SVMs, Decision Trees, Random Forest, and *k*-NNs

Machine Learning for Engineering Applications

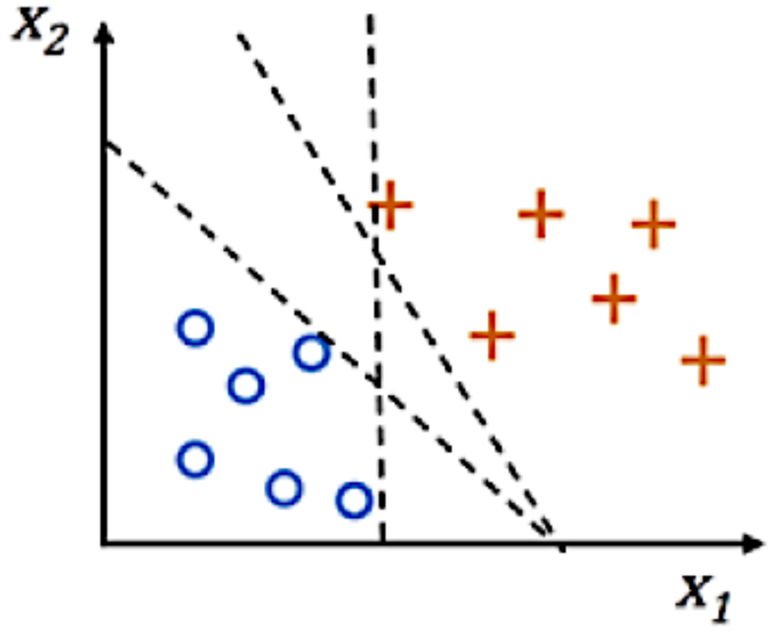
Fall 2023

Support Vector Machines (SVM)

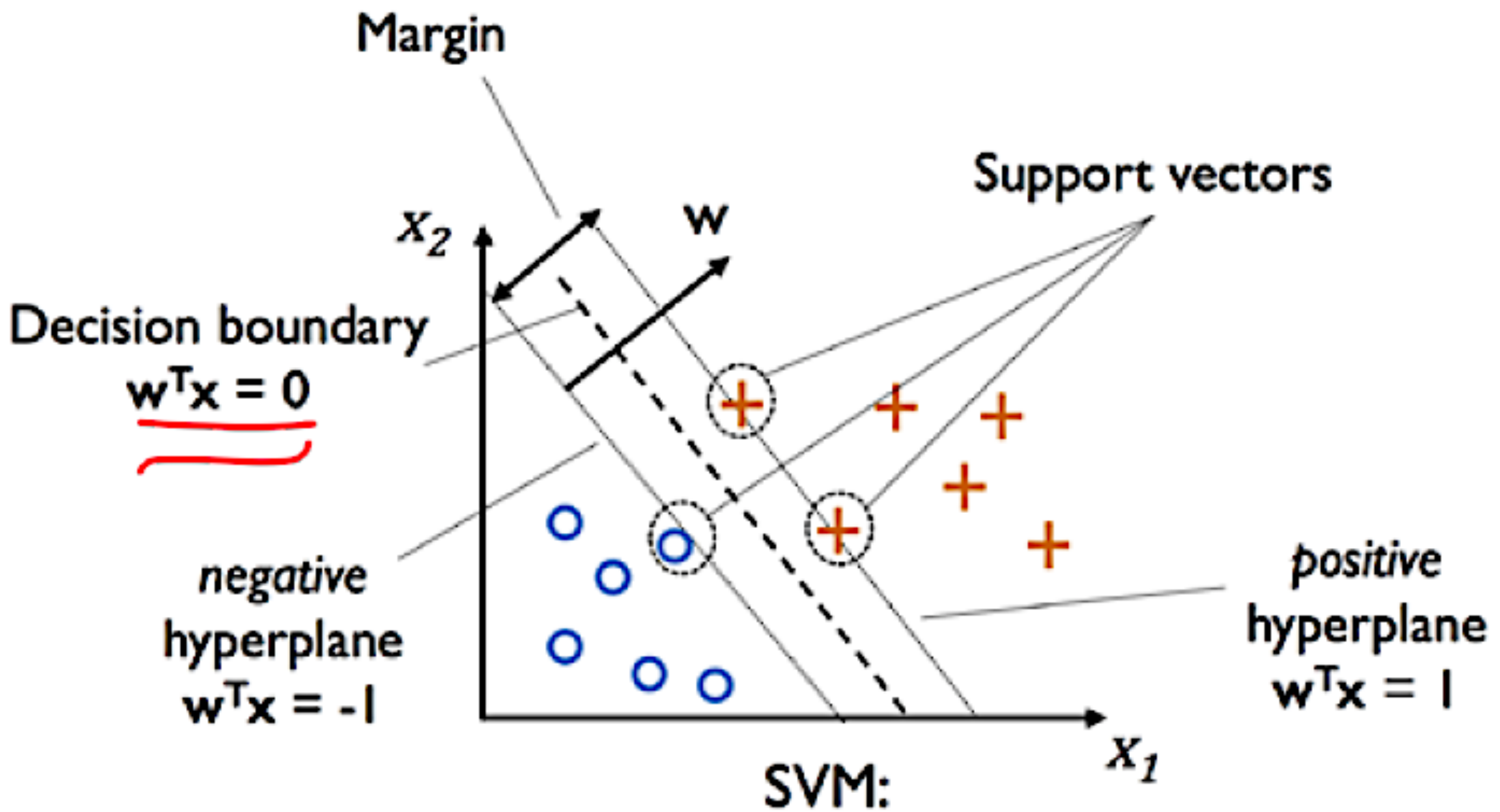
- **SVM** is one of the most power and most used classifier in the world!

- The Python community has created many SVM libraries that support all types of classification problems

- Non-linear classification is the essential part of this tool



Which hyperplane?

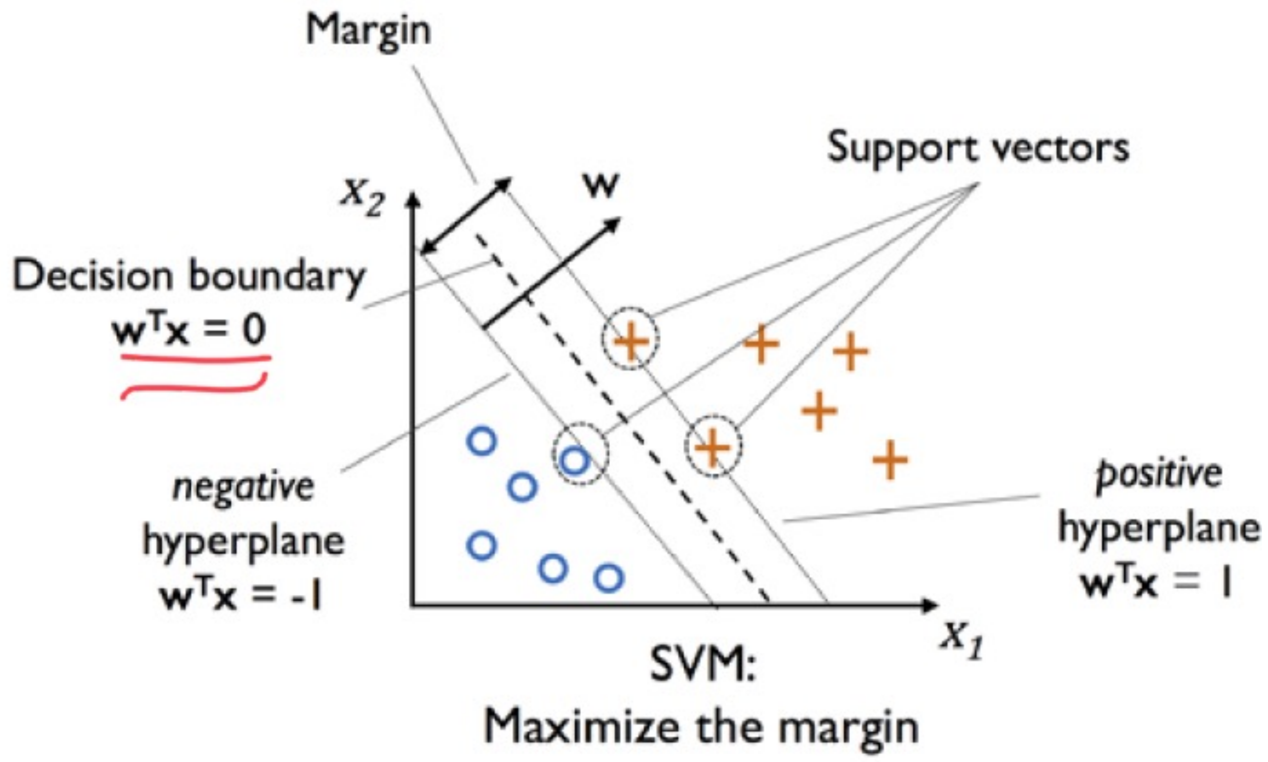


SVM:
Maximize the margin

- **SVM** determines many parts of a classification

- Identifying the vectors
- Drawing the hyperplanes
- Calculate direction (magnitude)
- Margins
- The distance of the hyperplanes
- So on…

