PATTERN RECOGNITION USING PYTHON

Classification

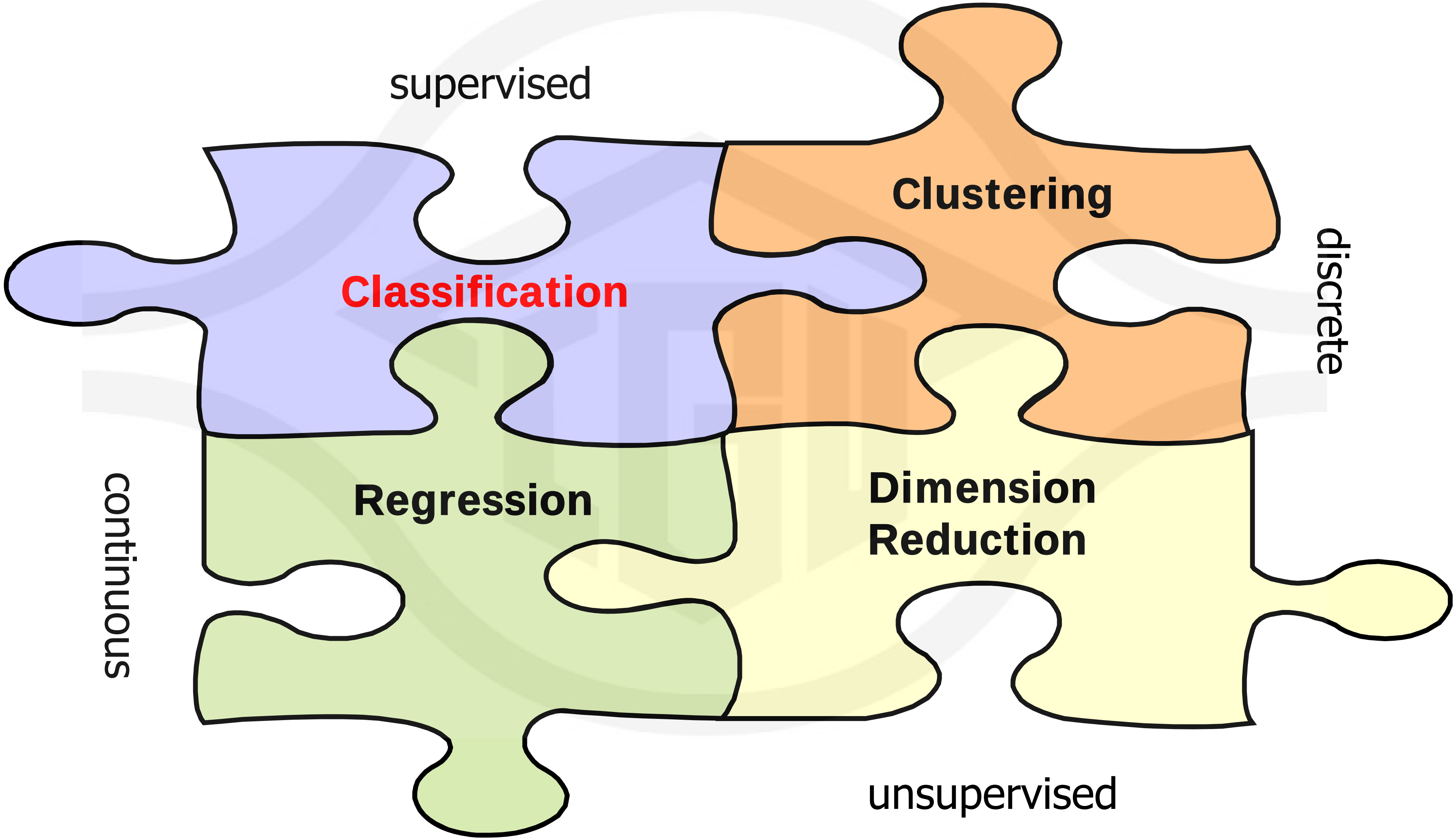
Wen-Yen Hsu
Dept Electrical Engineering
Chang Gung University, Taiwan

2019-Spring

Supervised Learning Algorithms for Classification

- Popular algorithms for classification, such as logistic regression, support vector machines, and decision trees
- Examples and explanations using the scikit-learn machine learning library, which provides a wide variety of machine learning algorithms via a user-friendly Python API

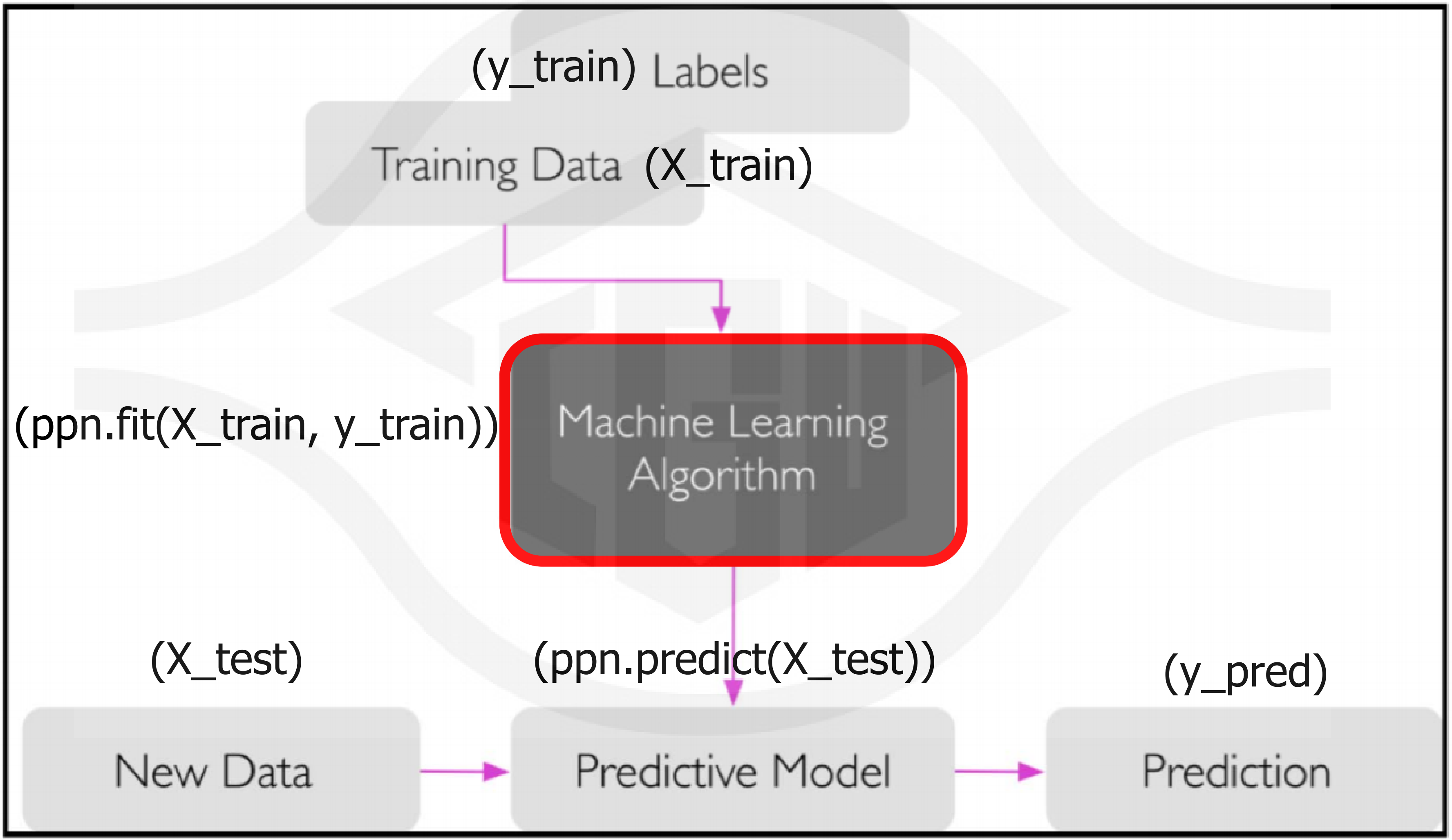
Machine Learning Organizational Chart



Choosing a Classification Algorithm

- Selecting features and collecting training samples
- Choosing a performance metric
- Choosing a classifier and optimization algorithm
- Evaluating the performance of the model
- Tuning the algorithm

Solve Classification Problem



Problem Transformation

Machine Learning Algorithm

Classification Problem

evaluate **error** (when label ≠ prediction)
by **cost function** (objective function)

