# 3. Logistic Regression, SVM, Decision Trees, KNN

COMP3314
Machine Learning

# Outline

- Introduction to more robust algorithms for classification
  - Logistic Regression
  - Support Vector Machines
  - Decision Trees
  - KNN
- Examples using the scikit-learn machine learning library
- Strengths and weaknesses of classifiers

# Scikit-learn Library

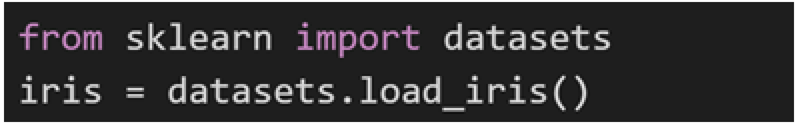
- In the previous chapter we implement the perceptron rule in Python ourselves
- We will now use the scikit-learn library and train a perceptron model similar to the one we implemented in the previous chapter
    - This will serve as an intro to the scikit-learn library

# Code - PerceptronSkLearn.ipynb

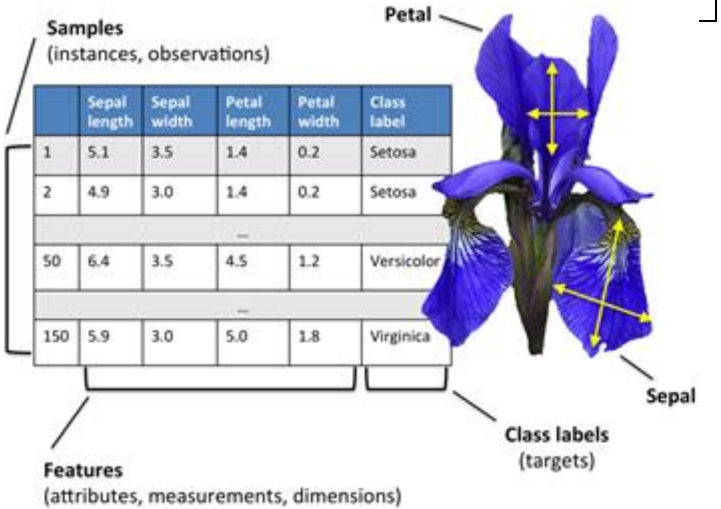
- Available here on CoLab

# Iris Dataset

- Conveniently, the Iris dataset is already available via scikit-learn



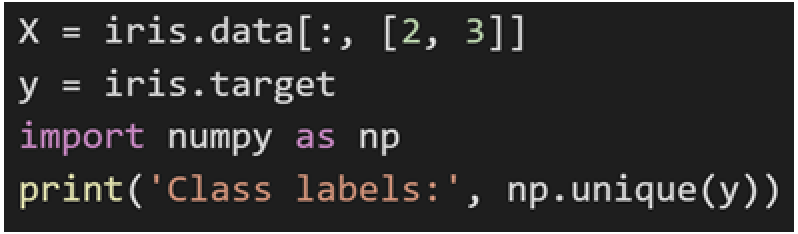
```
from sklearn import datasets
iris = datasets.load_iris()
```



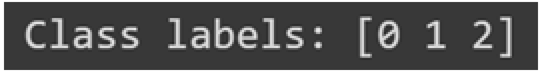
|  | Sepal length | Sepal width | Petal length | Petal width | Class label |
|---|---|---|---|---|---|
| 1 | 5.1 | 3.5 | 1.4 | 0.2 | Setosa |
| 2 | 4.9 | 3.0 | 1.4 | 0.2 | Setosa |
| ... | | | | | |
| 50 | 6.4 | 3.5 | 4.5 | 1.2 | Versicolor |
| ... | | | | | |
| 150 | 5.9 | 3.0 | 5.0 | 1.8 | Virginica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

Samples
(instances, observations)
Petal
Sepal
Class labels
(targets)
Features
(attributes, measurements, dimensions)

- We will only use two features (petal length and petal width) from the Iris dataset for visualization purposes
- Assign all flower samples to the feature matrix **X** and the corresponding class labels of the flower species to the vector **y**



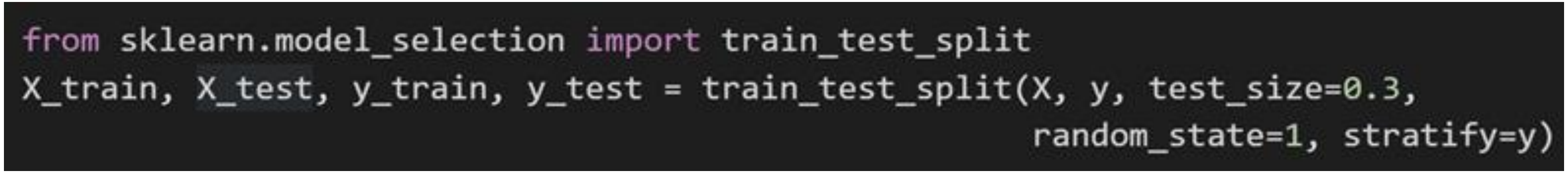
```
X = iris.data[:, [2, 3]]
y = iris.target
import numpy as np
print('Class labels:', np.unique(y))
```



```
Class labels: [0 1 2]
```

# Training and Testing

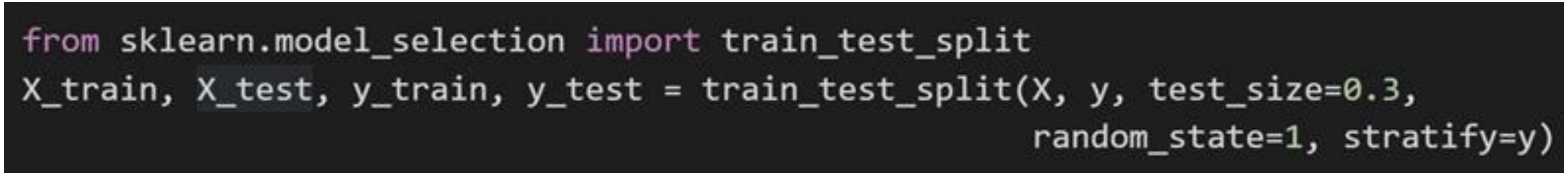
- To evaluate how well a trained model performs on unseen data, we will further split the dataset into separate training and test datasets



```
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)
```

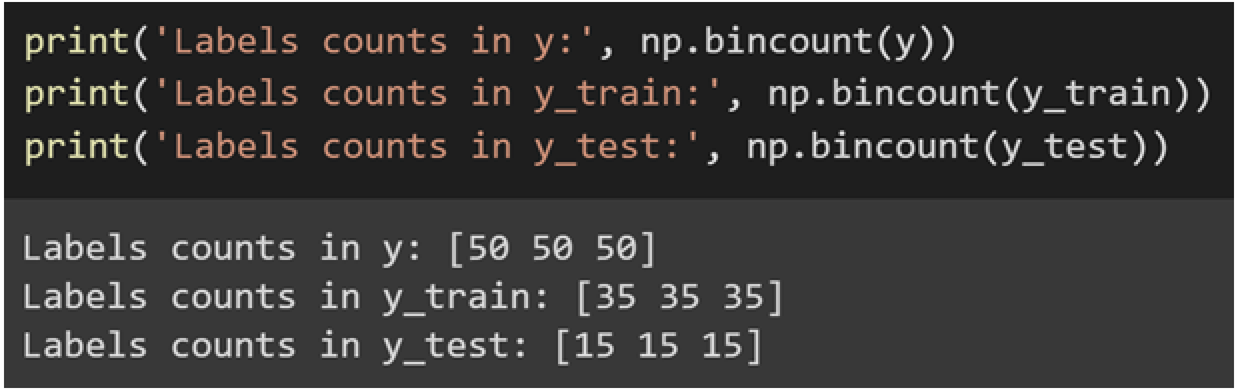
- Note that the train_test_split function shuffles the training sets internally and performs stratification before splitting
  - Otherwise, all class 0 and class 1 samples would have ended up in the training set, and the test set would consist of 45 samples from class 2

# Stratification



```
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)
```

- We took advantage of the built-in support for stratification
- In this context, stratification means that the train_test_split method returns training and test subsets that have the same proportions of class labels as the input dataset



```
print('Labels counts in y:', np.bincount(y))
print('Labels counts in y_train:', np.bincount(y_train))
print('Labels counts in y_test:', np.bincount(y_test))
```