- **Reason**: The best optimized (*larger*) boundary will help the model not to overfit



- **SVM** determines many parts of a classification

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

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

margin distance

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

- Normalize the weights:

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

- **You end up with:**

$$\frac{w^T\left(x_{pos} - x_{neg}\right)}{\|w\|} = \underbrace{\boxed{\frac{2}{\|w\|}}}_{\text{Margin}} \longrightarrow \text{Maximize this part!}$$

- For evaluation, the margin is expressed in its reciprocal:

$$\frac{1}{2}\|w\|^2$$

- **We need a slack variable (ξ):**
- ξ: This drives the model to be a soft-margin classifier
- Hyperplane effect:

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

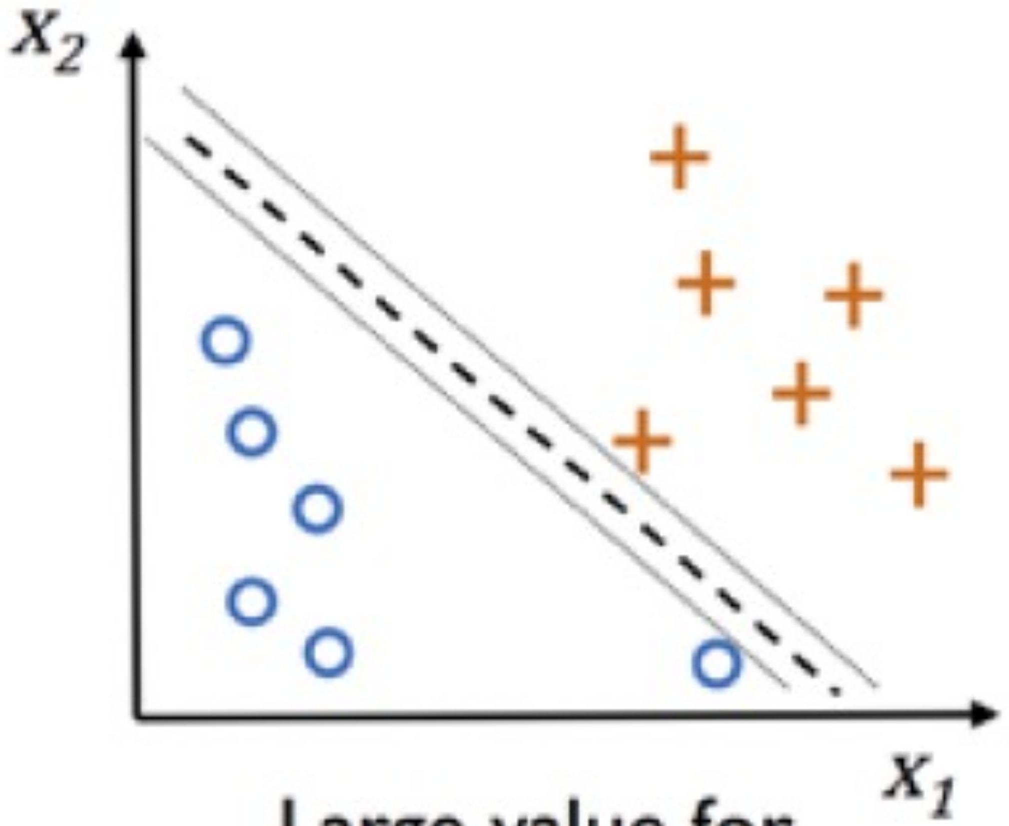
$$w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \leq -1 + \xi^{(i)} \quad if \;\; y^{(i)} = -1$$

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

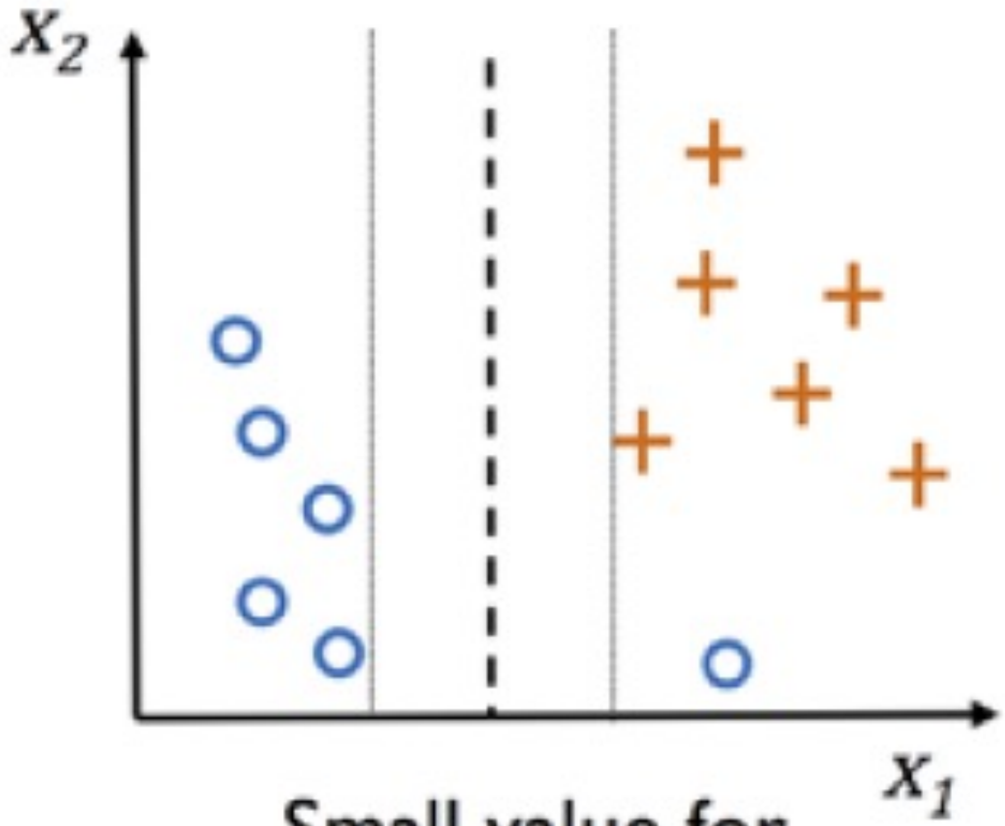
- **You now want to minimize the ξ:**

$$\frac{1}{2}\|w\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

- The C-variable controls the penalty intensity
- Large C-values: Large Error Penalties / Vice-Versa