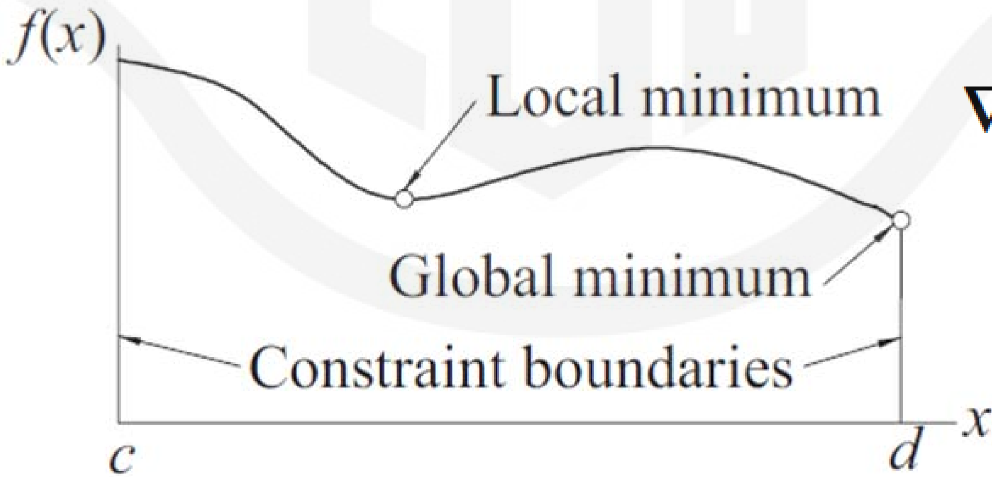
Optimization Problem

Optimization Problem

- In engineering, optimization is closely related to design.
- We wish to keep it **(objective function) as small as possible**, such as the cost or weight.

$$\boxed{\text{Find } \mathbf{x} \text{ that minimizes } F(\mathbf{x}) \text{ subject to } g(\mathbf{x}) = 0,\ h(\mathbf{x}) \geq 0.}$$

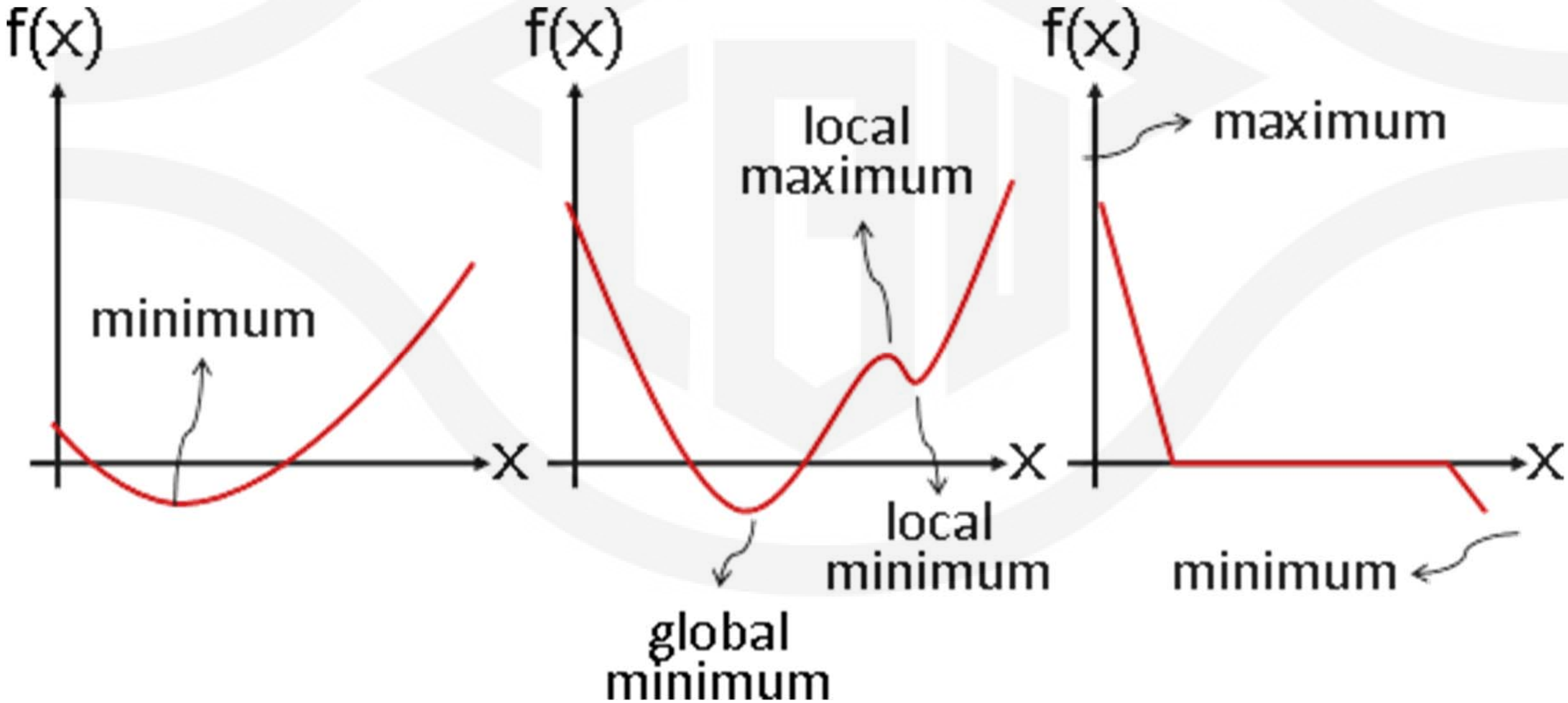
- Find the points where the gradient vector of F(x) vanishes



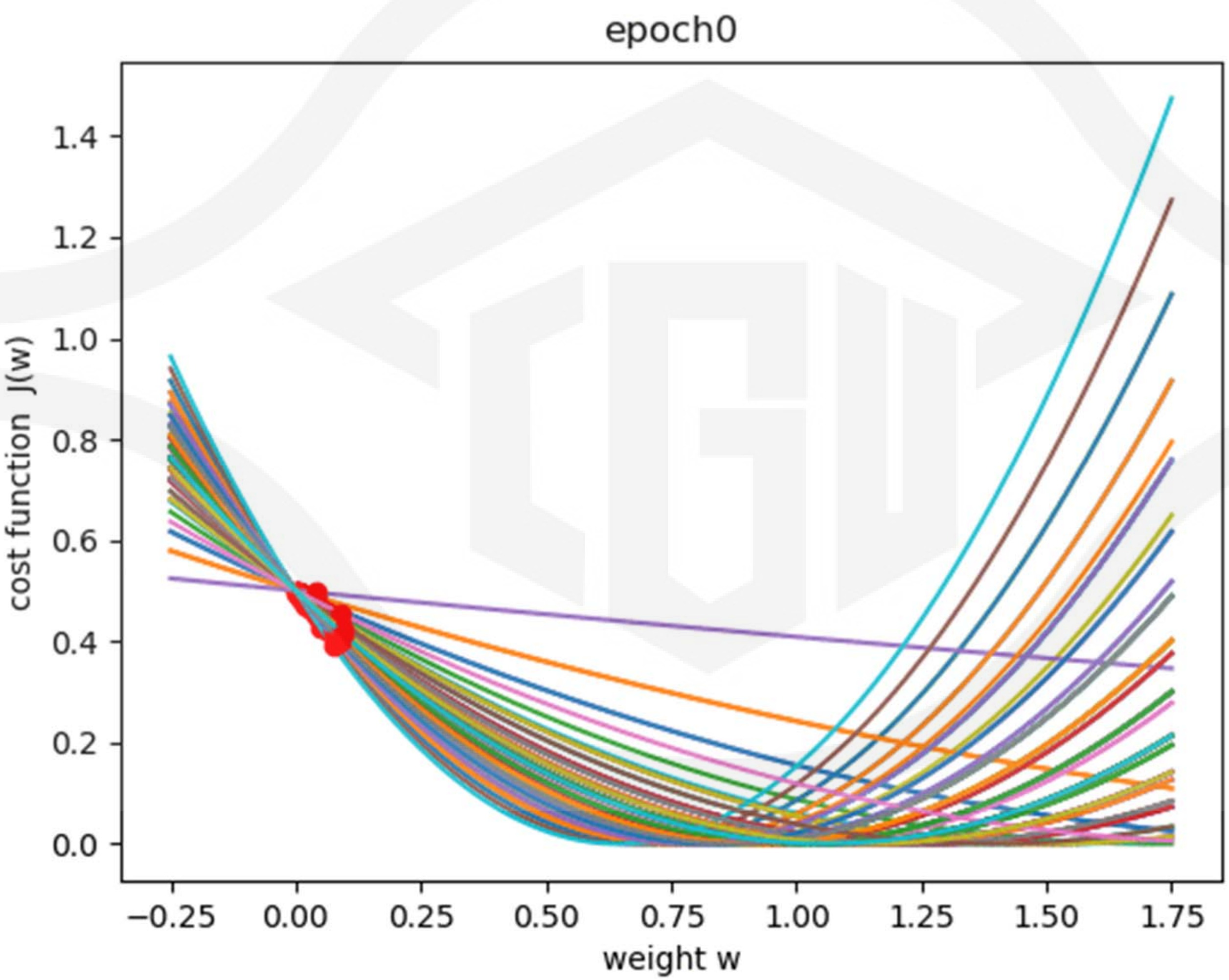
$$\nabla F(\mathbf{x}) = \mathbf{0}$$

Algorithms for Minimization

- It require **starting values (self.w_)**
- Processes by **iterative procedures (n_iter)**