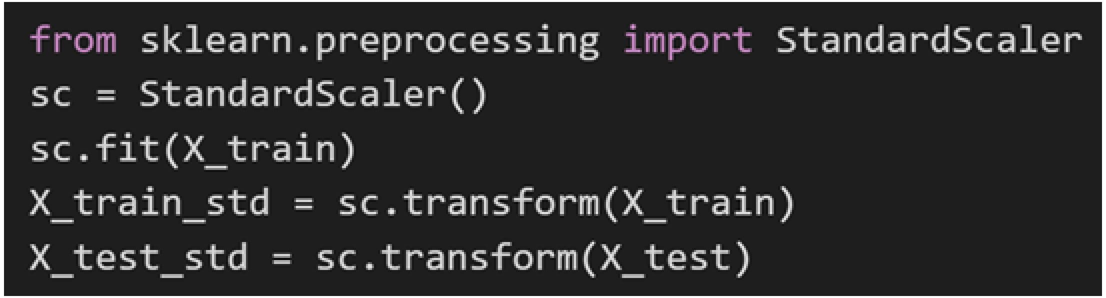
```
Labels counts in y: [50 50 50]
Labels counts in y_train: [35 35 35]
Labels counts in y_test: [15 15 15]
```

# Multiclass Classification

- n (number of classes) > 2
- Some classification algorithms naturally permit n > 2
  - Others are by nature binary algorithms
- OvA or One-versus-Rest (OvR)
  - Train one classifier per class, where the particular class is treated as the positive class
  - Samples from all other classes are considered negative classes
- If we were to classify a new data sample, we would use our n classifiers, and assign the class label with the highest confidence to the particular sample

# Feature Scaling

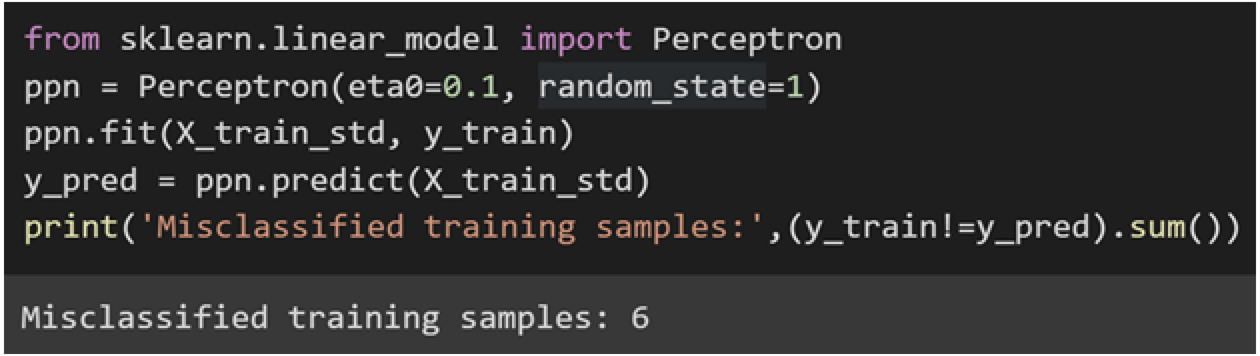
- Recall that many machine learning and optimization algorithms also require feature scaling for optimal performance
- Here, we will standardize the features using the StandardScaler class from scikit-learn's preprocessing module



```
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)
```

# Training

- We can now train a perceptron model
- Most algorithms in scikit-learn already support multiclass classification by default via the One-versus-Rest (OvR) method, which allows us to feed the three flower classes to the perceptron all at once



```
from sklearn.linear_model import Perceptron
ppn = Perceptron(eta0=0.1, random_state=1)
ppn.fit(X_train_std, y_train)
y_pred = ppn.predict(X_train_std)
print('Misclassified training samples:',(y_train!=y_pred).sum())
```

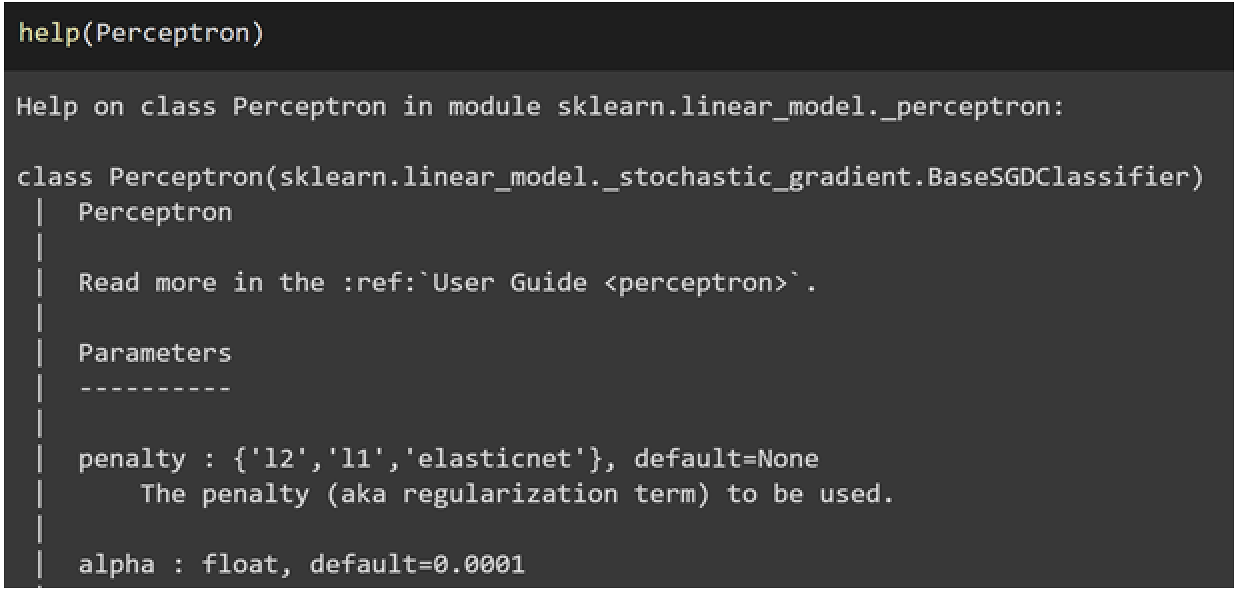
```
Misclassified training samples: 6
```

# Scikit-Learn Help

- The Perceptron, as well as other scikit-learn functions and classes, often have additional parameters that we omit for clarity
- You can read more about those parameters using the help function

# Scikit-Learn Help

- The Perceptron, as well as other scikit-learn functions and classes, often have additional parameters that we omit for clarity
- You can read more about those parameters using the help function



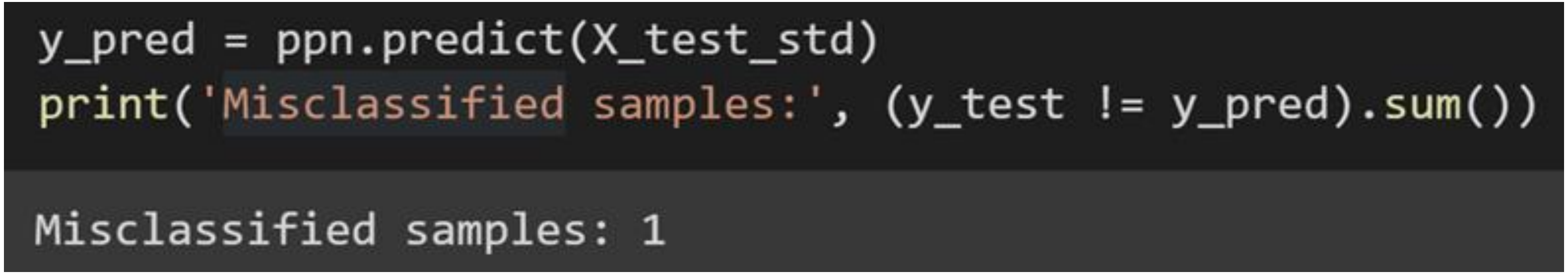
```
help(Perceptron)
```

```
Help on class Perceptron in module sklearn.linear_model._perceptron:

class Perceptron(sklearn.linear_model._stochastic_gradient.BaseSGDClassifier)
 |  Perceptron
 |
 |  Read more in the :ref:`User Guide <perceptron>`.
 |
 |  Parameters
 |  ----------
 |
 |  penalty : {'l2','l1','elasticnet'}, default=None
 |      The penalty (aka regularization term) to be used.
 |
 |  alpha : float, default=0.0001
```

# Testing

- Having trained a model in scikit-learn, we can make predictions via the predict method on the test data