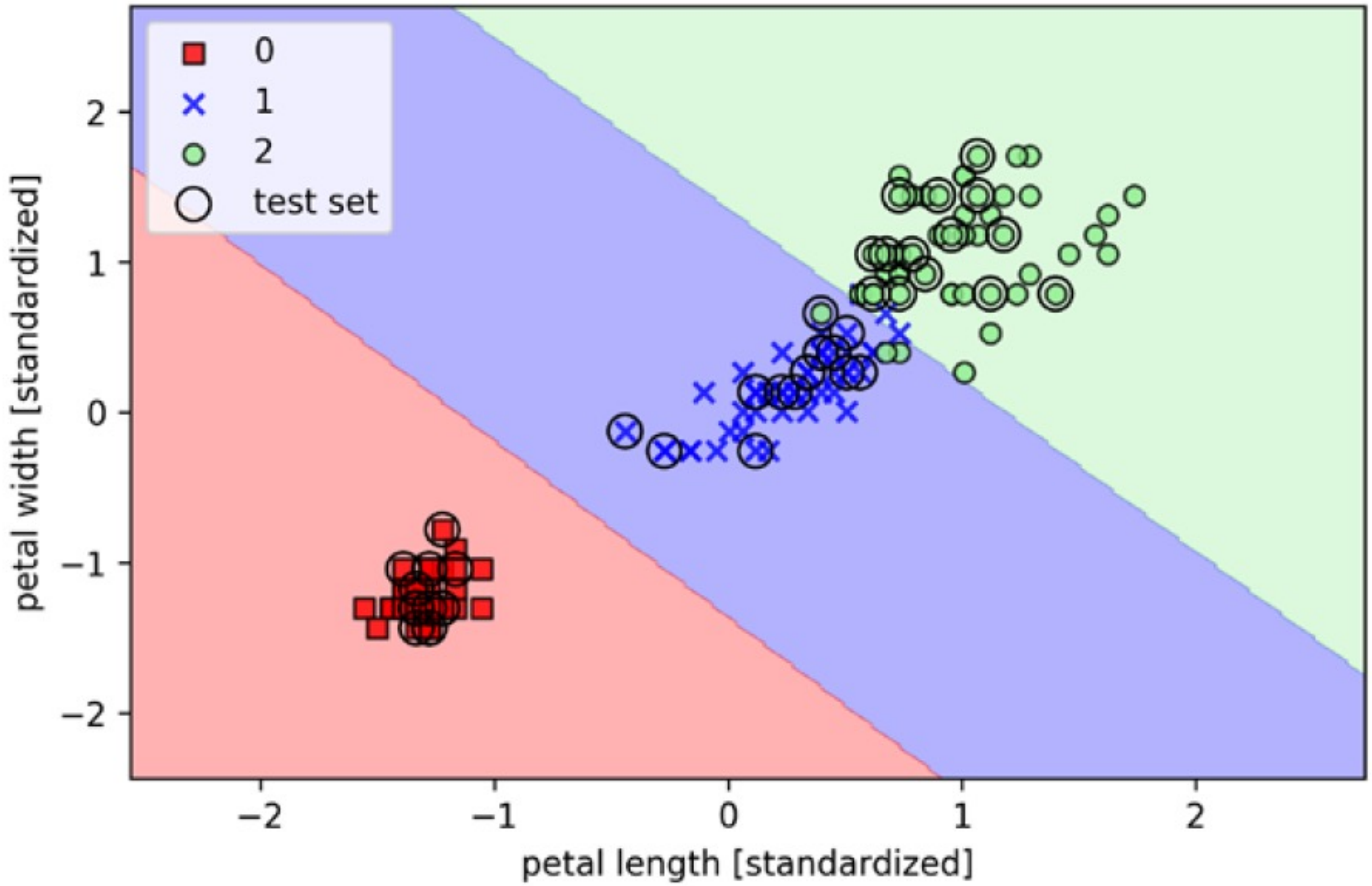


Large value for parameter C



Small value for parameter C

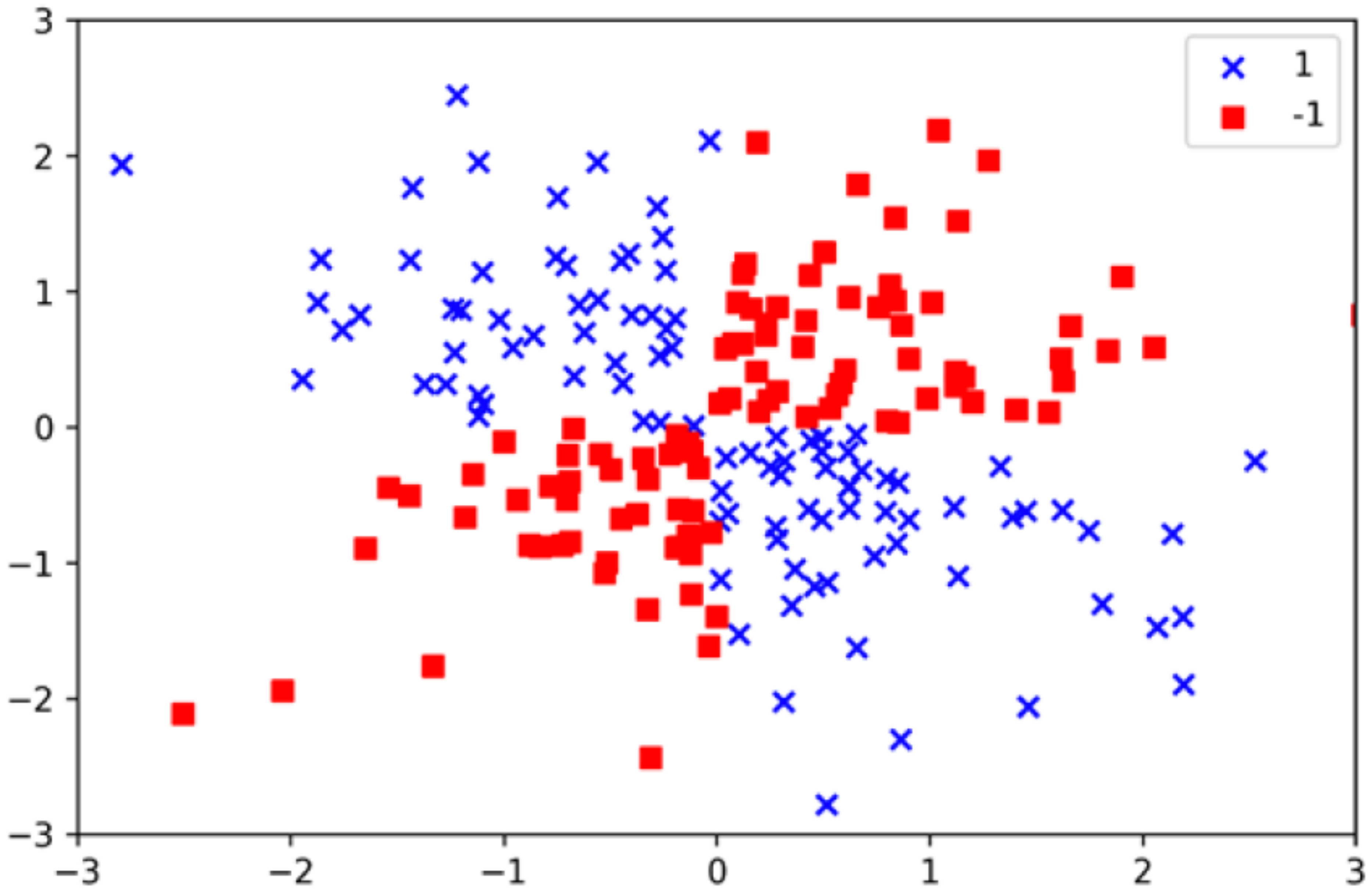
SVM LEAP
Demo


0
1
2
test set
petal width [standardized]
petal length [standardized]
2
1
0
−1
−2
−2
−1
0
1
2