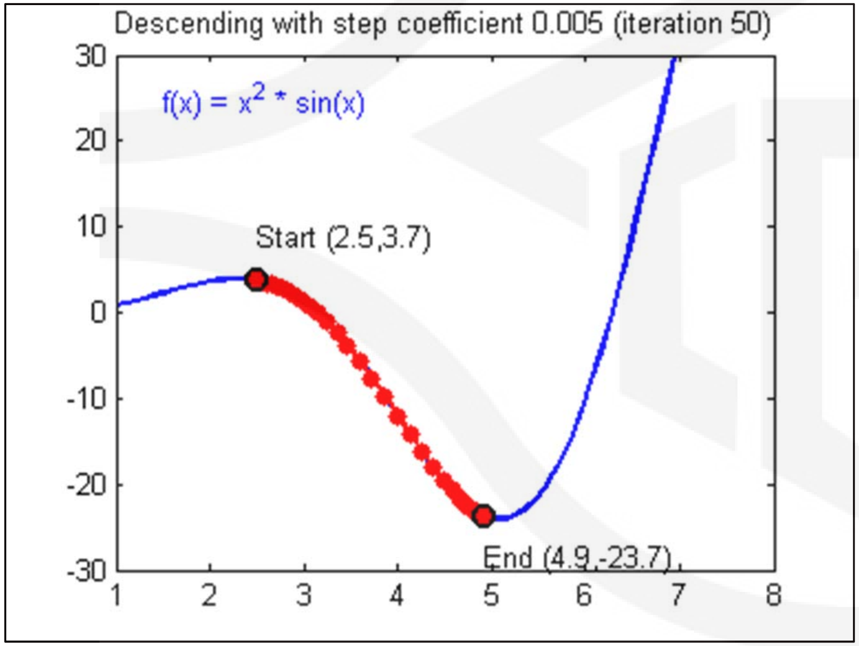


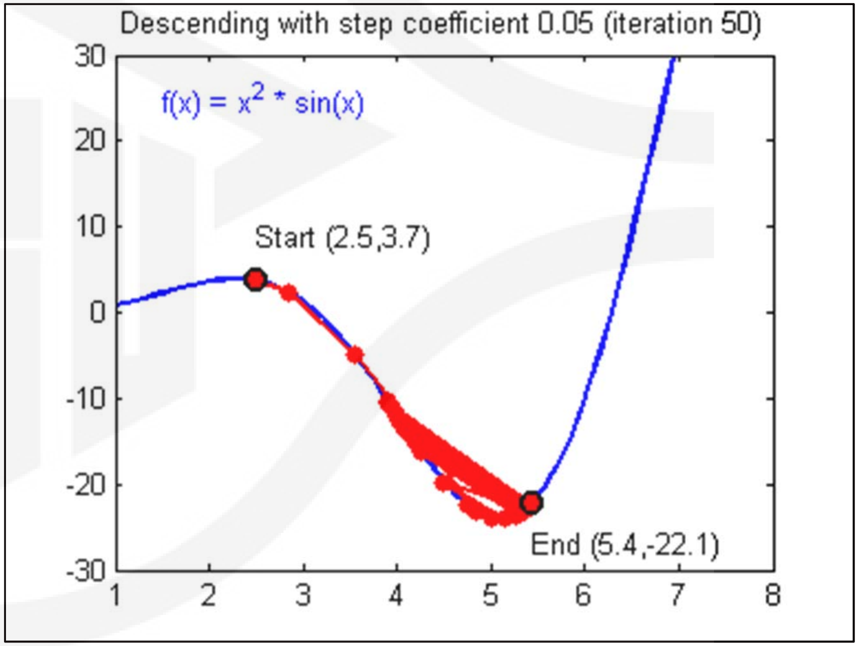
Gradient Descent to Achieve Minimization



Effect of Learning Rate

- Gradient descent with different learning rate



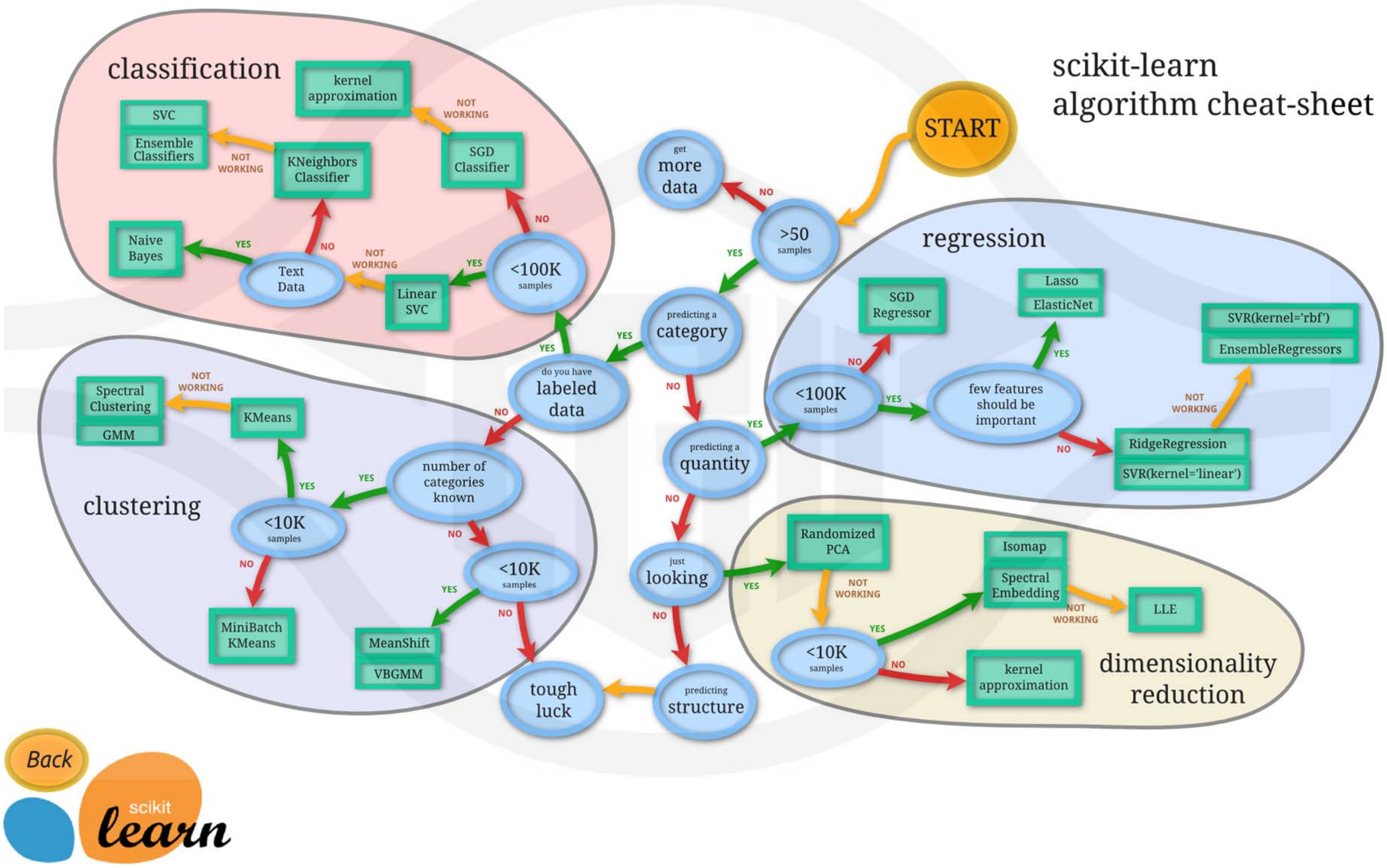


Coursera, Neural Networks and Deep Learning, Andrew Ng

What is Scikit-Learn

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

Learning Map of Scikit-Learn



Import Scikit-Learn (sklearn)

- from ... import ... (special use method)

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score
```

Load Data via sklearn

- Load data by built-in scikit-learn datasets

```
iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target
print('Class labels:', np.unique(y))
```

The iris dataset is a classic and very easy multi-class classification dataset.

Classes	3
Samples per class	50
Samples total	150
Dimensionality	4
Features	real, positive

Preprocessing via sklearn

- Split data (70% train set and 30% test set)

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1, stratify=y)
```

- Standardize features

```
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
```