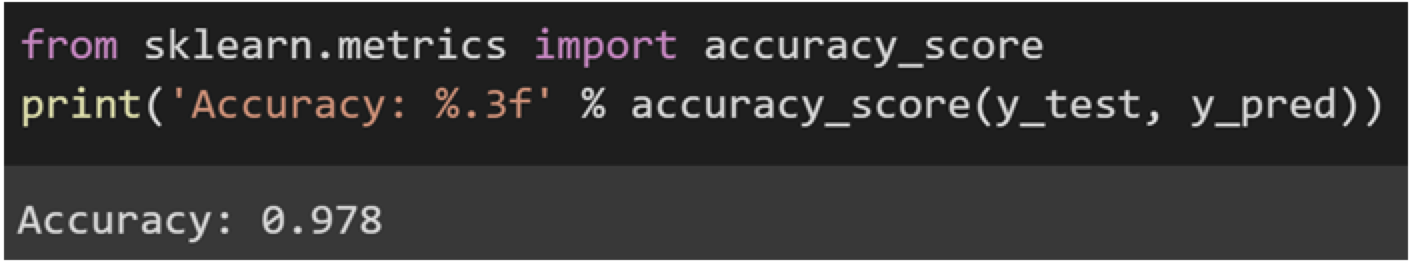


```
y_pred = ppn.predict(X_test_std)
print('Misclassified samples:', (y_test != y_pred).sum())
```

```
Misclassified samples: 1
```

# Testing - Accuracy

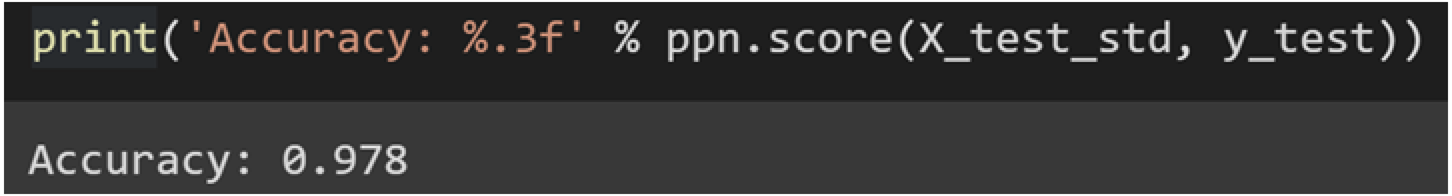
- The scikit-learn library also implements a large variety of different performance metrics that are available via the metrics module
  - We can calculate the classification accuracy as follows



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

```
Accuracy: 0.978
```

  - Alternatively, each classifier in scikit-learn has a score method



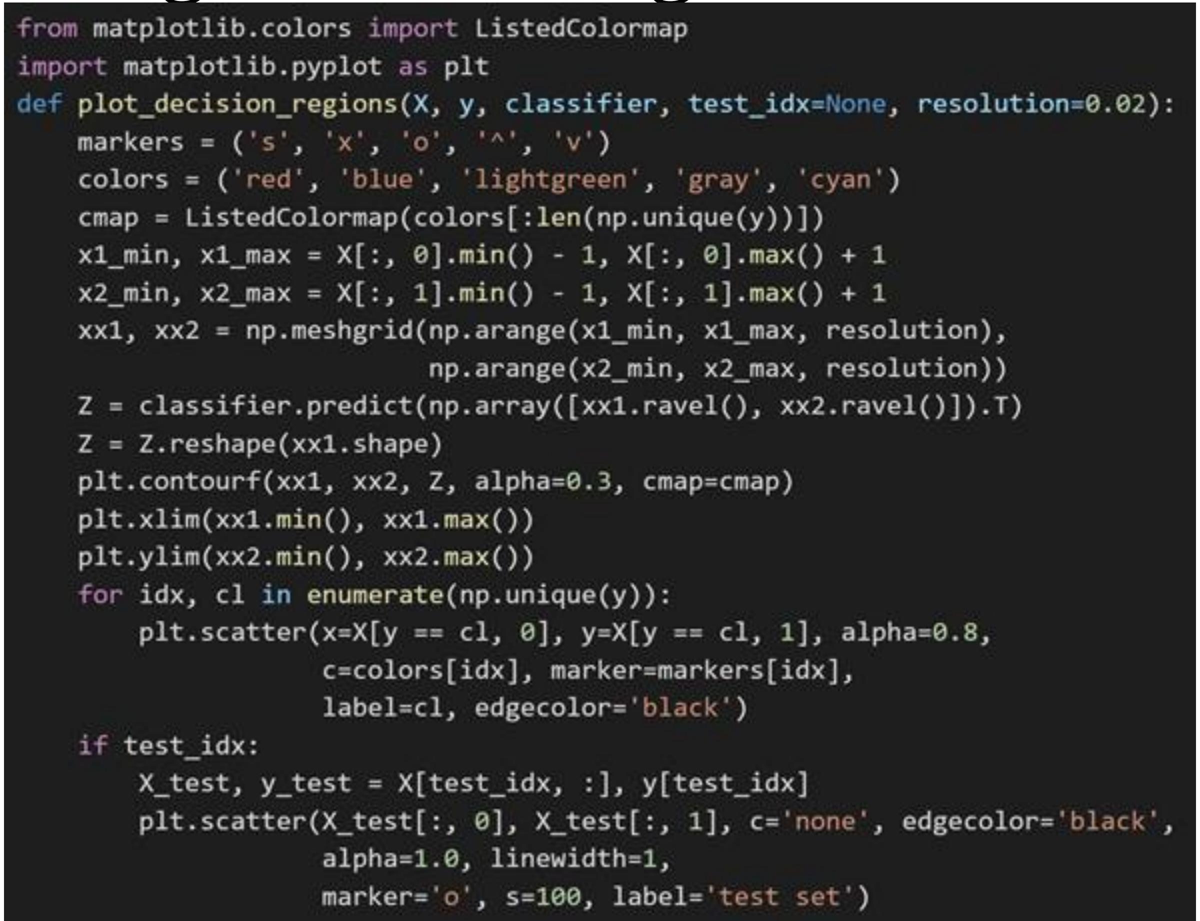
```
print('Accuracy: %.3f' % ppn.score(X_test_std, y_test))
```

```
Accuracy: 0.978
```

# Decision Regions Plotting

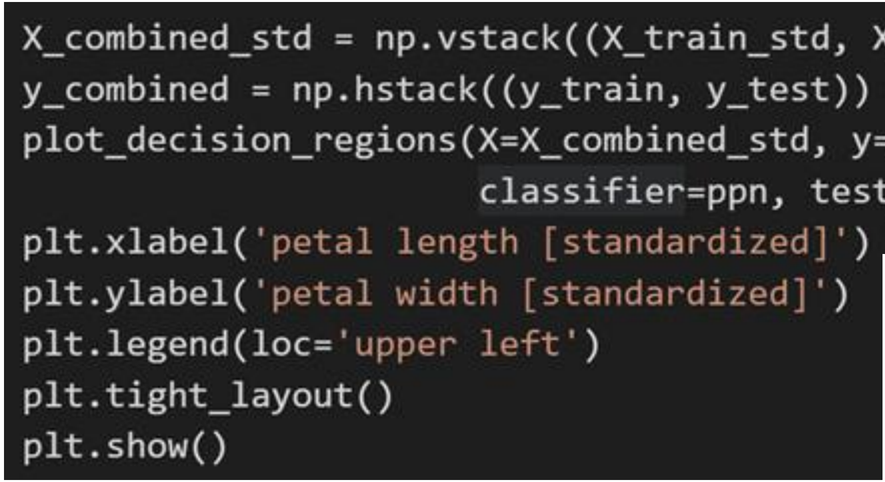
- Finally, we can use our plot_decision_regions function to plot the decision regions of our newly trained perceptron model
    - Visualize how well it separates the different flower samples
- However, let's add a small modification to highlight the samples from the test dataset via small circles