SVM with Non-Linearity

- The XOR non-linear example with random noise

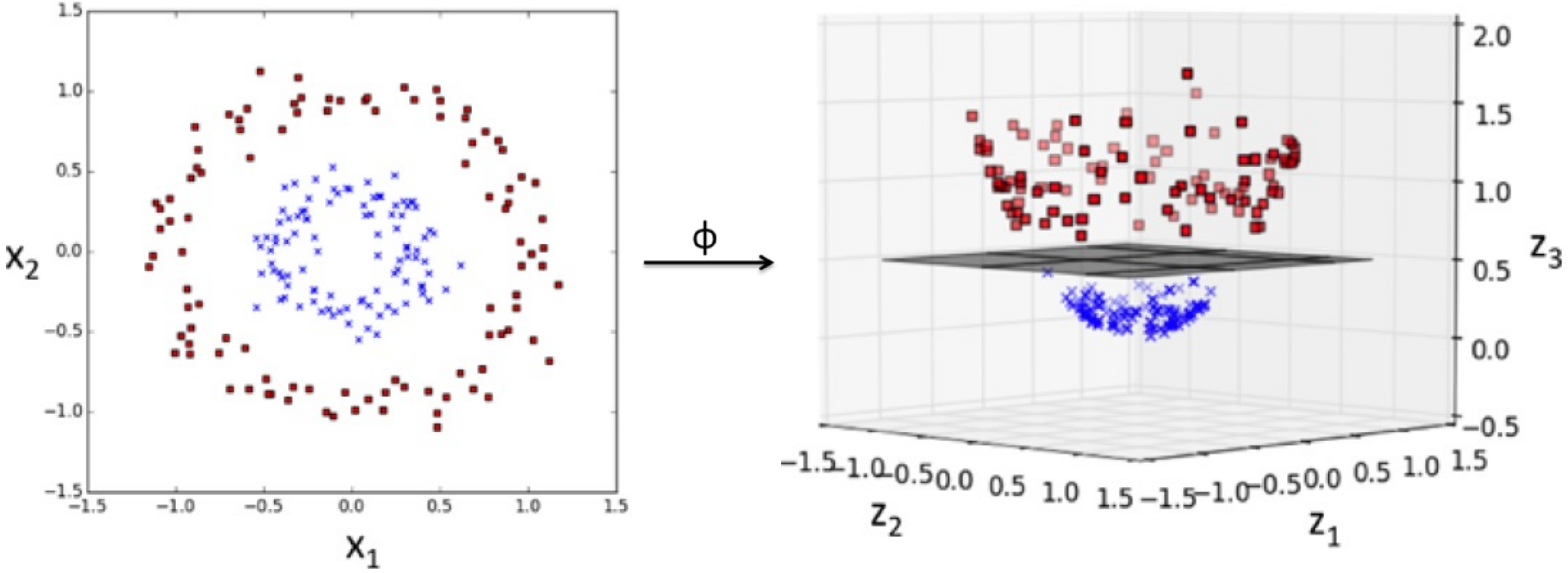
```
import matplotlib.pyplot as plt
import numpy as np

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)
```

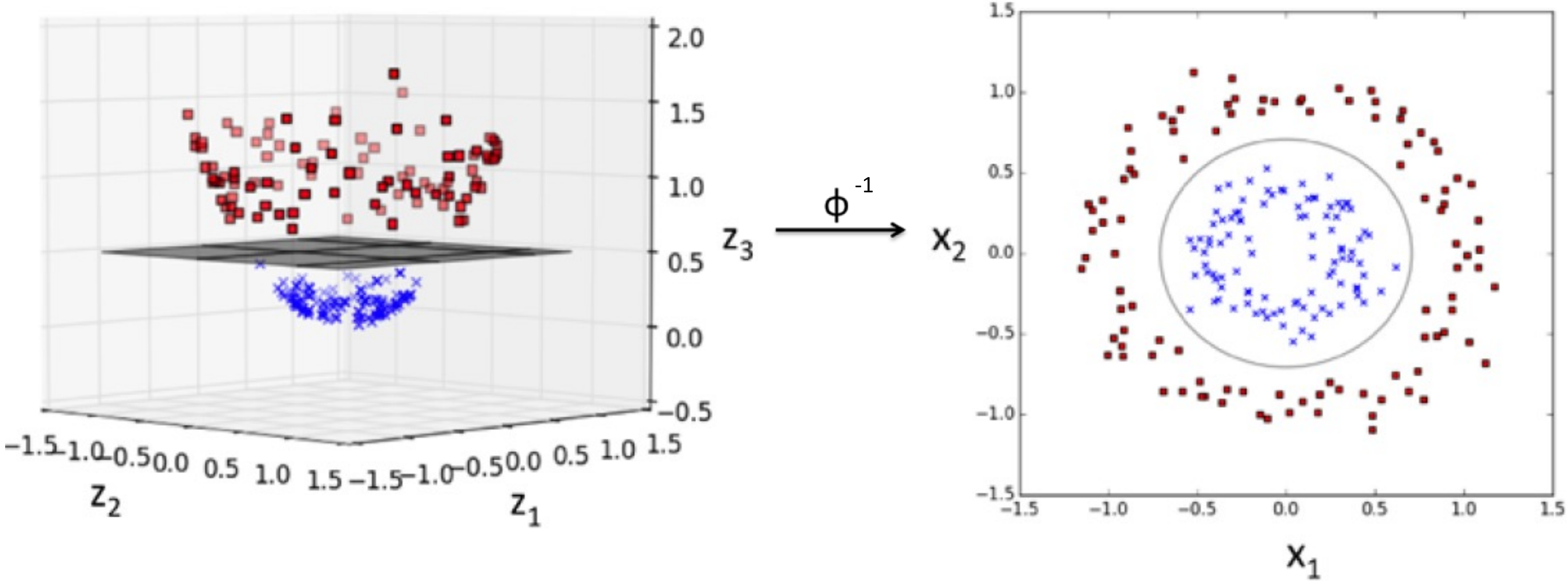


1
-1

- Kernel methods, or Kernel SVM, helps with this type of non-linear issue
- The kernel method is a transformation of the data: 2D to a 3D space:

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


x₂
x₁
−1.5 −1.0 −0.5 0.0 0.5 1.0 1.5
ϕ
z₃
2.0 1.5 1.0 0.5 0.0 −0.5
z₂
z₁


2.0
1.5
1.0
0.5
0.0
−0.5
z3
−1.5 −1.0 −0.5 0.0 0.5 1.0 1.5
z2
−1.5 −1.0 −0.5 0.0 0.5 1.0 1.5
z1
ϕ -1
x2
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5 −1.0 −0.5 0.0 0.5 1.0 1.5
x1

- The transformation:

$$\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}$$

$$\downarrow$$

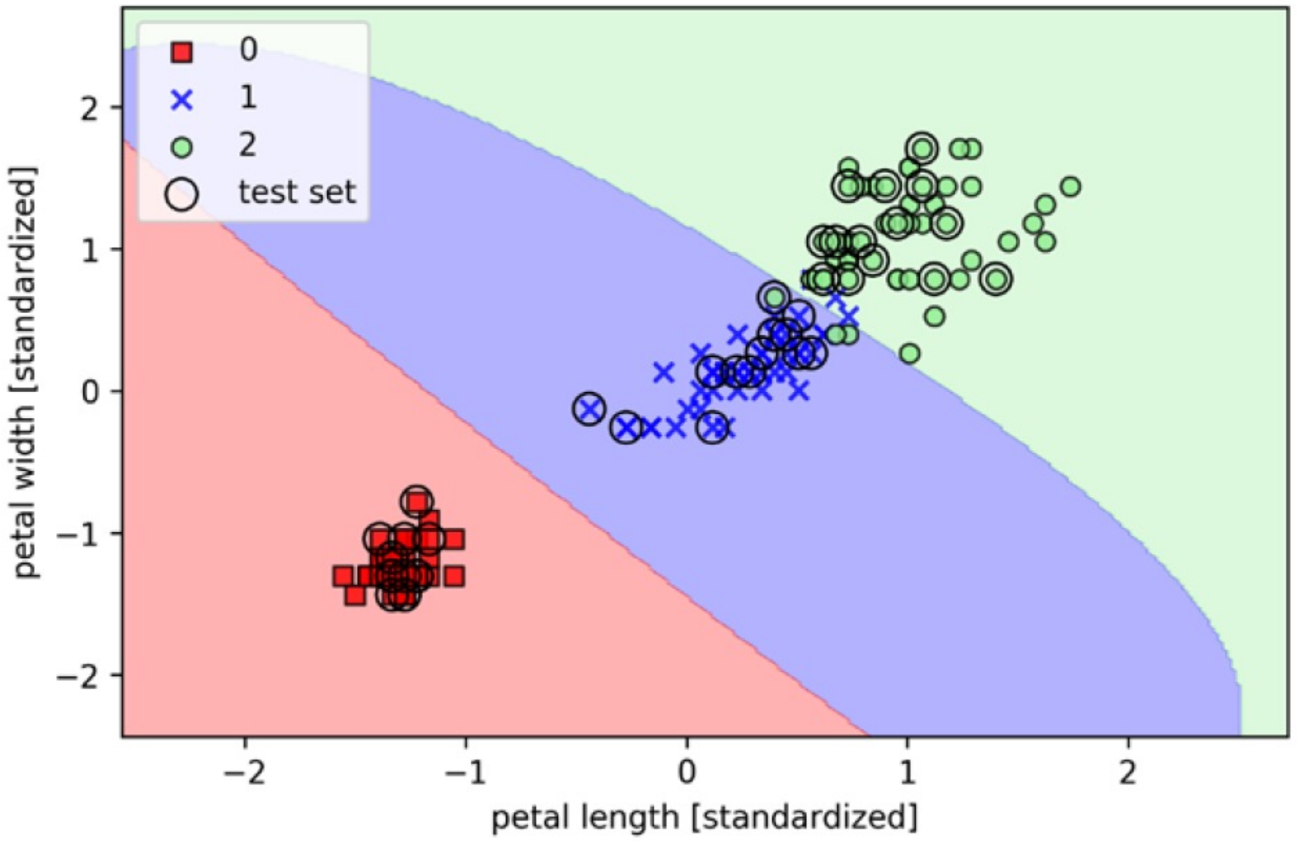
$$\mathcal{K}\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\phi\left(\boldsymbol{x}^{(i)}\right)^{T}\phi\left(\boldsymbol{x}^{(j)}\right)$$