Training and Testing

- Initialize the object and training

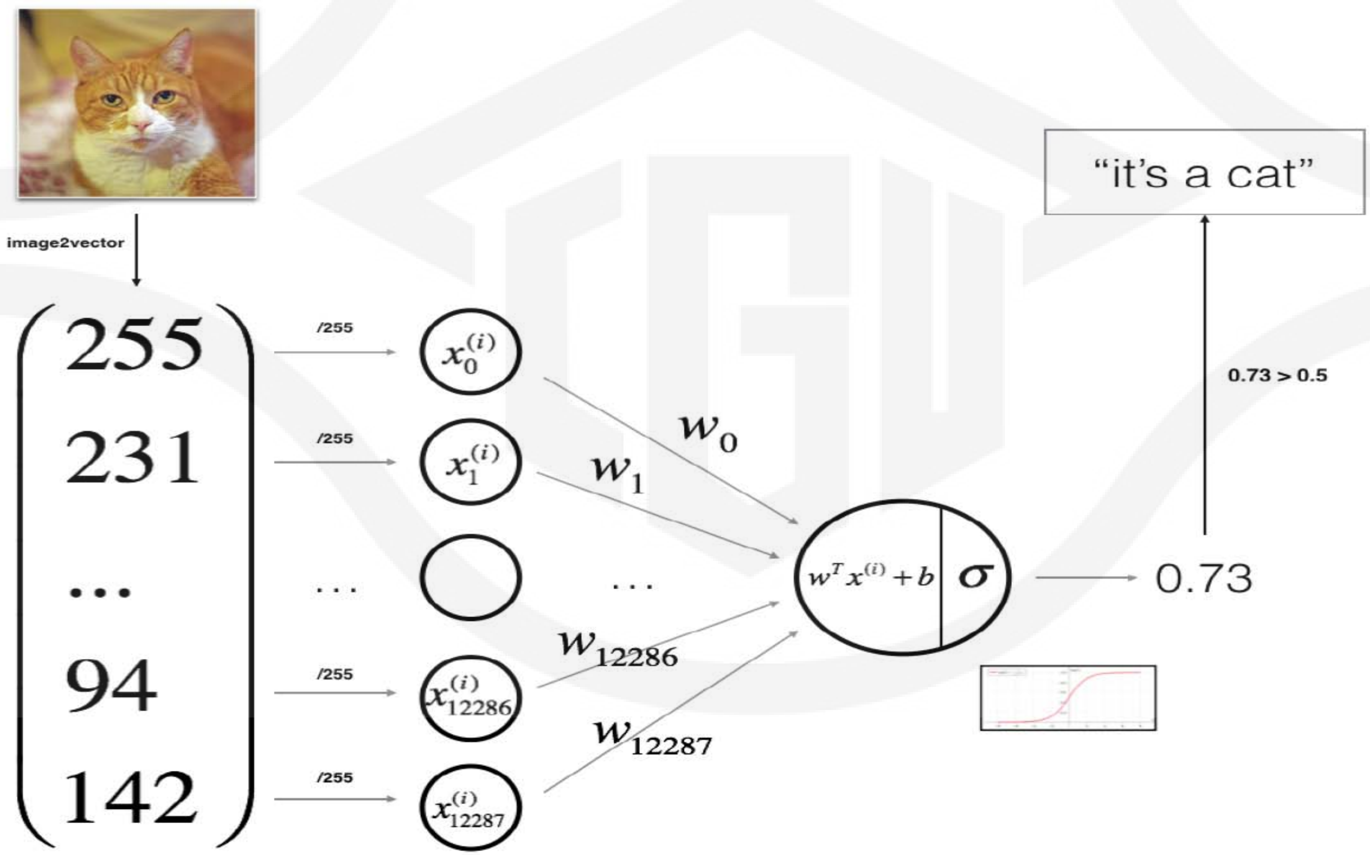
```
ppn = Perceptron(n_iter=40, eta0=0.1, random_state=1)
ppn.fit(X_train_std, y_train)
```

- Check the accuracy of test set

```
y_pred = ppn.predict(X_test_std)
print('Misclassified samples: %d' % (y_test !=
y_pred).sum())
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
```

