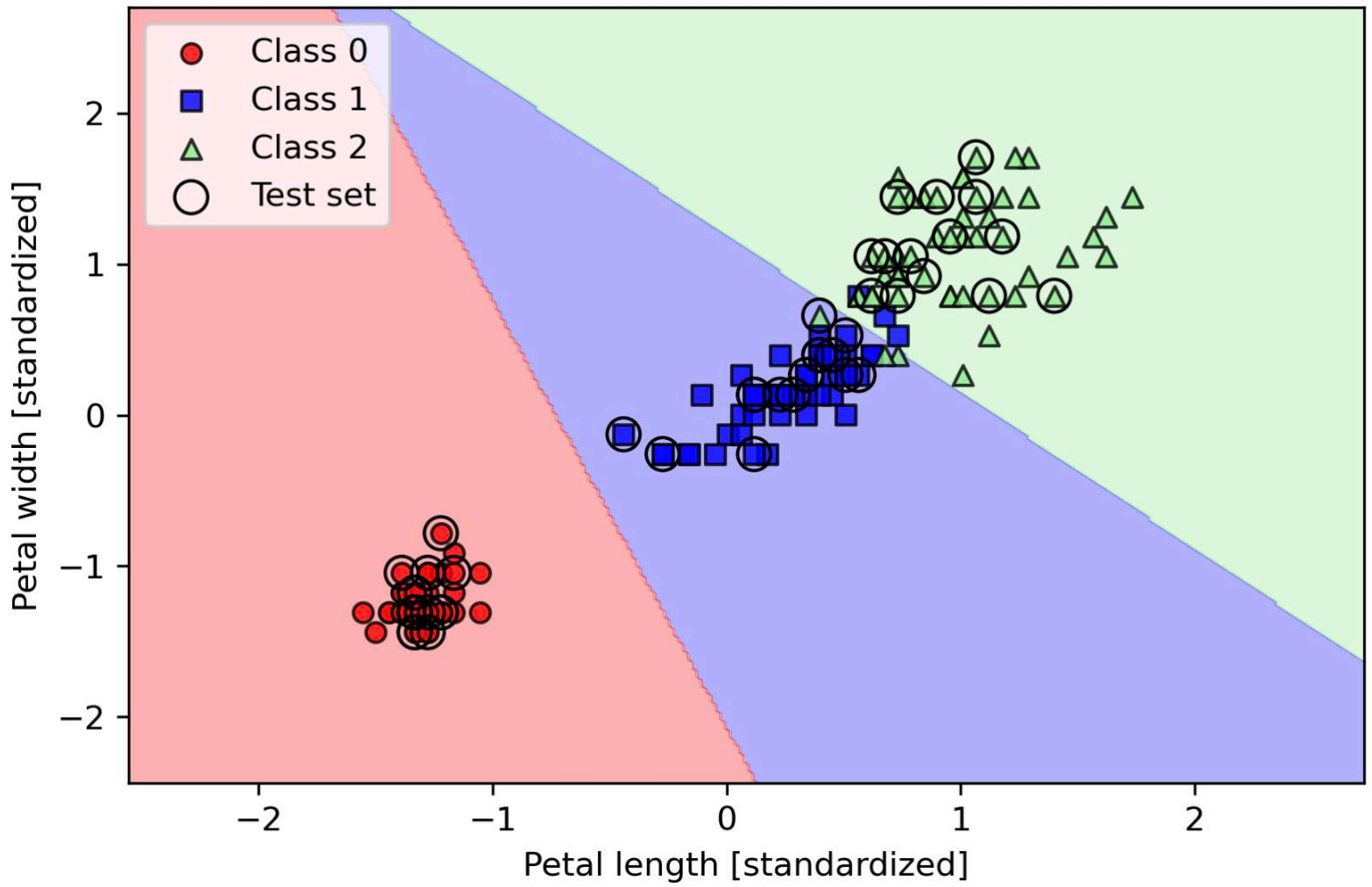
Unlike MSE, log-likelihood loss is convex, ensuring gradient descent finds the global minimum