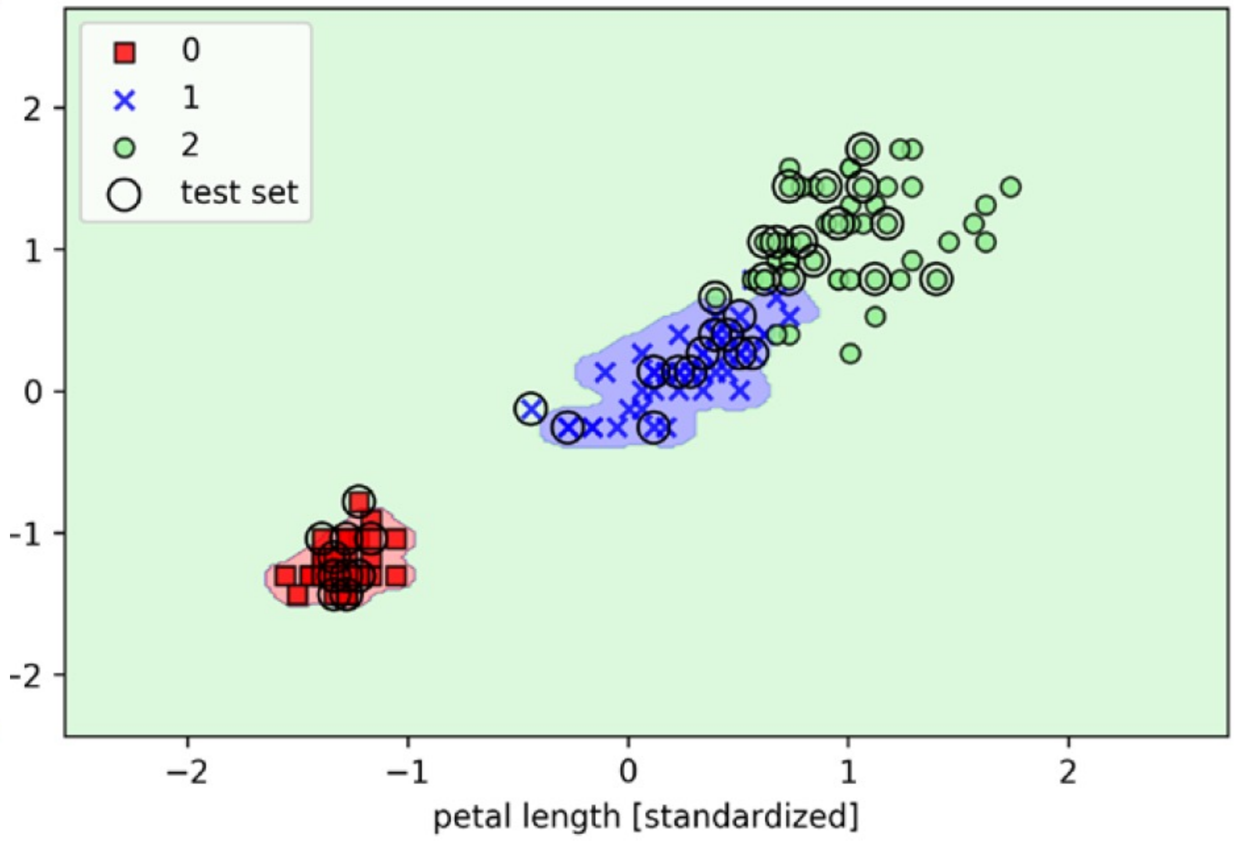
$$\mathcal{K}\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^{2}}{2\sigma^{2}}\right)\longrightarrow\boxed{\text{Radial Basis Function (RBF)}}$$

$$\mathcal{K}\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right)=\exp\left(-\gamma\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^{2}\right)\longrightarrow\gamma=\frac{1}{2\sigma^{2}}$$