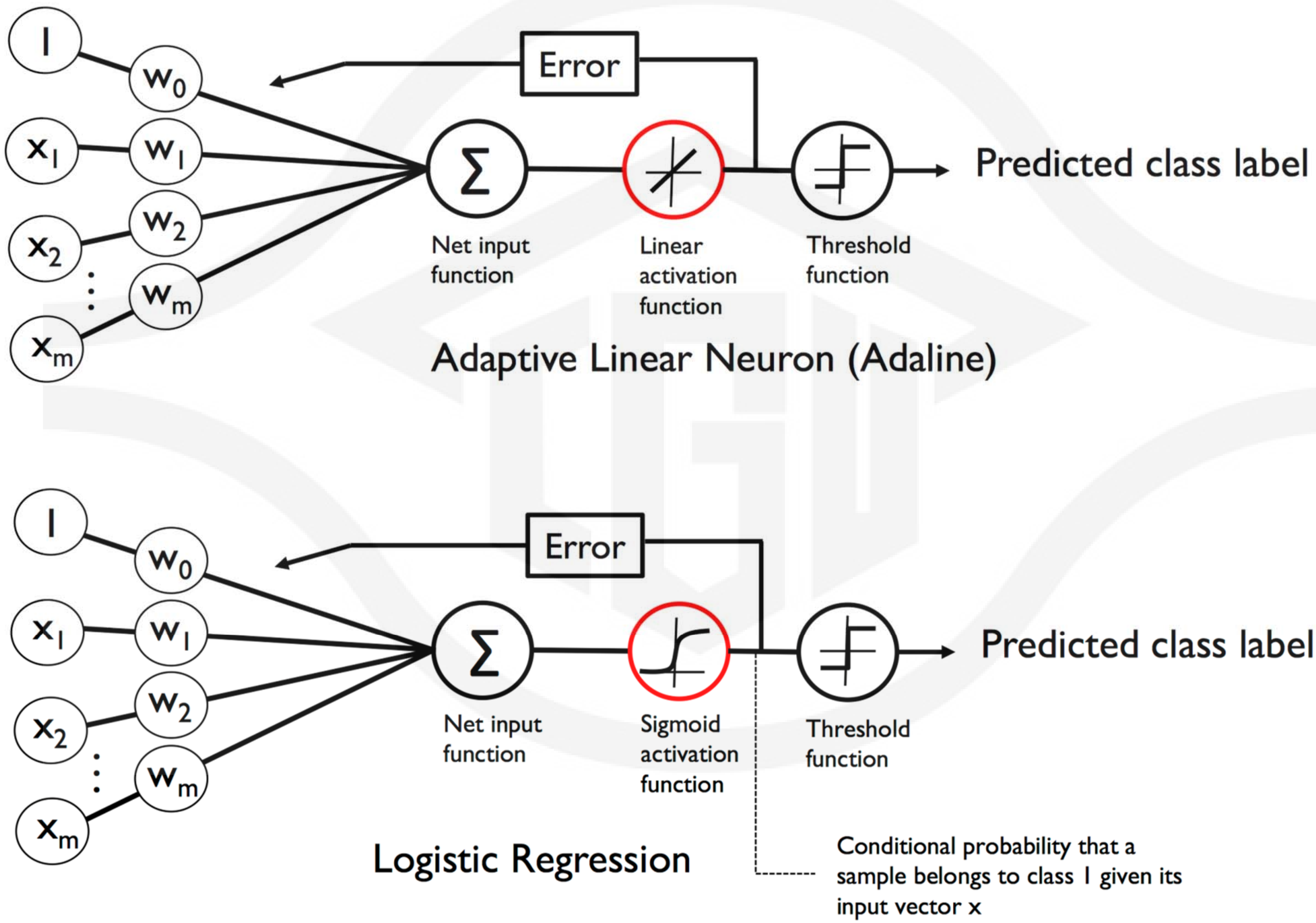
Logistic Regression

- Logistic Regression is actually a very simple Neural Network

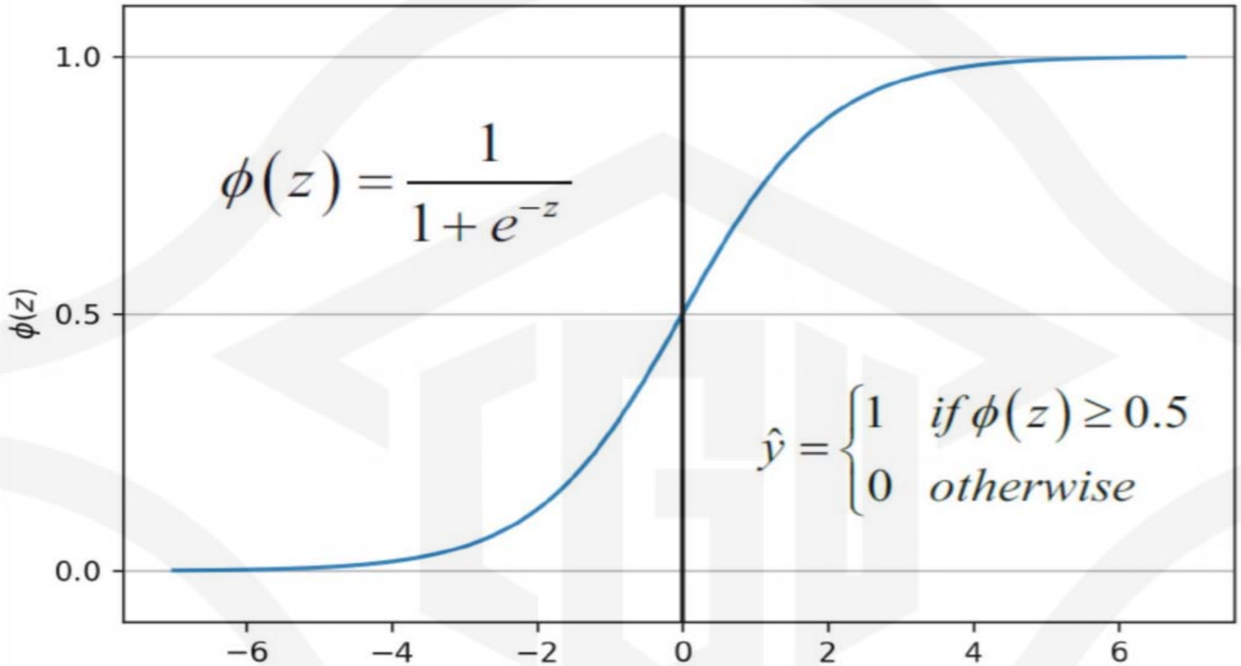


Coursera, Neural Networks and Deep Learning, Andrew Ng

Compare with Adaline



Activation Function with Probability



- Link $\phi(z)$ to Bernoulli's PDF $P(y \mid x, w)$

$$P(y|x,w) = p^y(1-p)^{1-y} \quad \begin{cases} p, if\ y = 1 \\ 1-p, if\ y = 0 \end{cases}$$

Logistic Regression Characteristic

- Replace with the sigmoid function

```
def activation(self, z):
    """Compute logistic sigmoid activation"""
    return 1. / (1. + np.exp(-z))
```

- Prediction range constrained between 0 to 1

```
def predict(self, X):
    """Return class label after unit step"""
    return np.where(self.activation(self.net_input(X))
>= 0.5, 1, 0)
```

Maximum Likelihood Estimation

- If training examples were identically independently distributed (IID), then maximum likelihood estimation (MLE) means to find the parameters of the model with given training data

$$L(\boldsymbol{w}) = P(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{w}) = \prod_{i=1}^{n} P\left(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{w}\right) = \prod_{i=1}^{n} \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

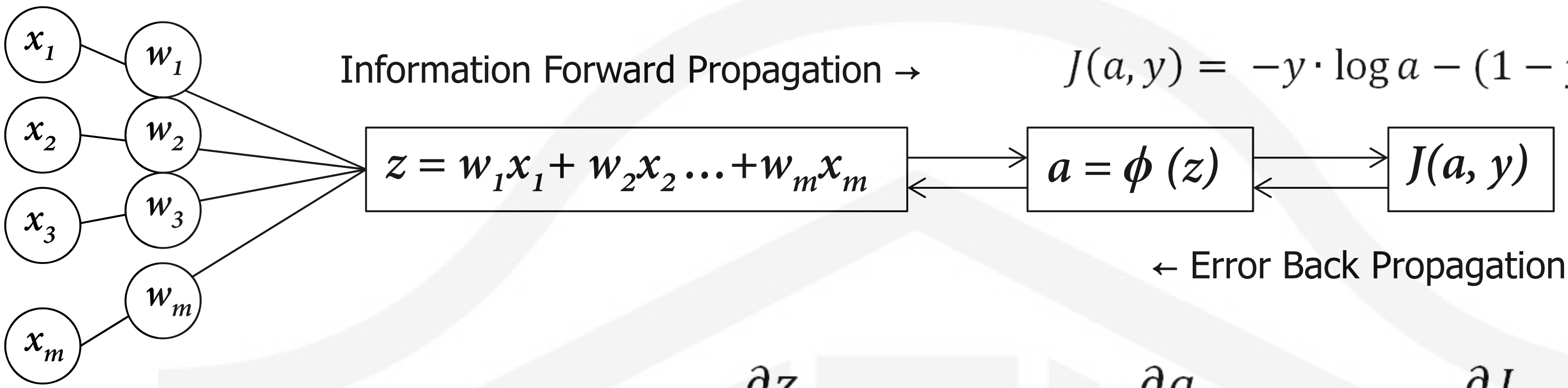
$$l(\boldsymbol{w}) = \log L(\boldsymbol{w}) = \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right]$$

- Negative log-likelihood as the cost function

$$J(\boldsymbol{w}) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right]$$

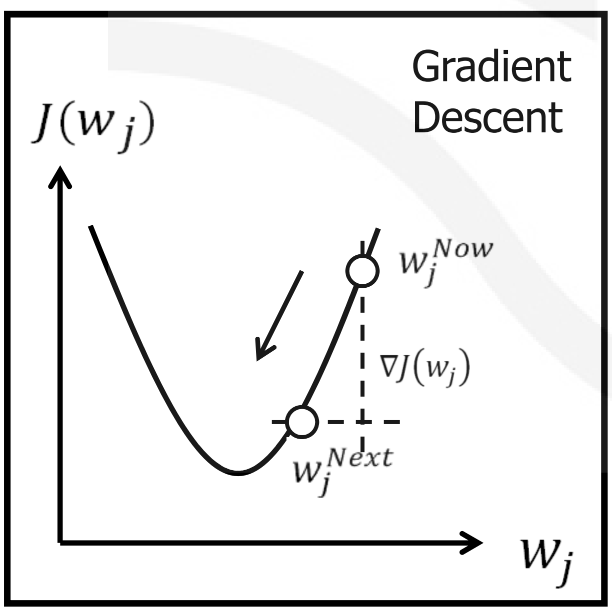
Learning Phase

$$J(a, y) = -y \cdot \log a - (1 - y) \cdot \log(1 - a)$$



$$\frac{\partial z}{\partial w_j} \times \frac{\partial a}{\partial z} \times \frac{\partial J}{\partial a} = \frac{\partial J(w_j)}{\partial w_j}$$

$$(x_j) \qquad (a \cdot (1 - a)) \qquad \left(-\frac{y}{a} + \frac{1-y}{1-a}\right)$$



$$\Rightarrow \quad \frac{\partial J(w_j)}{\partial w_j} = (\boldsymbol{a} - \boldsymbol{y})\boldsymbol{x}_j = \nabla J(w_j)$$

$$w_j^{Next} = w_j^{Now} - \eta \cdot \nabla J(w_j)$$

About Cost Function

- **Logistic regression** or **conditional log-likelihood cost function** $(-\log P(y|x))$ worked much better than the **quadratic cost**(squared errors) which was traditionally used to train feedforward neural networks for classification problems.

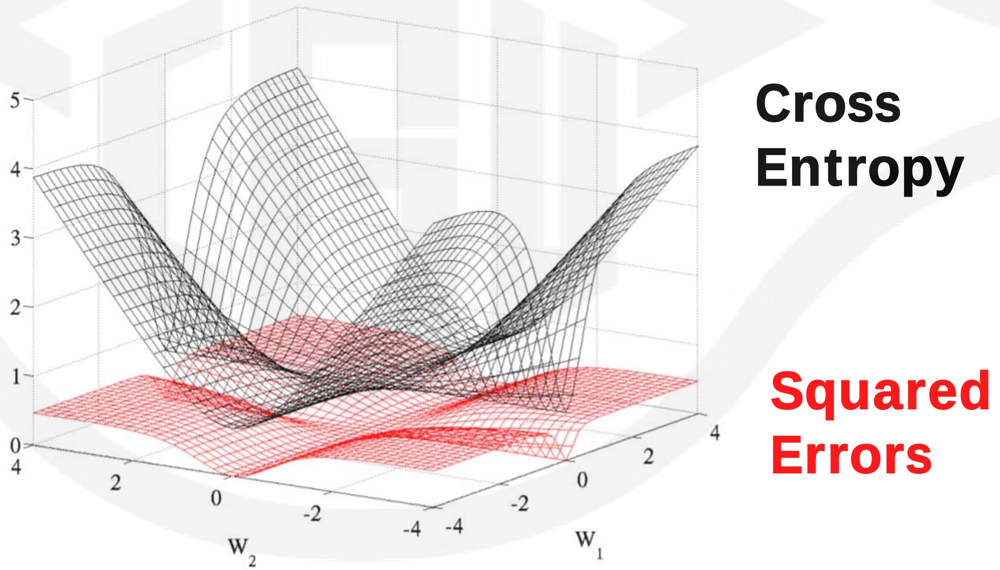


Figure 5: *Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two*

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In Proc. AISTATS, volume 9, pp. 249–256, 2010.