Training a logistic regression model with scikit-learn

- Homework # 2

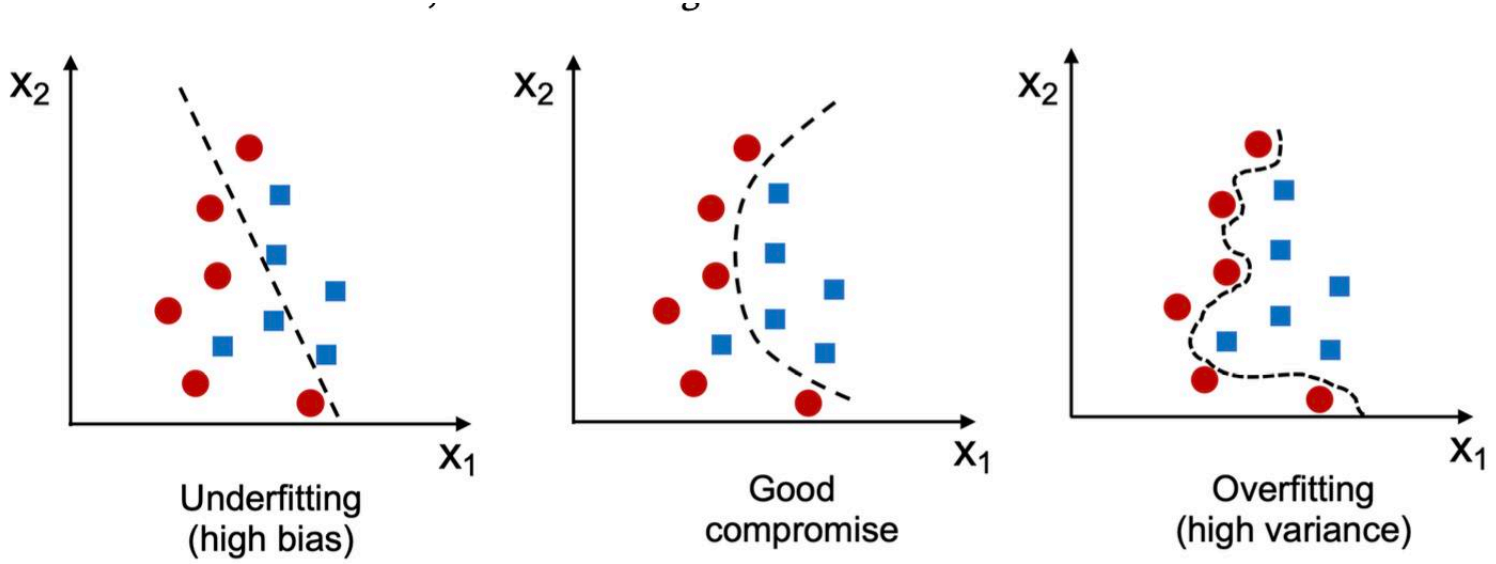
```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=100.0, solver='lbfgs',
...                         multi_class='ovr')
>>> lr.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined,
...                       classifier=lr,
...                       test_idx=range(105, 150))
>>> plt.xlabel('Petal length [standardized]')
>>> plt.ylabel('Petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

Visualizing the decision boundary



Overfitting

- **Overfitting** is common in ML: the model performs well on training data, but does not generalize well to unseen data (test data)
- The model has a high variance: too complex to generalize
- **Underfitting**: the model has high bias, the model is not complex enough to capture the pattern



bias-variance tradeoff

regularization

- One way of finding a good bias-variance tradeoff is to tune the complexity of the model via regularization
- Adding a penalty term in the loss function
- **L2 regularization**

$$L(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left[-y^{(i)}\log\left(\sigma\left(z^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-\sigma\left(z^{(i)}\right)\right)\right] + \frac{\lambda}{2n}\|\boldsymbol{w}\|^2$$

λ: regularization parameter.

Note: in scikit-learn's logistic regression function, the parameter C=1/ λ is the inverse of the regularization parameter

```
lr = LogisticRegression(C=10.**c,
                        multi_class='ovr')
lr.fit(X_train_std, y_train)
```