Low Gamma Values

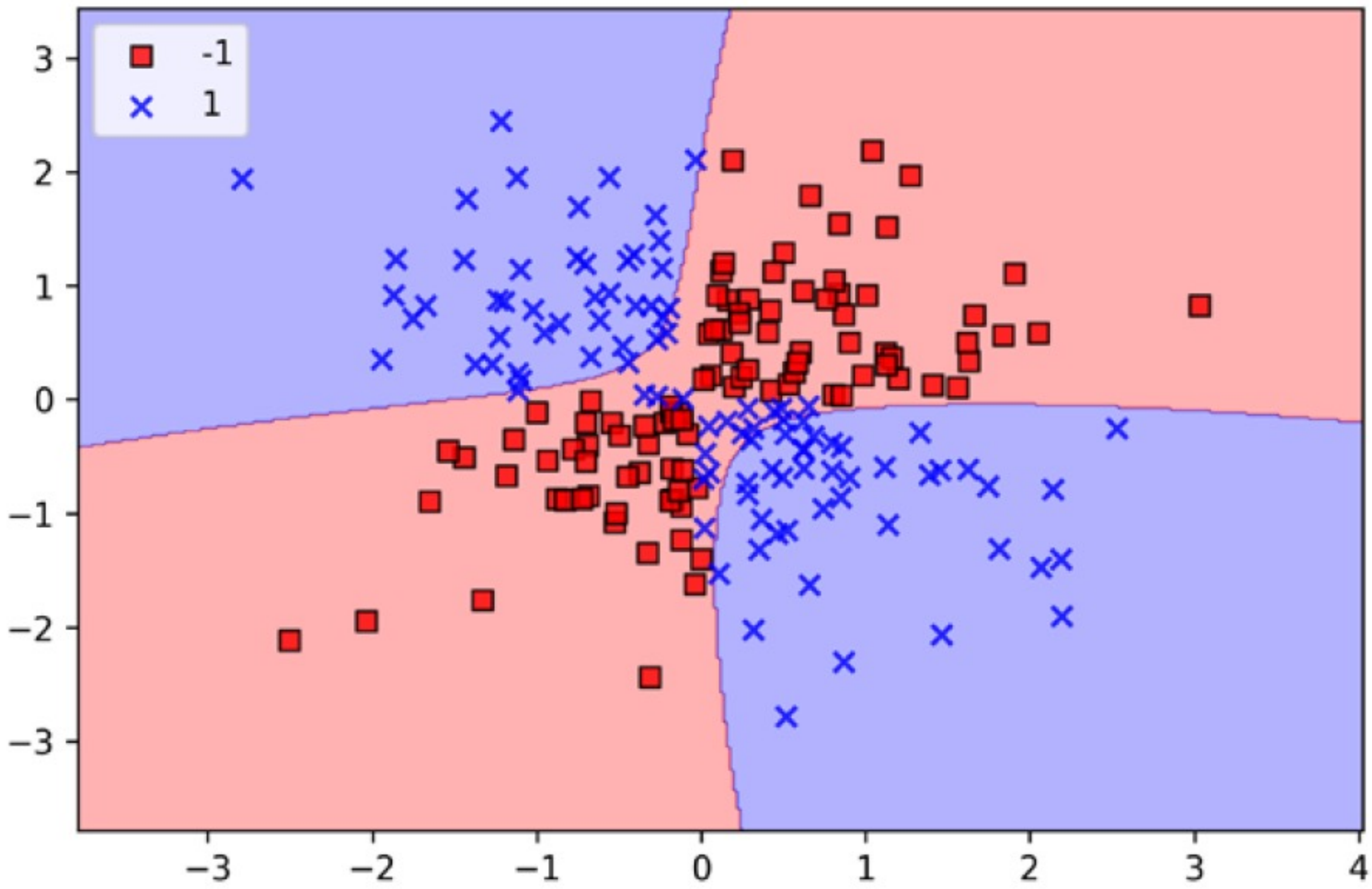
Large Gamma Values





- The XOR non-linear example with random noise

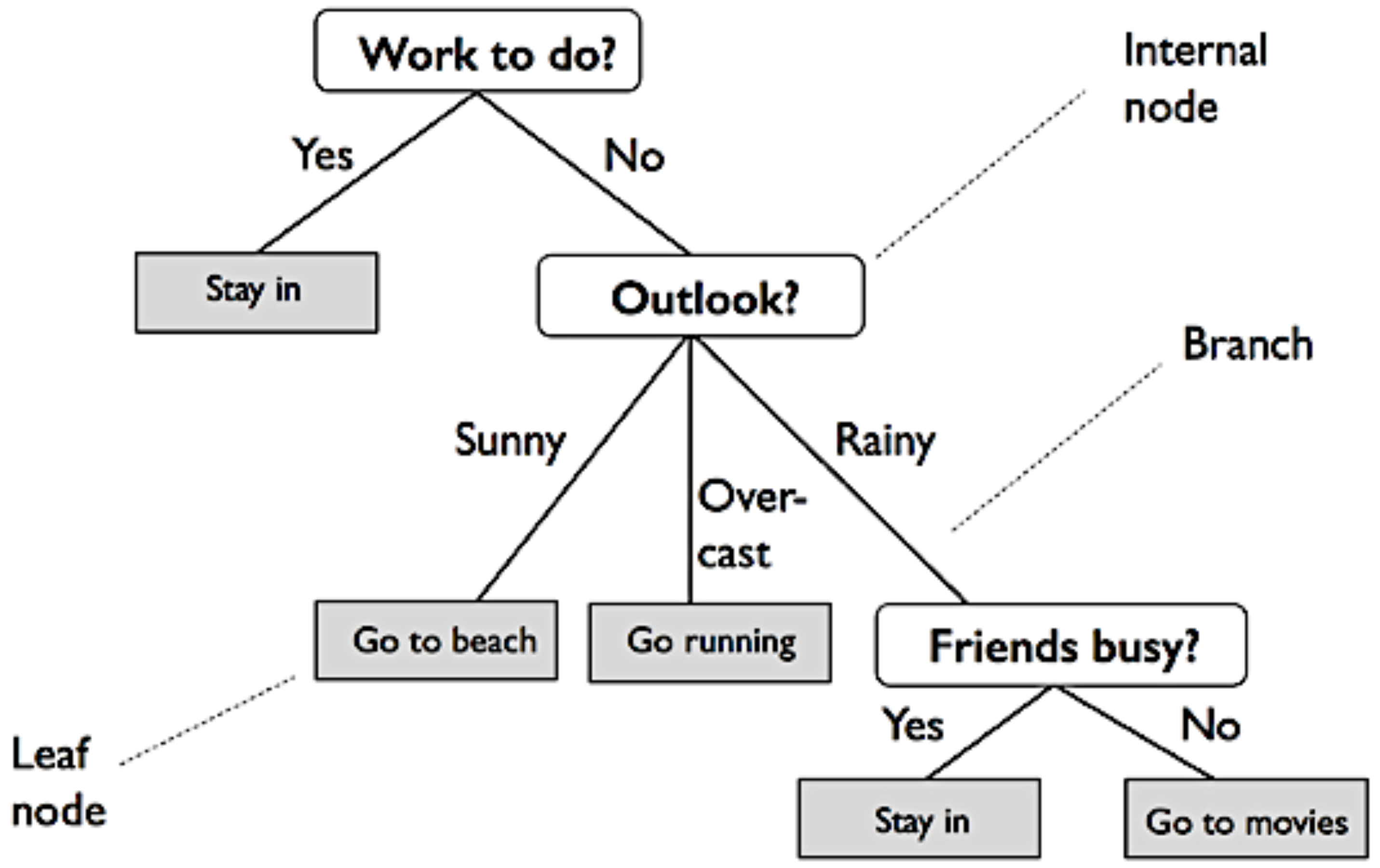
```
svm = SVC(kernel='rbf', random_state=1, gamma=0.10, C=10.0)

svm.fit(X_xor, y_xor)


plot_decision_regions(X_xor, y_xor, classifier=svm,
                    ...test_idx=range(y_train.size,
                         ..y_train.size + y_test.size))

plt.legend(loc='upper left')
plt.show()
```


-1
1
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
4

Decision Tree Learning

- Decision Tree classification

- A classifier that **breaks down a decision** of which class a new sample belongs to through Q&As

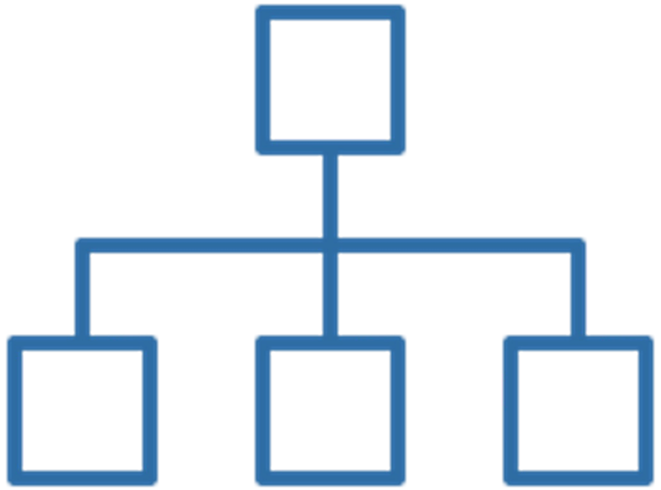


- Decision Tree is a model developed by deducting the class labels of the samples

- The splits are based on features with the largest **Information Gains** (IG)

- Samples @ each node belong to the same class

- Depending on the data: Trees can have a lot of nodes -> **overfitting** -> pruning the tree helps this issue by setting maximum depths (*hyperparameter*)



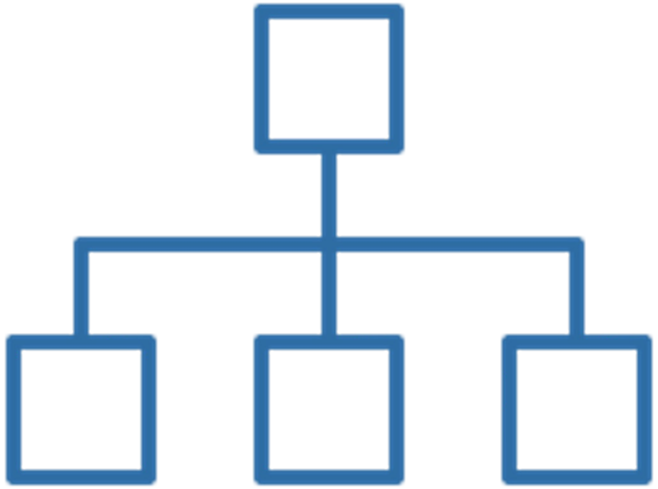
- What makes the tree split?

- **Information Gains** – To know the gain, there must be a calculation relationship between the **parent node** and the **child nodes**.