Logistic Regression Training Algorithm

- Implement

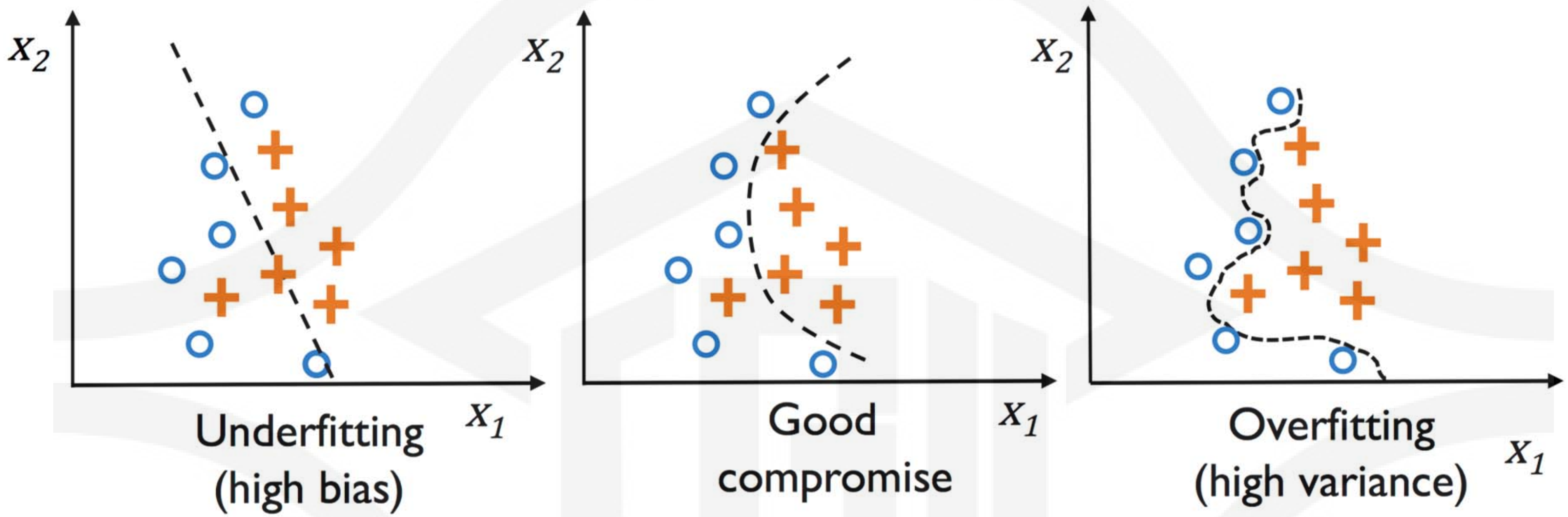
```
def fit(self, X, y):
  rgen = np.random.RandomState(self.random_state)
  self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
  self.cost_ = []
  for _ in range(self.n_iter):
      net_input = self.net_input(X)
      output = self.activation(net_input)
      errors = (y - output)
      self.w_[1:] += self.eta * X.T.dot(errors)
      self.w_[0] += self.eta * errors.sum()
      # note that we compute the logistic `cost` now
      # instead of the sum of squared errors cost
      cost = -y.dot(np.log(output)) - ((1 - y).dot(np.log(1 - output)))
      self.cost_.append(cost)
  return self
```

Logistic Regression via sklearn

- Initialize the object and training

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=100.0, random_state=1)
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()
```

Tackling Overfitting via Regularization



- L2 regularization

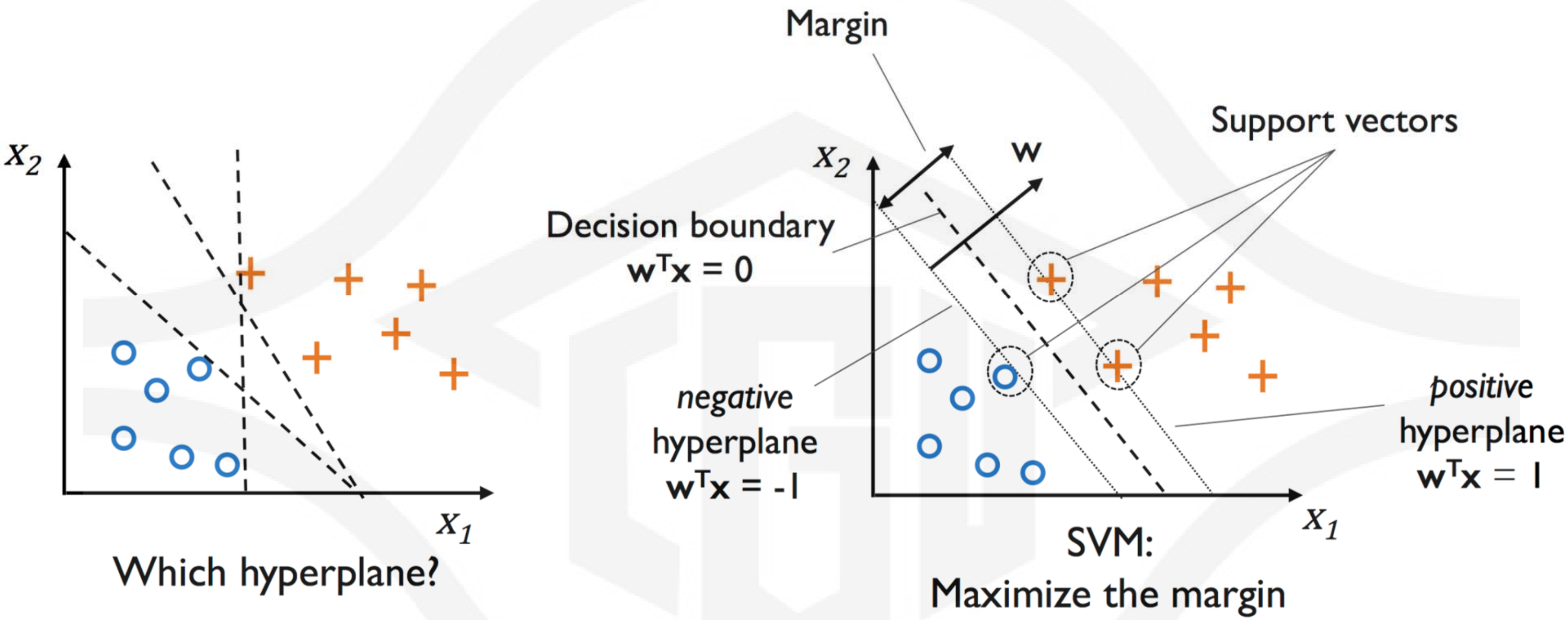
$$J(w) = \sum_{i=1}^{n}\left[-y^{(i)}\log\left(\phi\left(z^{(i)}\right)\right)-\left(1-y^{(i)}\right)\log\left(1-\phi\left(z^{(i)}\right)\right)\right]+\frac{\lambda}{2}\|w\|^2$$

Regularization Parameter

- C is regularization parameter λ 's inverse