# Decision Regions Plotting

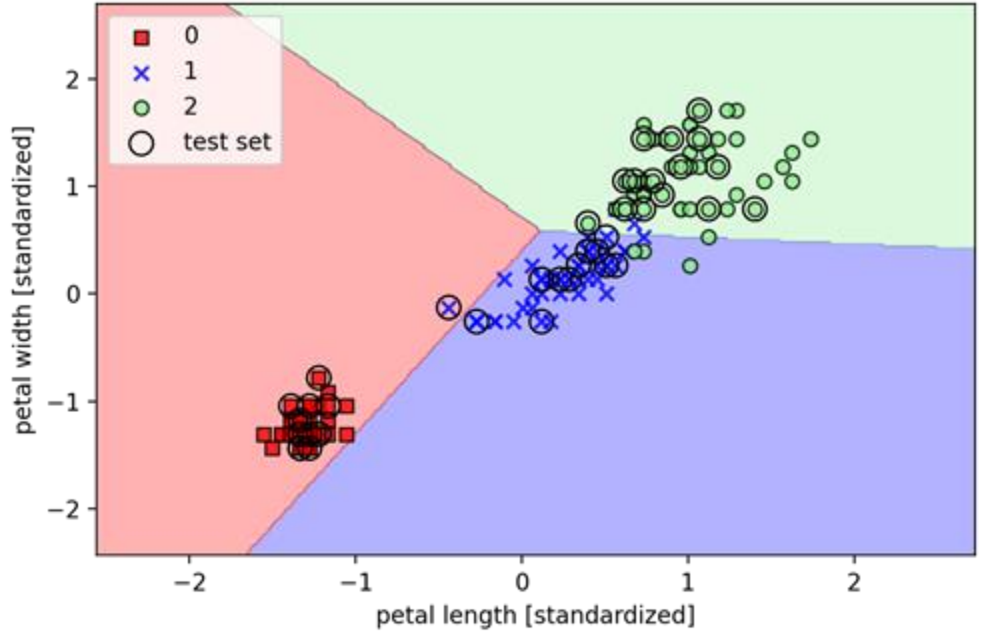


```python
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1], alpha=0.8,
                    c=colors[idx], marker=markers[idx],
                    label=cl, edgecolor='black')
    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1], c='none', edgecolor='black',
                    alpha=1.0, linewidth=1,
                    marker='o', s=100, label='test set')
```

# Decision Regions Plotting



```
X_combined_std = np.vstack((X_train_std, X_test_std))
y_combined = np.hstack((y_train, y_test))
plot_decision_regions(X=X_combined_std, y=y_combined,
                      classifier=ppn, 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()
```



# Perceptron

- The three flower classes cannot be perfectly separated by a linear decision boundary
- Recall that the perceptron algorithm never converges on datasets that aren't perfectly linearly separable
- In the following sections, we will look at more powerful linear classifiers that converge to a cost minimum even if the classes are not perfectly linearly separable

# Logistic Regression - Intuition

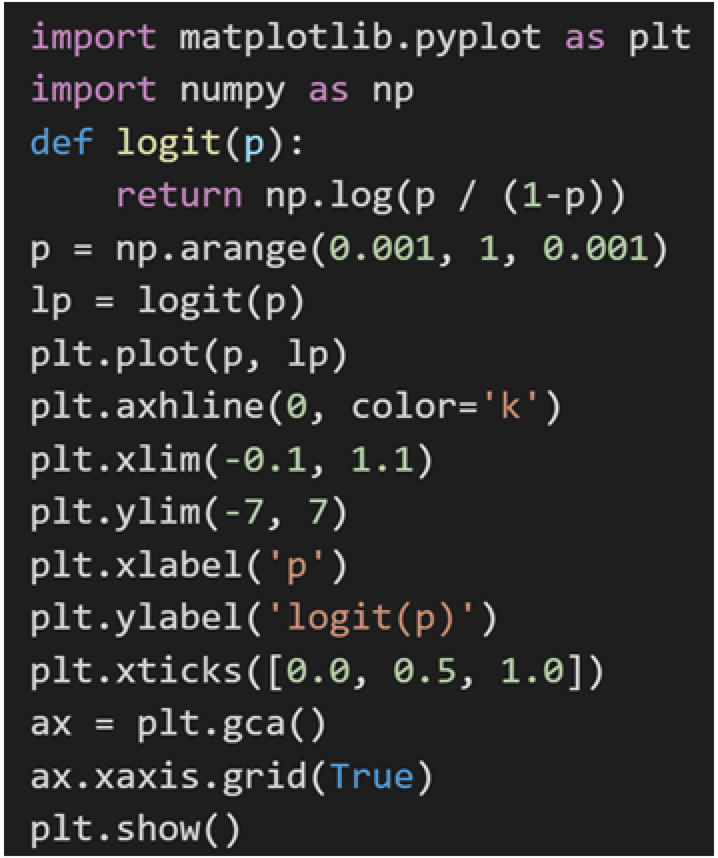
- Widely used algorithm for binary classification
  - For classification, not regression
- The logit function is defined as the logarithm of odds, i.e.,

$$logit\left(p\right) = log\frac{p}{\left(1-p\right)}$$

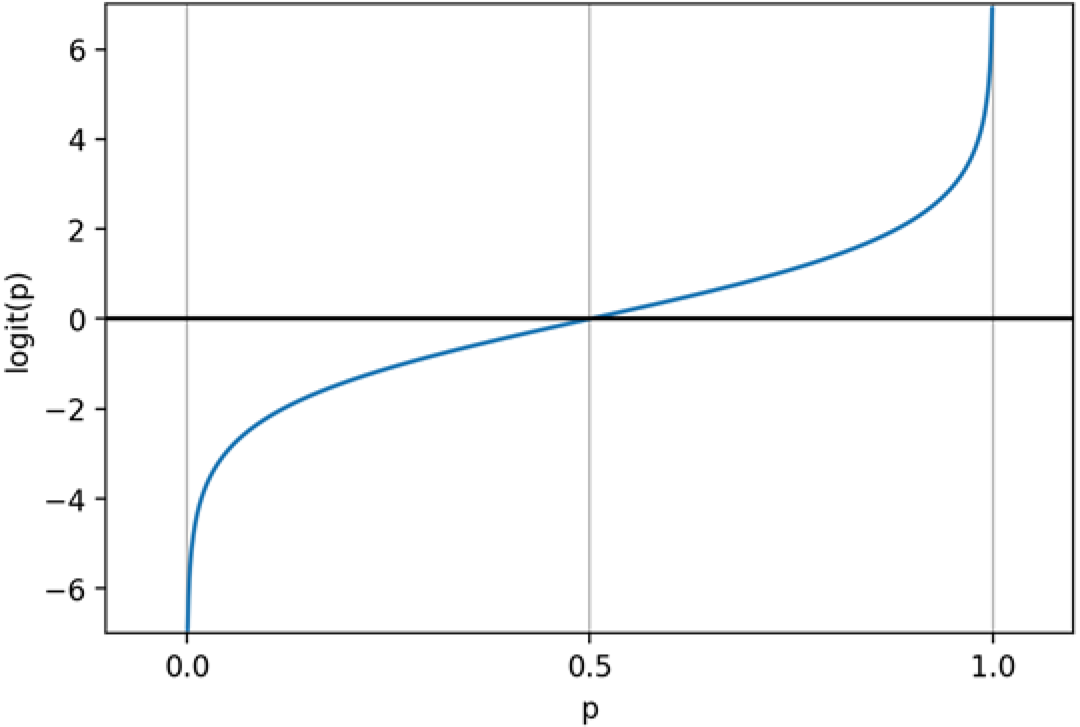
- It takes as input values in the range 0 to 1 and transforms them to values over the entire real-number range

# Logit Function

$$logit(p) = log\frac{p}{(1-p)}$$



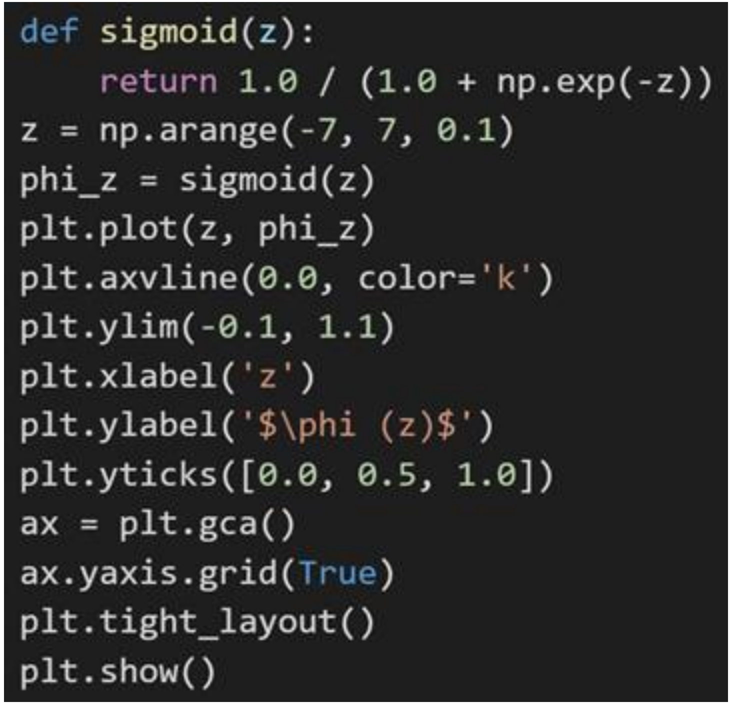
```
import matplotlib.pyplot as plt
import numpy as np
def logit(p):
    return np.log(p / (1-p))
p = np.arange(0.001, 1, 0.001)
lp = logit(p)
plt.plot(p, lp)
plt.axhline(0, color='k')
plt.xlim(-0.1, 1.1)
plt.ylim(-7, 7)
plt.xlabel('p')
plt.ylabel('logit(p)')
plt.xticks([0.0, 0.5, 1.0])
ax = plt.gca()
ax.xaxis.grid(True)
plt.show()
```