- The overall idea:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

- Different between the impurities of the parent node and the aggregated child nodes



- *f* – feature (column) that performs the split
- I – Impurity function
- D_P – dataset of the parent
- D_j – dataset of the jth child node
- N_P - # of samples @ parent node
- N_j - # of samples @ the jth child node

$$IG(D_p, f) = \underbrace{I(D_p)}_{\text{Parent's}} - \underbrace{\sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)}_{\text{Children's}}$$

- **Binary Decision Tree**

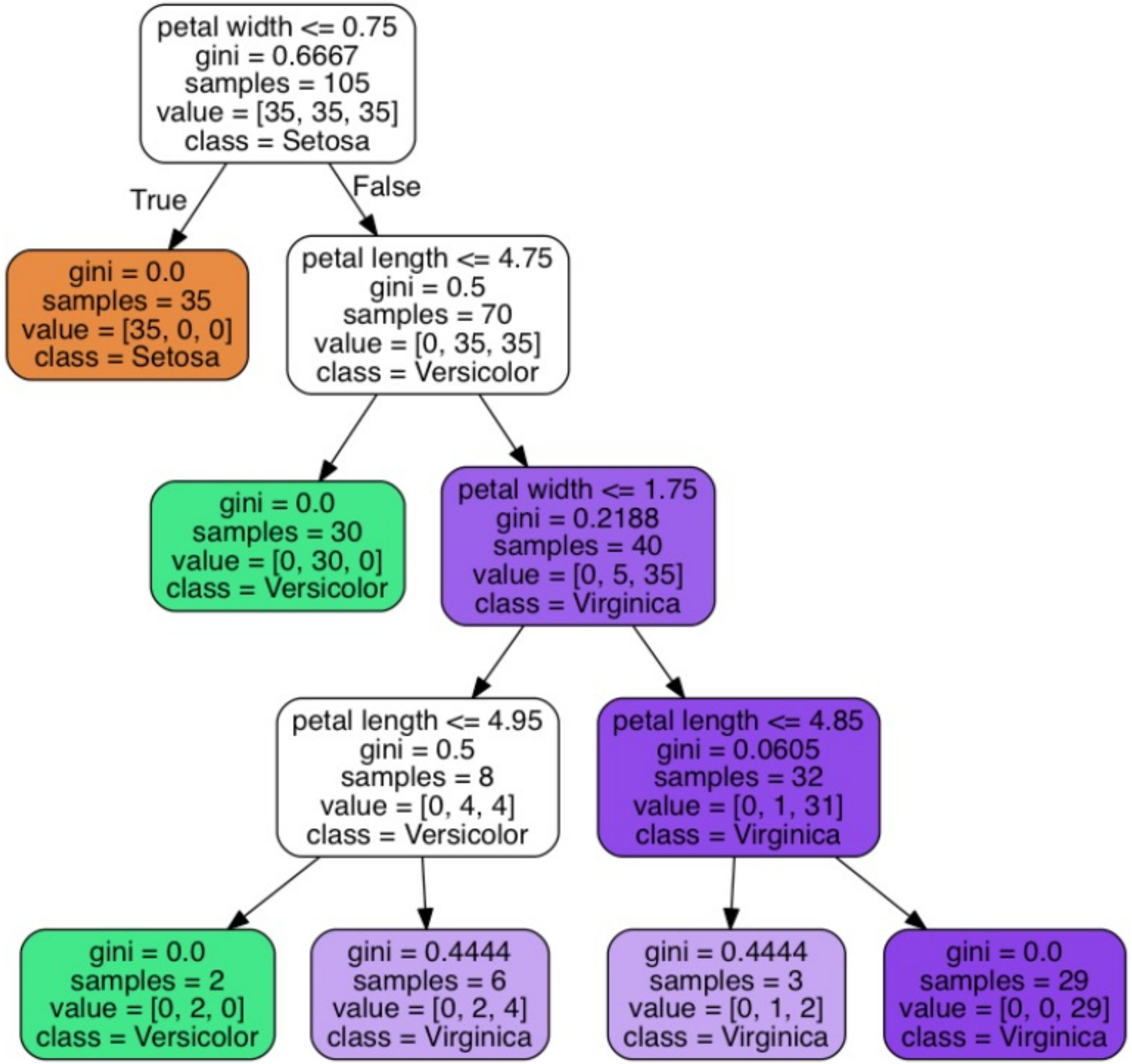
- Easier to implement by most ML libraries

- Each Parent node can only spawn 2 child nodes:

$$IG(D_p, f) = \underbrace{I(D_p)}_{\text{Parent's}} - \underbrace{\frac{N_{left}}{N_p} I(D_{left})}_{\text{Child 1}} - \underbrace{\frac{N_{right}}{N_p} I(D_{right})}_{\text{Child 2}}$$

- **Types of Binary Decision Trees**:
  1. Gini
  2. Entropy
  3. Classification Error

```
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(y_train.size,
                        .. y_train.size + y_test.size))
plt.xlabel('petal length [cm]')
plt.ylabel('petal width [cm]')
plt.legend(loc='upper left')
plt.show()
```



See textbook code example to visualize a decision tree (Chapter 3)

Random Forests

- **Random Forest** is random by nature
- "*Easy*" to implement, hard to explain.
- Basically: A collection of *Decision Trees*
- Each Decision Tree is randomly trained
- **The goal**: Aggregate the accuracy of all the trees

- **Implementation of the Random Forest:**

1. Draw a random **bootstrap** sample of size n (randomly choose n samples from the training set with replacement).
2. Grow a decision tree from the bootstrap sample. At each node:
   a. Randomly select d features without replacement.
   b. Split the node using the feature that provides the best split according to the objective function, for instance, maximizing the information gain.
3. Repeat the steps 1-2 k times.
4. Aggregate the prediction by each tree to assign the class label by **majority vote**.

- **Random Forest Classification:**

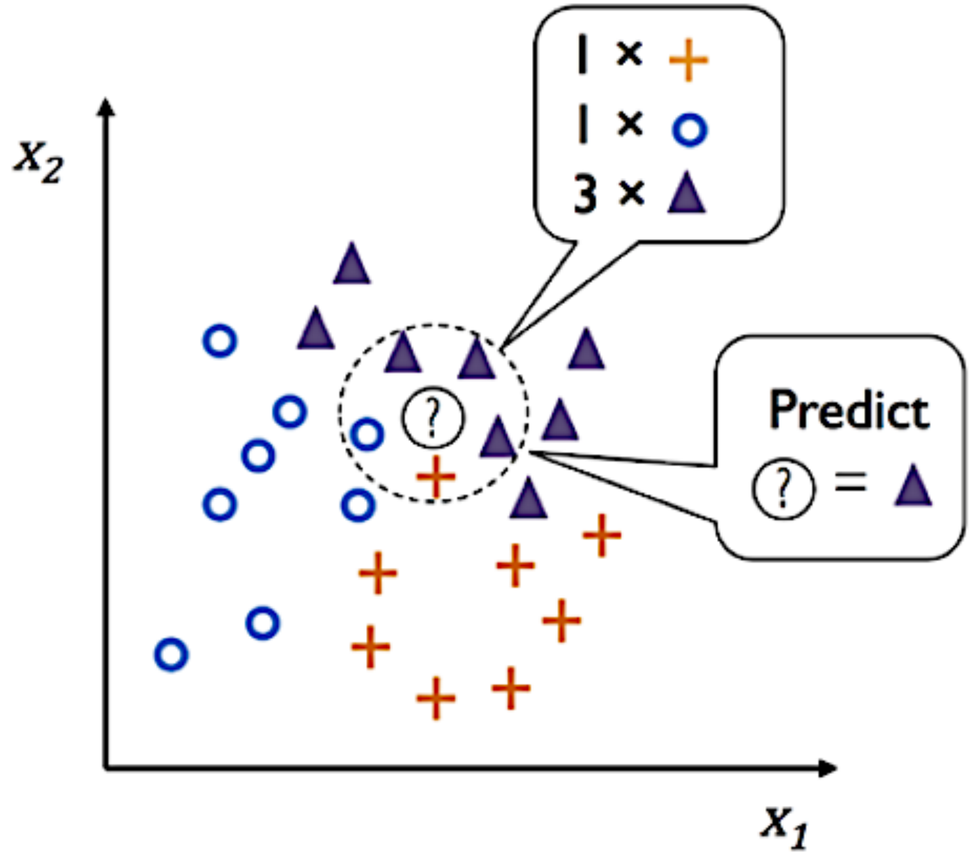
```
from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier(criterion='gini',
     ... n_estimators=25,  #25 random decision trees!
     ... random_state=1)
forest.fit(X_train, y_train)
plot_decision_regions(X_combined, y_combined,
               ... classifier=forest, test_idx= range
                    ..(y_train.size, y_train.size +
                    .. y_test.size))
plt.xlabel('petal length')
plt.ylabel('petal width')
plt.legend(loc='upper left')
plt.show()
```