```
weights, params = [], []
for c in np.arange(-5, 5):
    lr = LogisticRegression(C=10.**c, random_state=1)
    lr.fit(X_train_std, y_train)
    weights.append(lr.coef_[2])
    params.append(10.**c)
weights = np.array(weights)
plt.plot(params, weights[:, 0],
         label='petal length')
plt.plot(params, weights[:, 1], linestyle='--',
         label='petal width')
plt.ylabel('weight coefficient')
plt.xlabel('C')
plt.legend(loc='upper left')
plt.xscale('log')
plt.show()
```

Support Vector Machine



$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{pos} = 1 \qquad (1)$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{neg} = -1 \qquad (2)$$

$$\Rightarrow \boldsymbol{w}^T \left(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg}\right) = 2$$

$$\|\boldsymbol{w}\| = \sqrt{\sum_{j=1}^{m} w_j^2}$$

$$\frac{\boldsymbol{w}^T \left(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg}\right)}{\|\boldsymbol{w}\|} = \frac{2}{\|\boldsymbol{w}\|}$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 \ \ if \ \ y^{(i)} = 1$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \leq -1 \ \ if \ \ y^{(i)} = -1$$

$$\text{for } i = 1 \ldots N$$

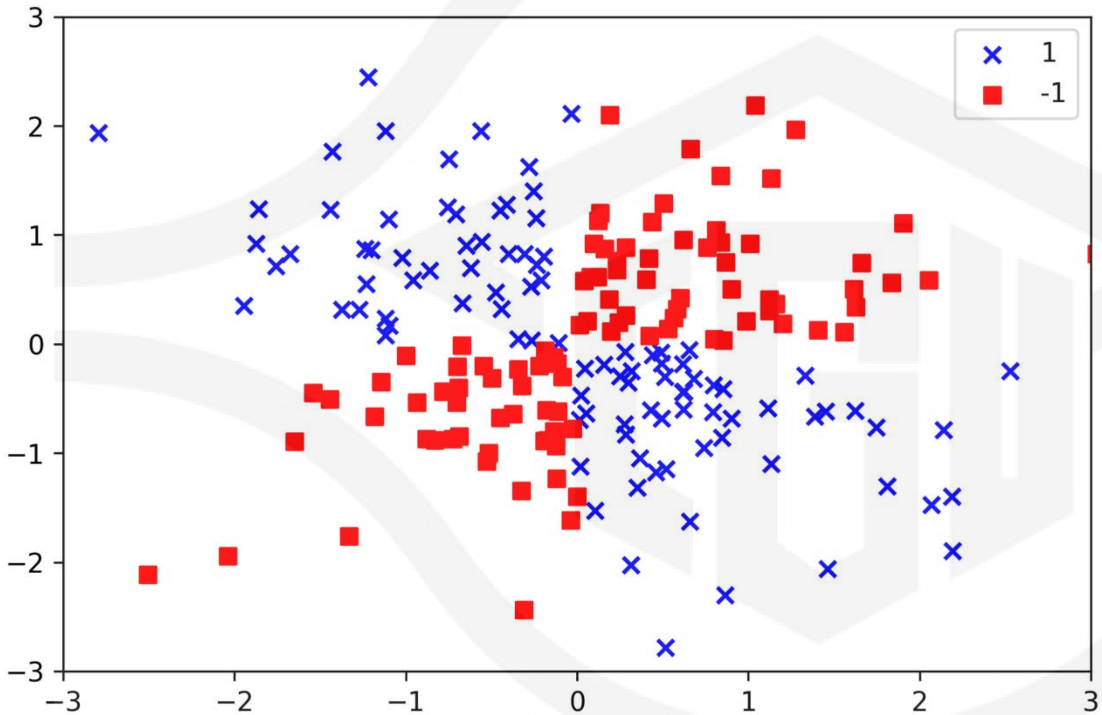
Linear Support Vector Machine via sklearn

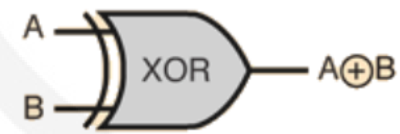
- Initialize the object and training

```
from sklearn.svm import SVC
svm = SVC(kernel='linear', C=1.0, random_state=1)
svm = SVC(kernel='rbf', random_state=1, gamma=0.10,
C=10.0)
svm.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std,
                      y_combined,
                      classifier=svm,
                      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()
```

Non-Linear Problems

- Exclusive-OR problems

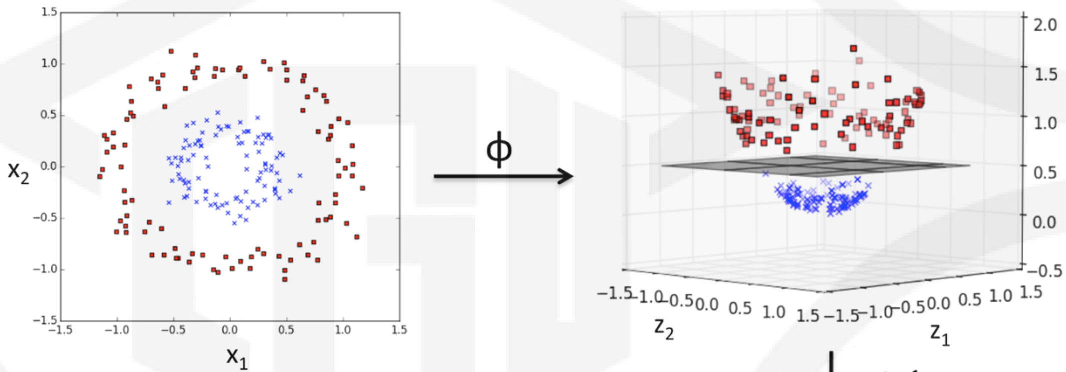




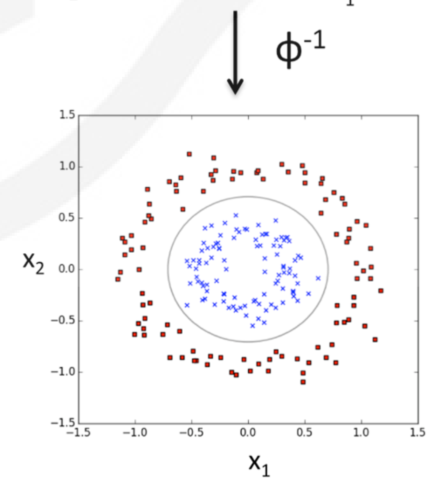
A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

Kernel Methods

- Create nonlinear combinations of original features to project them onto a higher-dimensional space where it becomes linearly separable.



$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$



Separating Hyperplanes in High-dimensional Space

- Increase features' dimension is computationally expensive especially if dealing with high-dimensional data
- Kernel trick
  - Replace the dot product x(i)T x(j) by K(x(i),x(j))
- Radial Basis Function (RBF) kernel or Gaussian kernel
  - Similarity function between a pair of samples.
  - A range between 1 (for exactly similar samples) and 0 (for very dissimilar samples)

$$\mathcal{K}\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\gamma \left\| \boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)} \right\|^2\right)$$

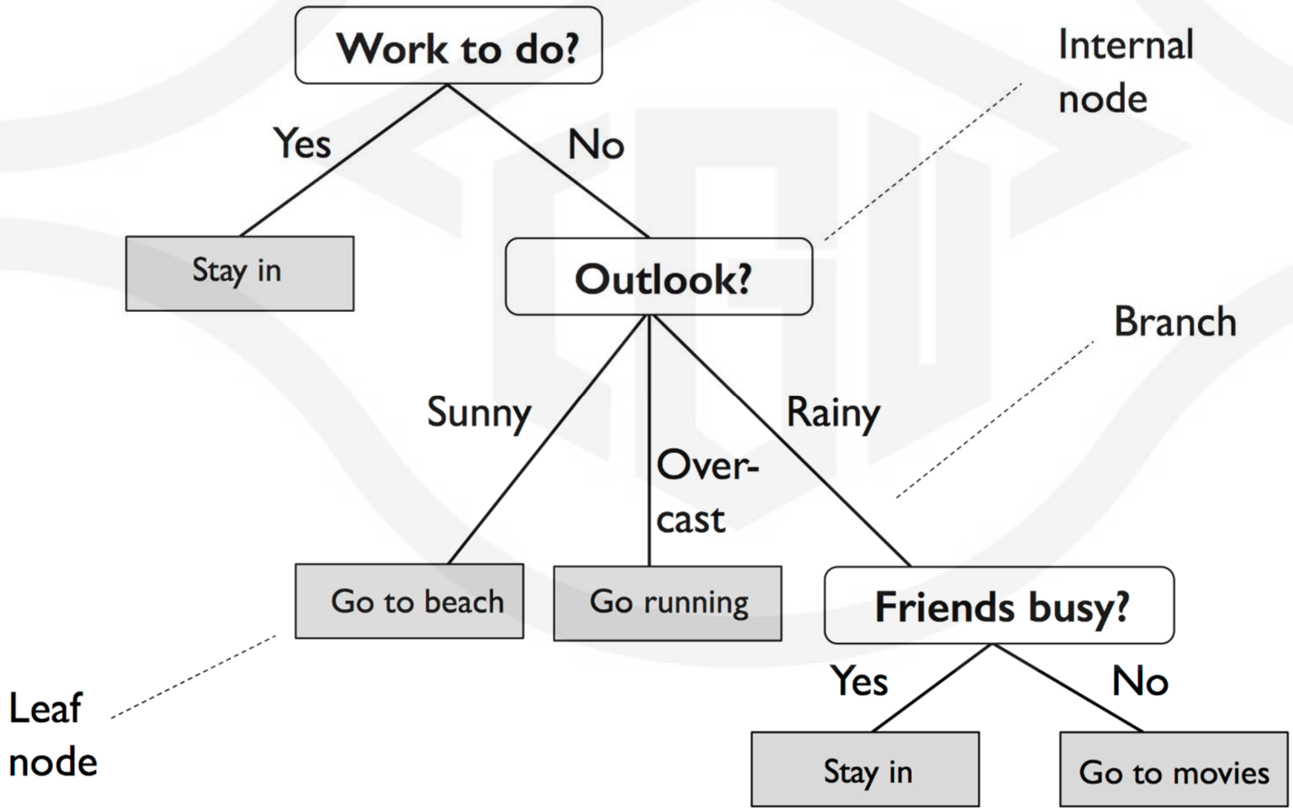
Using the Kernel Trick

- RBF kernel

```
np.random.seed(1)
X_xor = np.random.randn(200, 2)
y_xor = np.logical_xor(X_xor[:, 0] > 0,
                       X_xor[:, 1] > 0)
y_xor = np.where(y_xor, 1, -1)
svm = SVC(kernel='rbf', random_state=1, gamma=0.10,
C=10.0)
svm.fit(X_xor, y_xor)
plot_decision_regions(X_xor, y_xor,
                      classifier=svm)
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