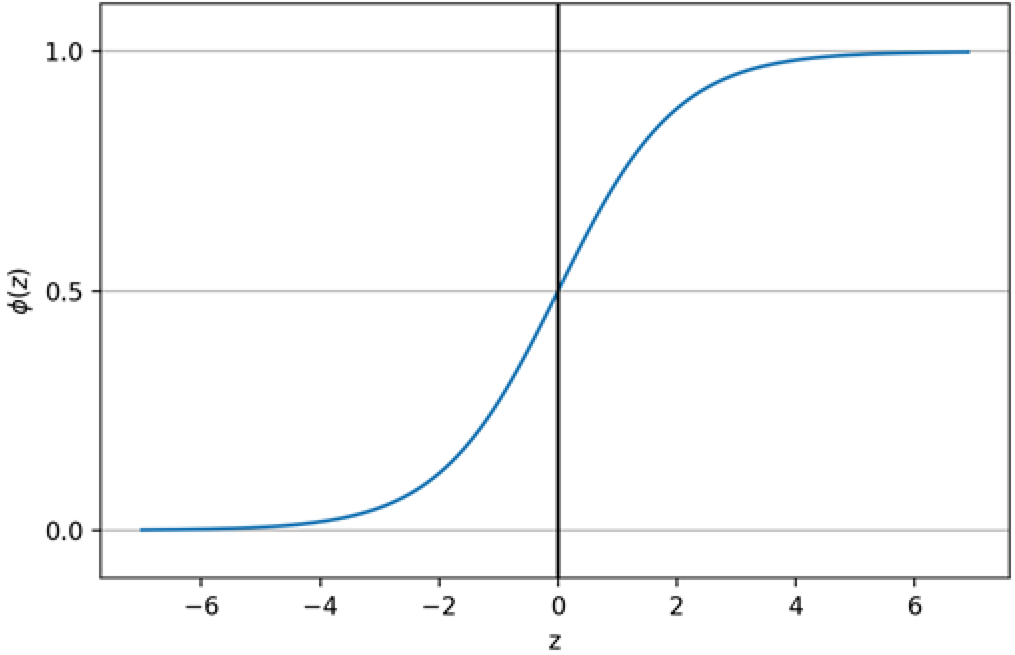
# Inverse of Logit (Sigmoid) Function

$$logit(p) = log\frac{p}{(1-p)}$$

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



```
def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))
z = np.arange(-7, 7, 0.1)
phi_z = sigmoid(z)
plt.plot(z, phi_z)
plt.axvline(0.0, color='k')
plt.ylim(-0.1, 1.1)
plt.xlabel('z')
plt.ylabel('$\phi (z)$')
plt.yticks([0.0, 0.5, 1.0])
ax = plt.gca()
ax.yaxis.grid(True)
plt.tight_layout()
plt.show()
```



# Logistic Regression - Intuition

- The output of the logit function can be used to express a linear relationship between feature values and the log-odds

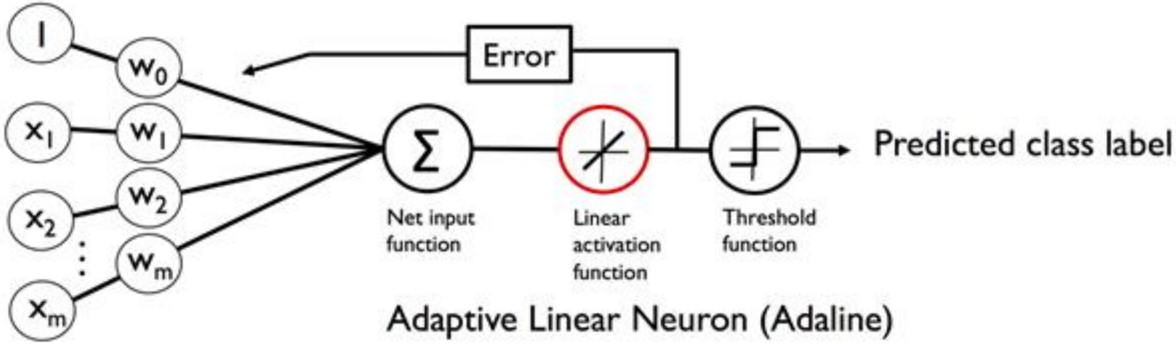
$$logit\left(p\left(y=1\mid \boldsymbol{x}\right)\right)=w_{\mathbf{0}}x_{\mathbf{0}}+w_{\mathbf{1}}x_{\mathbf{1}}+\cdots+w_{m}x_{m}=\sum_{i=0}^{m}w_{i}x_{i}=\boldsymbol{w}^{T}\boldsymbol{x}$$

- p ( y = 1| **x** ) is the conditional probability that a particular sample belongs to class 1 given its features **x**

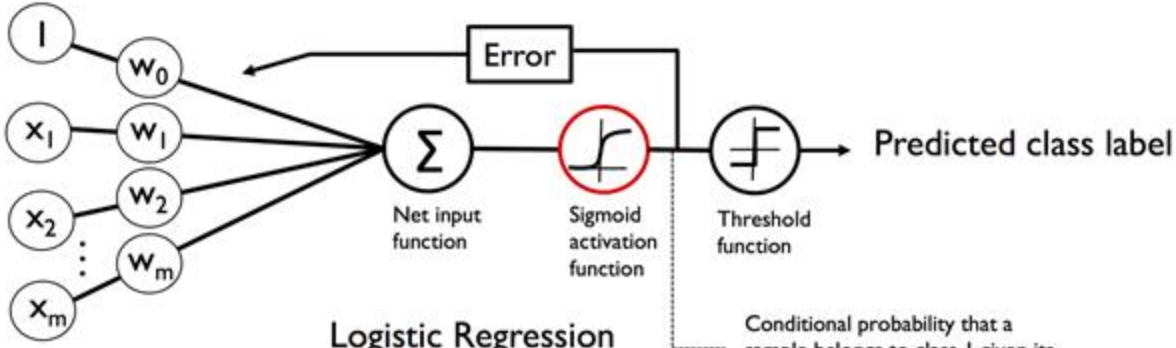
$$\phi\left(z\right)=\frac{1}{1+e^{-z}}$$

$$\mathrm{z}=\boldsymbol{w}^{T}\boldsymbol{x}=w_{0}x_{0}+w_{\mathbf{1}}x_{\mathbf{1}}+\cdots+w_{\boldsymbol{m}}x_{\boldsymbol{m}}$$

# Adaline vs. Logistic Regression



Adaptive Linear Neuron (Adaline)



Logistic Regression

Conditional probability that a sample belongs to class 1 given its input vector x

# Output of Sigmoid Function

- The output of the sigmoid function is interpreted as the probability of a particular sample belonging to class 1
  ϕ(z) = P ( y = 1| **x** ; **w** ), given its features **x** parameterized by the weights **w**
  - E.g, let ϕ ( z ) = 0.8 for a particular flower sample. I.e., the probability that this sample is an Iris-versicolor flower is 80 %
  - The probability that this flower is an Iris-setosa flower can be calculated as P ( y = 0 | **x** ; **w** ) = 1 − P ( y = 1| **x** ; **w** ) = 0.2
    - The predicted probability can then simply be converted into a binary outcome via a threshold function

$$\hat{y} = \begin{cases} 1 & if\, \phi(z) \geq 0.5 \\ 0 & otherwise \end{cases}$$

# Learning **w**

- Recall that we defined the sum-squared-error cost function as follows

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

  - We minimized this function in order to learn the weights **w** for our Adaline classification model
- For logistic regression, we define the likelihood L that we want to maximize as

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

# Learning **w**

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

- In practice, it is easier to maximize the (natural) log of this equation, which is called the log-likelihood function

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

- Let's rewrite the log-likelihood as a cost function that can be minimized using gradient descent

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

# Learning **w**

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

- Let us take a look at the cost that we calculate for one single training sample