k-Nearest Neighbors

- *k*-Nearest Neigbor (*kNN*) is a supervised instance-based learning algorithm

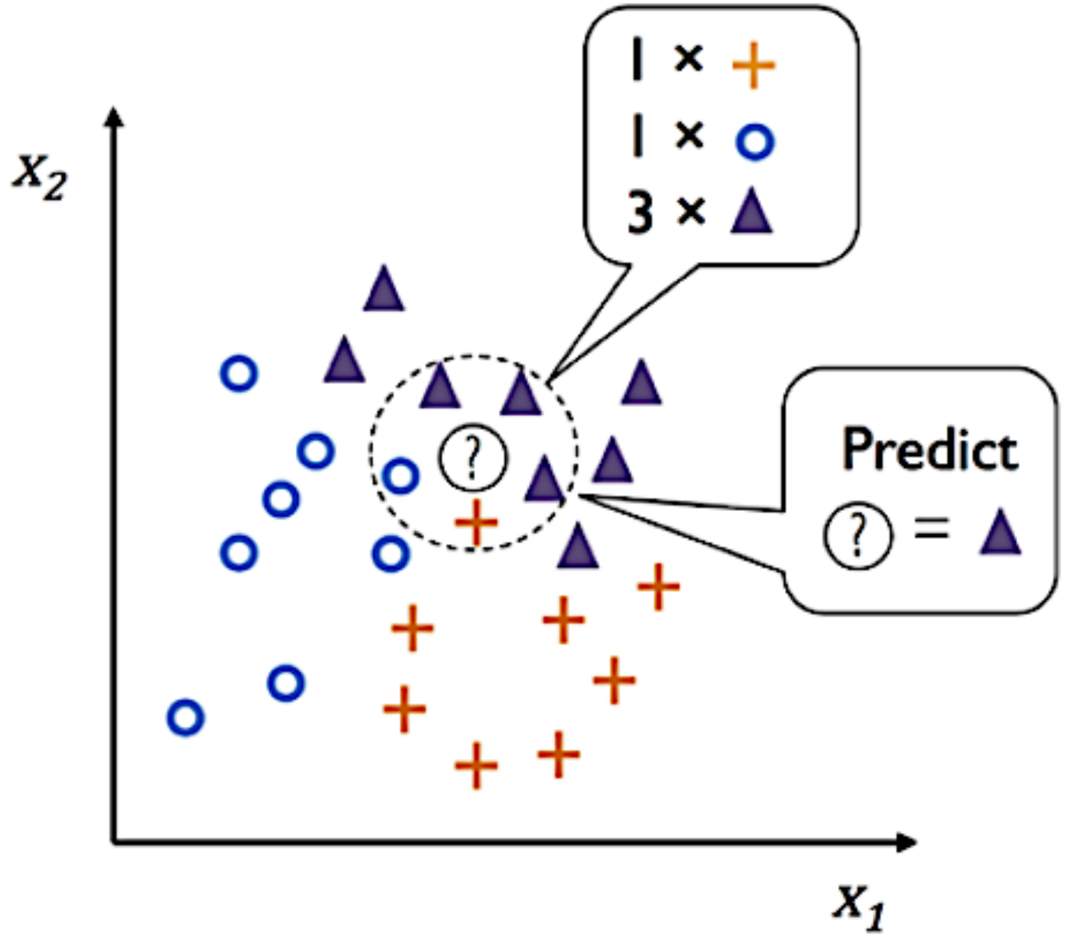
- *k* – is a variable

- **How does it work:**
  1. Choose the # of k and distance metric
  2. Find the *k*-NN of the sample that needs to be classified
  3. Assign the category label based on majority vote



- **The Pro:**
  - New data is easily determined by the votes (adaptable)

- **The Con:**
  - The large number of new samples will take over your physical memory (a linear growth)
  - This will depend on the # of features in the dataset

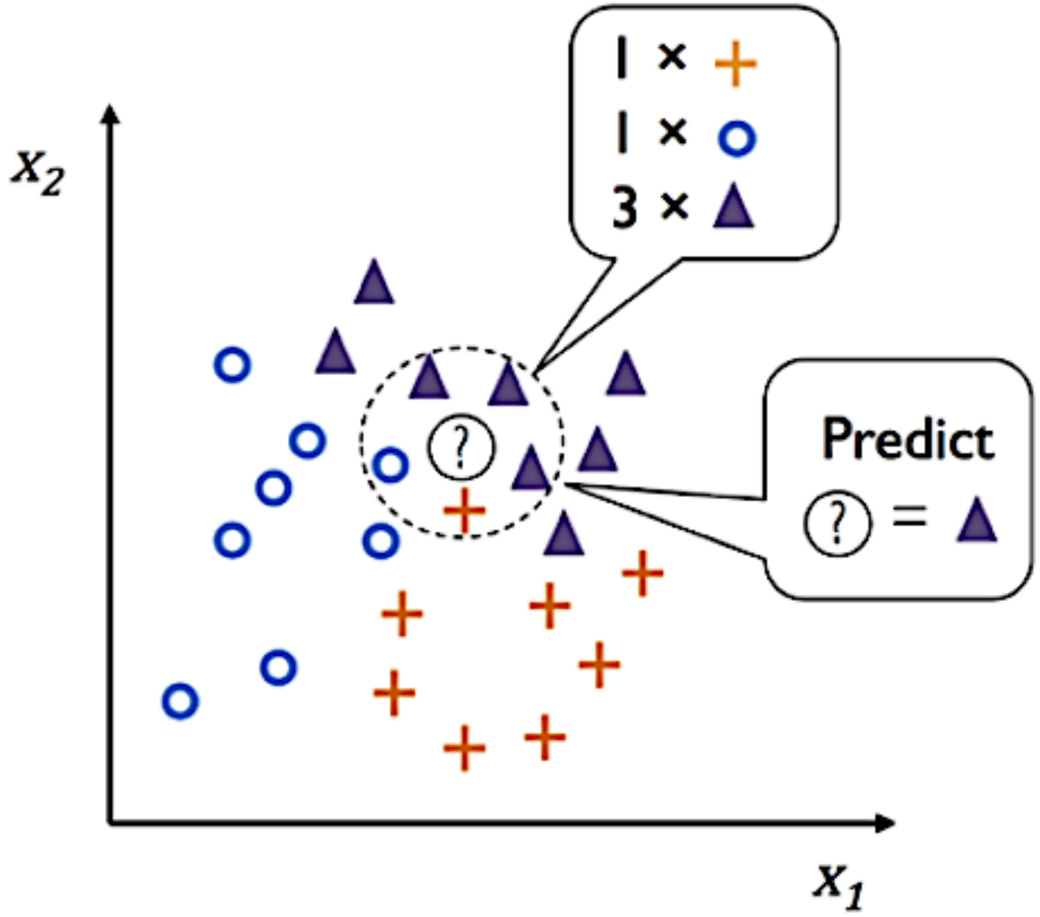


- **k-NN Rules:**

  - If the votes are tied:
    - Distance to the closest
  - If two or more classes have the same distance:
    - Which ever class is listed first in the dataset (*user-determined*)

- The ***k*-factor** determines great balance, overfitting or underfitting.

- Distance is controlled by the **p-parameter** (*in the code*)



- Popular Distances:

Metrics intended for real-valued vector spaces:

identifier	class name	args	distance function
“euclidean”	EuclideanDistance		`sqrt(sum((x - y)^2))`
“manhattan”	ManhattanDistance		`sum(|x - y|)`
“chebyshev”	ChebyshevDistance		`max(|x - y|)`
“minkowski”	MinkowskiDistance	p	`sum(|x - y|^p)^(1/p)`
“wminkowski”	WMinkowskiDistance	p, w	`sum(|w * (x - y)|^p)^(1/p)`
“seuclidean”	SEuclideanDistance	V	`sqrt(sum((x - y)^2 / V))`
“mahalanobis”	MahalanobisDistance	V or VI	`sqrt((x - y)' V^-1 (x - y))`

- $p=1 \rightarrow$ Manhattan
- $p=2 \rightarrow$ Euclidean

```
from sklearn.neighbors import KNeighborsClassifier

# all preprocessing of data goes here…

knn = KNeighborsClassifier(n_neighbors=5, p=2,
                                                ... metric='minkowski’)

# create the classes through training
knn.fit(X_train, y_train)


# vote on unseen data into the trained classes
plot_decision_regions(X_combined, y_combined,
                      ... classifier=knn,
                      ... test_idx=range(y_train.size,
                           … y_train.size + y_test.size))
```

For next class…

- Start reading Chapter 4
- HW# 2 will be posted after class