Decision Tree

- **Interpretability:** make a decision based on asking a series of questions



Information Gain

- Using the decision algorithm, we start at the tree root and split the data on the feature that results in the largest **Information Gain (IG)**
- **Maximizing information gain**

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

Impurity Measure by Entropy (IH)

$$I_H(t) = -\sum_{i=1}^{c} p(i|t)\log_2 p(i|t)$$

- p(i|t): proportion of samples that belong to class i for a node t
  - The entropy is 0 if all samples belong to the same class
  - The entropy is maximal if a uniform class distribution
  - For example, entropy=0 if $p(i=1|t)=1$, $p(i=0|t)=0$; entropy=1 if $p(i=1|t)=0.5$, $p(i=0|t)=0.5$.
- **The entropy criterion attempts to maximize the mutual information in the tree**

Gini Impurity (IG)

$$I_G(t) = \sum_{i=1}^{c} p(i \mid t)\left(1 - p(i \mid t)\right) = 1 - \sum_{i=1}^{c} p(i \mid t)^2$$

- Gini impurity is maximal if the classes are perfectly mixed, for example, in a binary class setting (c = 2):

$$I_G(t) = 1 - \sum_{i=1}^{c} 0.5^2 = 0.5$$

Impurity Measure by Classification Error (IE)

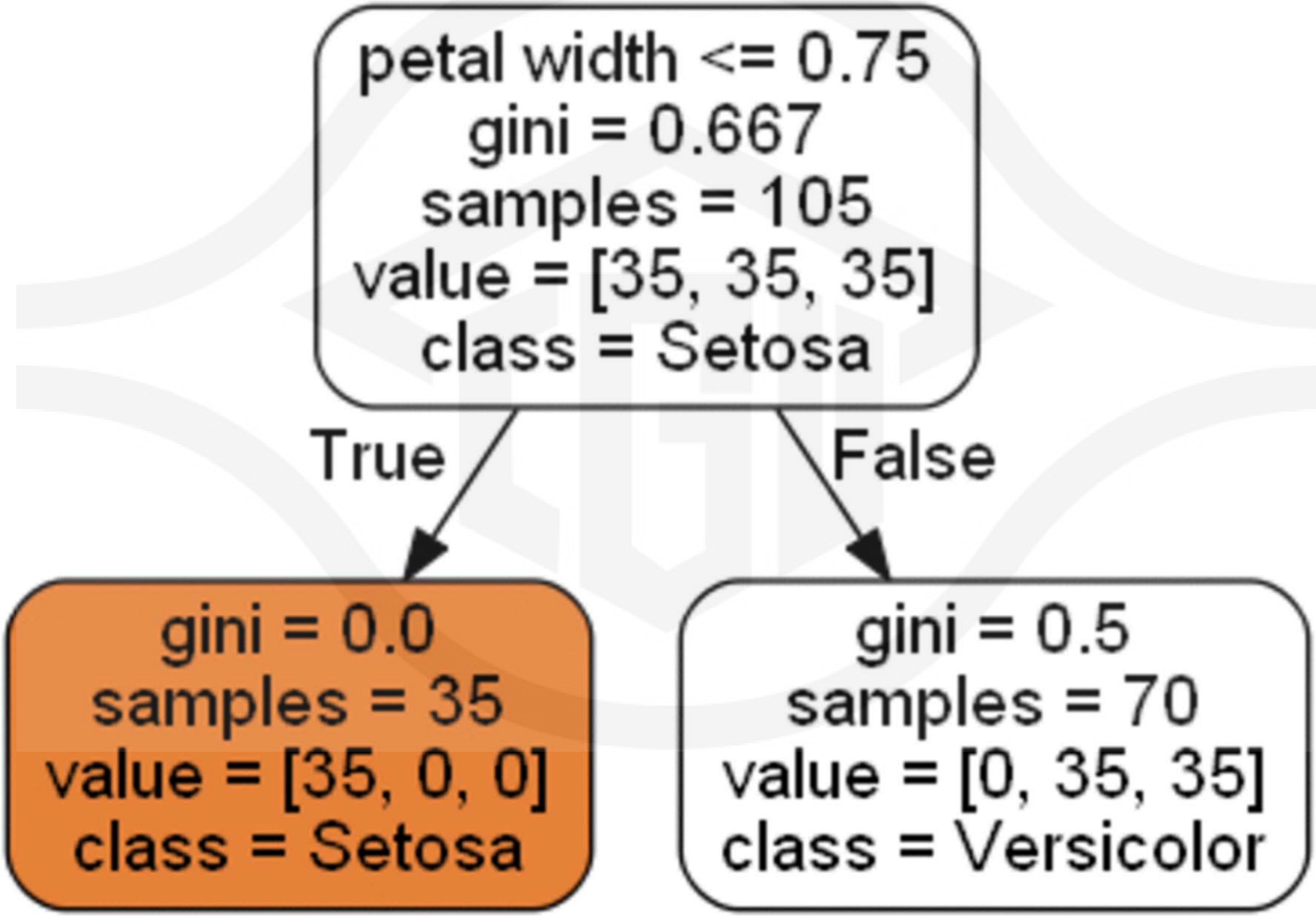
$$I_E = 1 - \max\left\{ p\left(i \mid t \right) \right\}$$

- A useful criterion for pruning but not recommended for growing a decision tree, since it is less sensitive to changes in the class probabilities of the nodes.

Decision Tree via sklearn

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(criterion='gini',
                              max_depth=4,
                              random_state=1)
tree.fit(X_train, y_train)
X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))
plot_decision_regions(X_combined, y_combined,
                      classifier=tree,
test_idx=range(105, 150))
plt.xlabel('petal length [cm]')
plt.ylabel('petal width [cm]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

Max Depth


petal width <= 0.75
gini = 0.667
samples = 105
value = [35, 35, 35]
class = Setosa
True
False
gini = 0.0
samples = 35
value = [35, 0, 0]
class = Setosa
gini = 0.5
samples = 70
value = [0, 35, 35]
class = Versicolor

Multiple Decision Trees to Random Forests

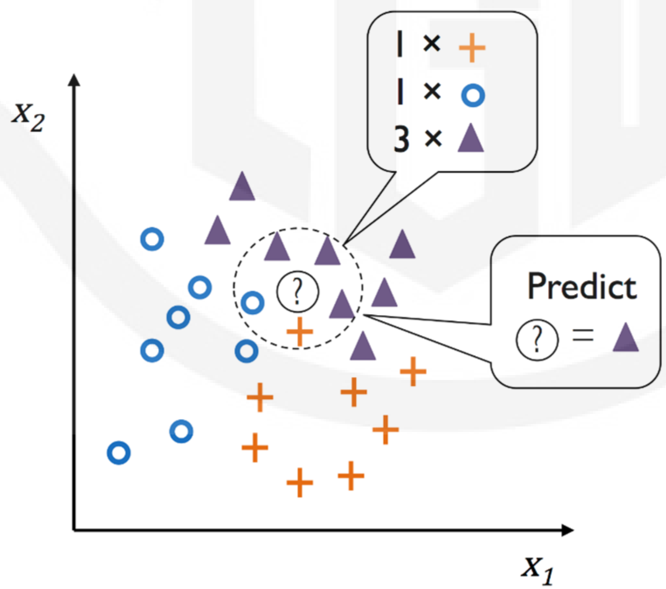
- Random forest algorithm
  - Draw a random **bootstrap** sample (randomly choose n samples from the training set).
  - Grow a decision tree from the bootstrap sample. At each node:
    - Randomly select d features.
    - Split the node using the feature that provides the best split (maximal information gain).
  - Repeat the steps 1-2 k times.
  - Aggregate the prediction by each tree to assign the class label by **majority vote**.

Random Forests via sklearn

```
from sklearn.ensemble import RandomForestClassifier
forest = RandomForestClassifier(criterion='gini',
                                n_estimators=25,
                                random_state=1,
                                n_jobs=2)
forest.fit(X_train, y_train)
plot_decision_regions(X_combined, y_combined,
                      classifier=forest,
test_idx=range(105, 150))
plt.xlabel('petal length [cm]')
plt.ylabel('petal width [cm]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

K-Nearest Neighbors

- It doesn't learn a discriminative function from the training data, but memorizes the training dataset instead
  - Choose the number of *k* and a distance metric.
  - Find the *k*-nearest neighbors of the sample to classify.
  - Assign the class label by majority vote.



Minkowski distance

- Minkowski distance is typically used with *p* being 1 or 2, which correspond to the Manhattan distance and the Euclidean distance, respectively.

$$D(X,Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

K-Nearest Neighbors via sklearn

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5,
                           p=2,
                           metric='minkowski')
knn.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std, y_combined,
                      classifier=knn,
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()
```

Reference

- Sebastian Raschka, Vahid Mirjalili. Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow. Second Edition. Packt Publishing, 2017.