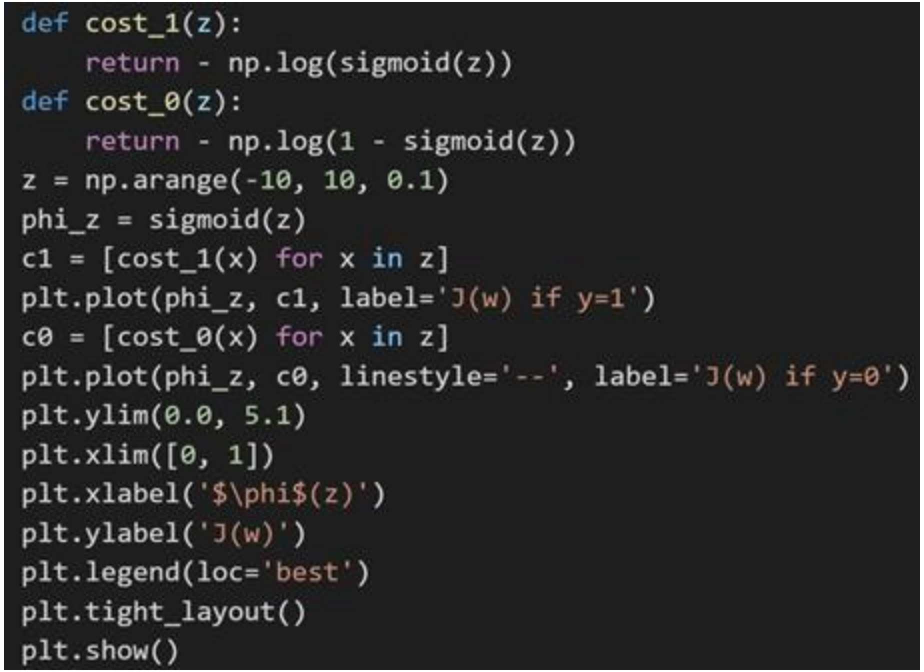
$$J\left(\phi(z), y; \boldsymbol{w}\right) = -y\log\left(\phi(z)\right) - (1-y)\log\left(1-\phi(z)\right)$$

- We can see that the first term is zero if y = 0, and the second term is zero if y = 1

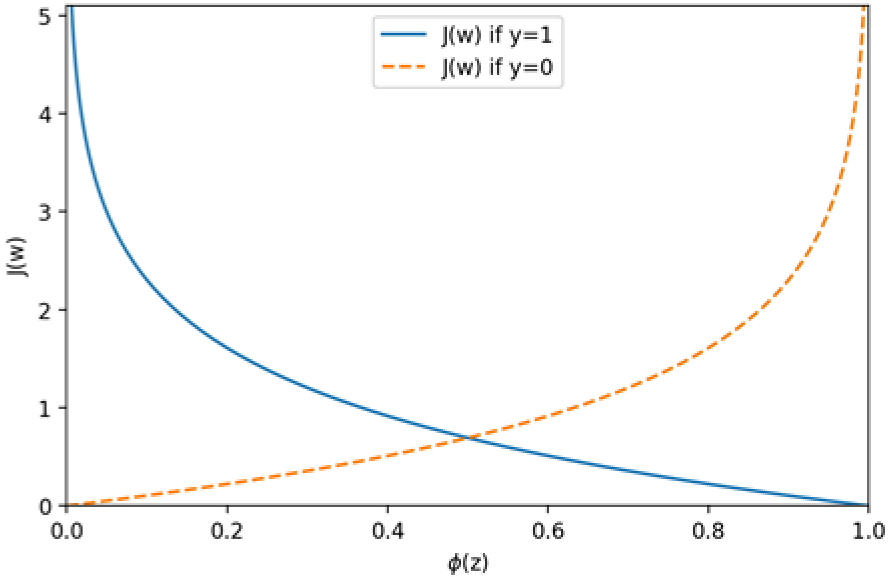
$$J\left(\phi(z), y; \boldsymbol{w}\right) = \begin{cases} -\log\left(\phi(z)\right) & \text{if } y = 1 \\ -\log\left(1-\phi(z)\right) & \text{if } y = 0 \end{cases}$$

# Plotting our new J

$$J\left(\phi(z), y; w\right) = \begin{cases} -\log\left(\phi(z)\right) & \text{if } y = 1 \\ -\log\left(1 - \phi(z)\right) & \text{if } y = 0 \end{cases}$$



```
def cost_1(z):
    return - np.log(sigmoid(z))
def cost_0(z):
    return - np.log(1 - sigmoid(z))
z = np.arange(-10, 10, 0.1)
phi_z = sigmoid(z)
c1 = [cost_1(x) for x in z]
plt.plot(phi_z, c1, label='J(w) if y=1')
c0 = [cost_0(x) for x in z]
plt.plot(phi_z, c0, linestyle='--', label='J(w) if y=0')
plt.ylim(0.0, 5.1)
plt.xlim([0, 1])
plt.xlabel('$\phi$(z)')
plt.ylabel('J(w)')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```

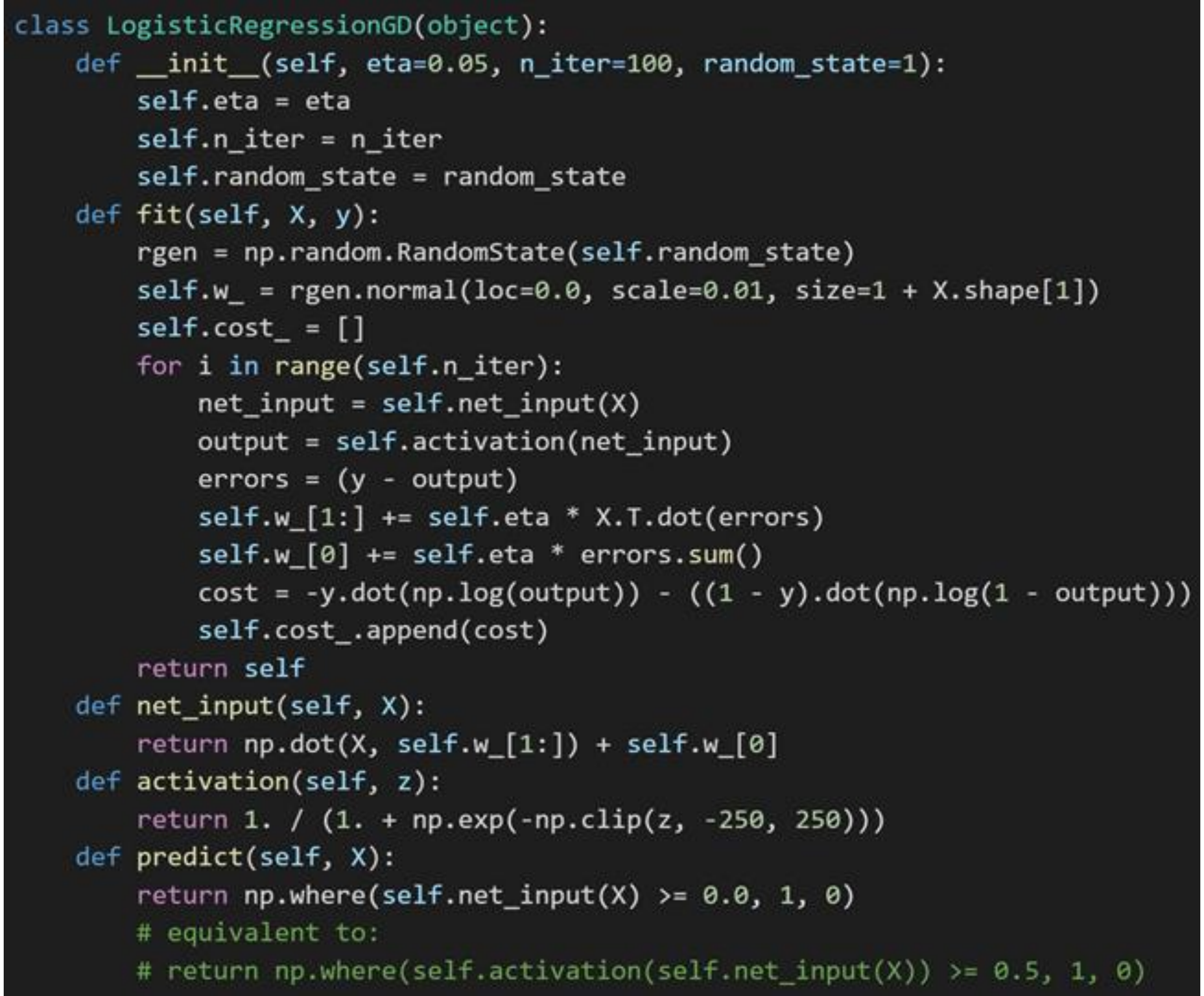


# From Adaline to Logistic Regression

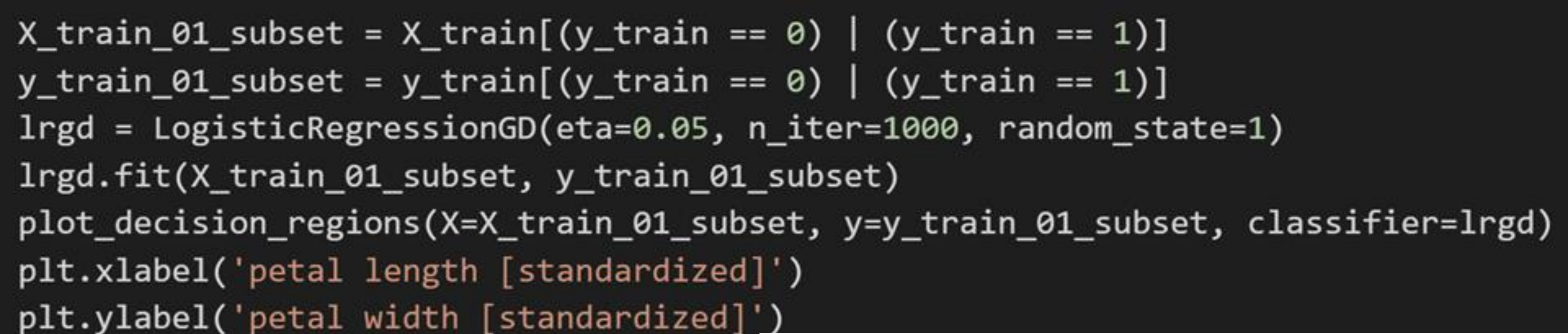
- If we were to implement logistic regression ourselves, we could simply substitute the cost function J in our Adaline implementation from the previous chapter

# Code - LogisticRegression.ipynb

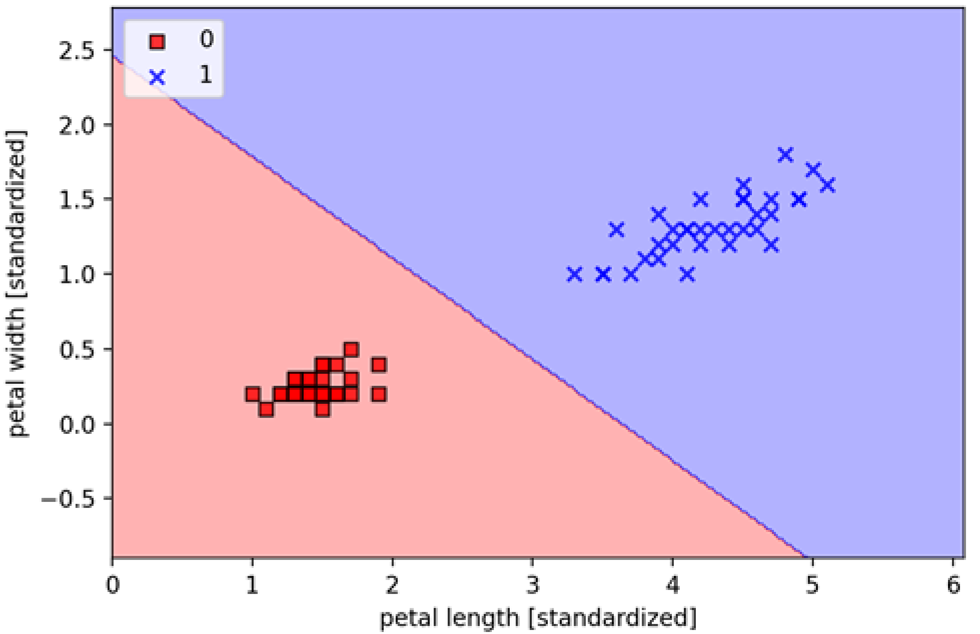
- Available here on CoLab



```
class LogisticRegressionGD(object):
    def __init__(self, eta=0.05, n_iter=100, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
    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 i 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()
            cost = -y.dot(np.log(output)) - ((1 - y).dot(np.log(1 - output)))
            self.cost_.append(cost)
        return self
    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]
    def activation(self, z):
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))
    def predict(self, X):
        return np.where(self.net_input(X) >= 0.0, 1, 0)
        # equivalent to:
        # return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
```

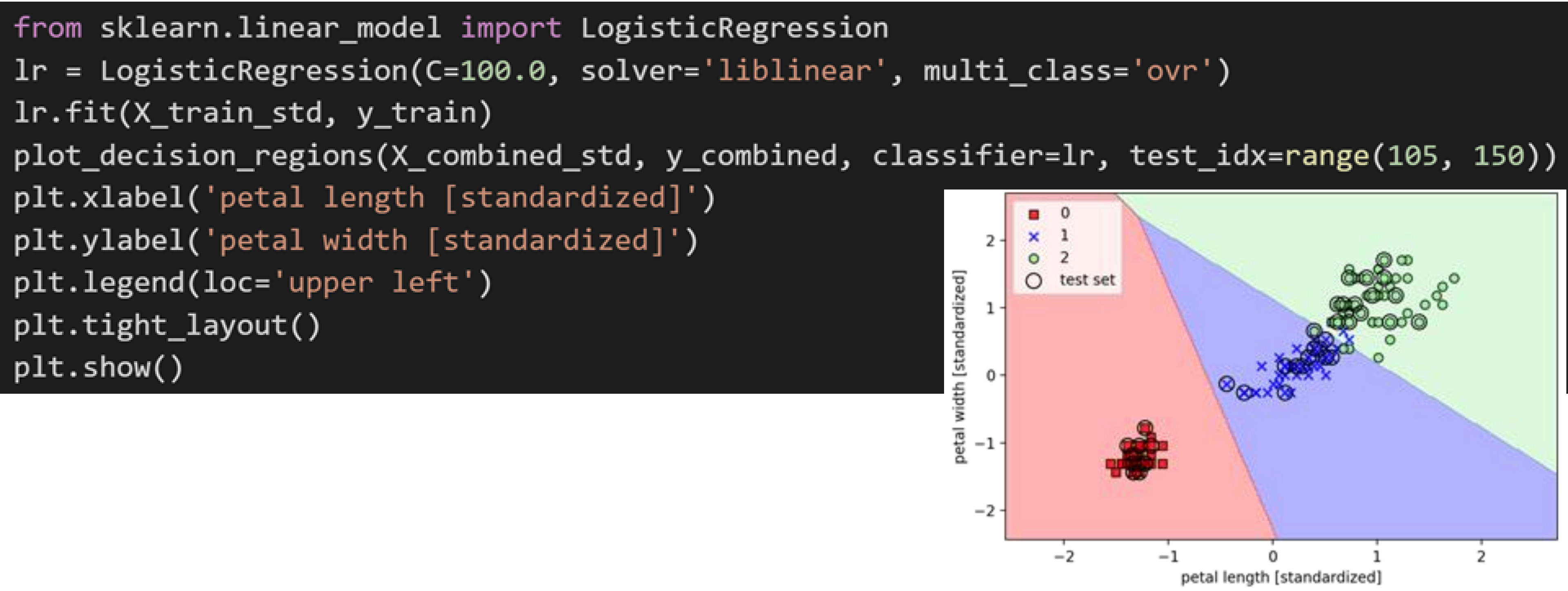


```
X_train_01_subset = X_train[(y_train == 0) | (y_train == 1)]
y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]
lrgd = LogisticRegressionGD(eta=0.05, n_iter=1000, random_state=1)
lrgd.fit(X_train_01_subset, y_train_01_subset)
plot_decision_regions(X=X_train_01_subset, y=y_train_01_subset, classifier=lrgd)
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```



# Logistic Regression with scikit-learn

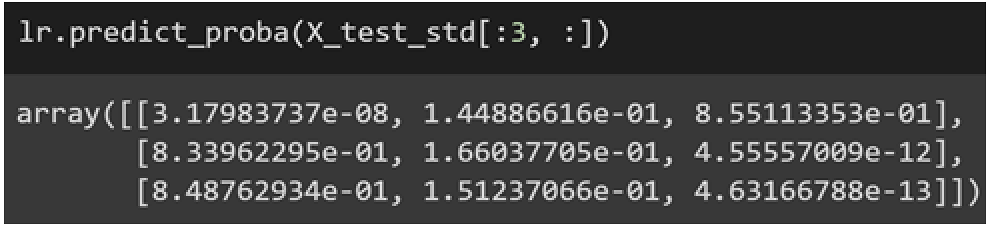
- Scikit-learn's implementation of logistic regression also supports multi-class settings off the shelf (OvR by default)



```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=100.0, solver='liblinear', 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()
```

# Prediction

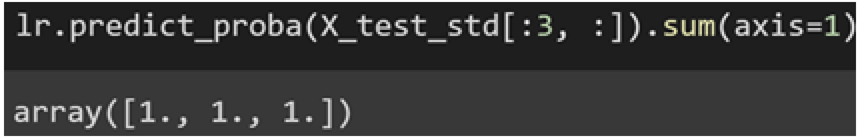
- The probability that training examples belong to a certain class can be computed using the predict_proba method
- For example, we can predict the probabilities of the first three samples in the test set as follows



```
lr.predict_proba(X_test_std[:3, :])
```

```
array([[3.17983737e-08, 1.44886616e-01, 8.55113353e-01],
       [8.33962295e-01, 1.66037705e-01, 4.55557009e-12],
       [8.48762934e-01, 1.51237066e-01, 4.63166788e-13]])
```

- Notice that for each row the columns sum all up to one

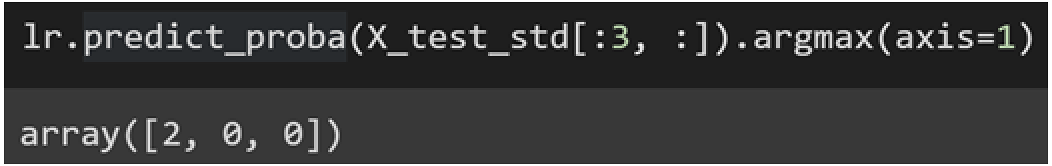


```
lr.predict_proba(X_test_std[:3, :]).sum(axis=1)
```

```
array([1., 1., 1.])
```

# Prediction

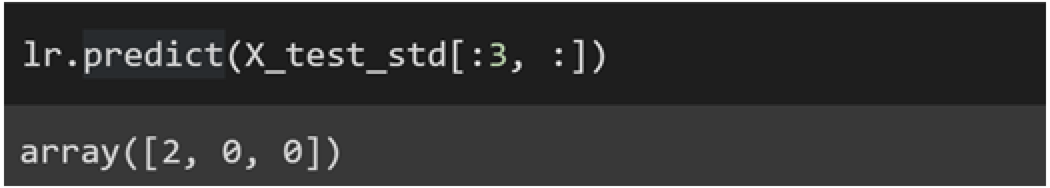
- We can get the predicted class labels by identifying the largest column in each row, for example, using NumPy's argmax function



```
lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

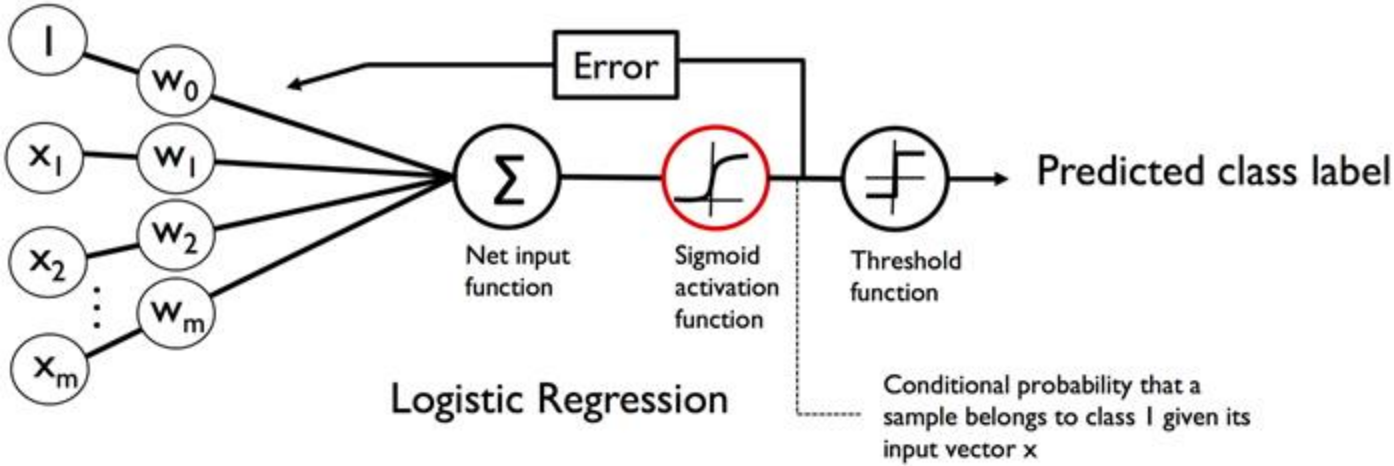
```
array([2, 0, 0])
```

- The returned class indices correspond to Iris-virginica, Iris-setosa, and Iris-setosa
- This is just a manual approach to calling the predict method directly



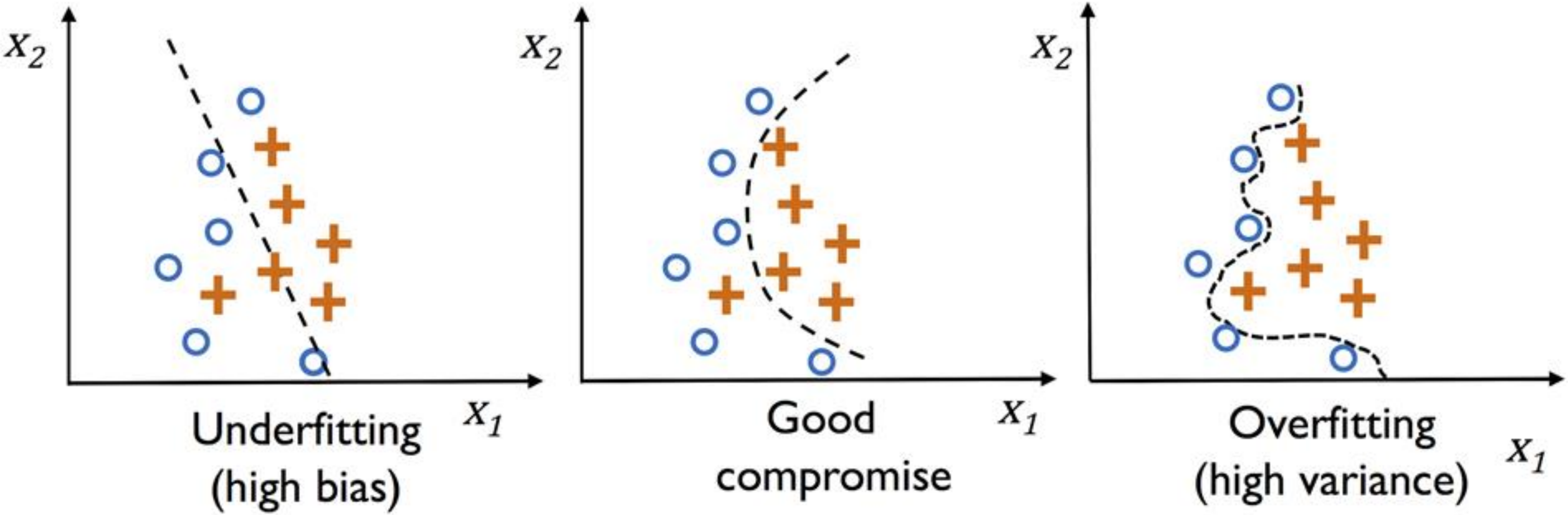
```
lr.predict(X_test_std[:3, :])
```

```
array([2, 0, 0])
```


1
$w_0$
$x_1$
$w_1$
$x_2$
$w_2$
$x_m$
$w_m$
Error
Net input function
Sigmoid activation function
Threshold function
Predicted class label
Logistic Regression
Conditional probability that a sample belongs to class 1 given its input vector x

# Overfitting and Underfitting

- A model may perform well on training data but does not generalize well to unseen data (test data)
- Two main categories of things that can go wrong
    - Overfitting (high variance)
        - Could be caused by having too many parameters that lead to a model that is too complex
    - Underfitting (high bias)
        - Model is not complex enough to capture the pattern in the training data well


$x_2$
$x_1$
Underfitting
(high bias)
$x_2$
$x_1$
Good
compromise
$x_2$
$x_1$
Overfitting
(high variance)

# Regularization

- One way of finding a good bias-variance tradeoff is to tune the complexity of the model via regularization
  - Useful method to handle collinearity (high correlation among features), filter out noise from data, and eventually prevent overfitting
- Requires feature scaling such as standardization
  - Need to ensure that all our features are on comparable scales

# L2 Regularization

- The concept behind regularization is to introduce additional information (bias) to penalize extreme parameter (weight) values
- A common form of regularization is so-called L2 regularization (sometimes also called L2 shrinkage or weight decay), which can be written as follows

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

- λ is the so-called regularization parameter

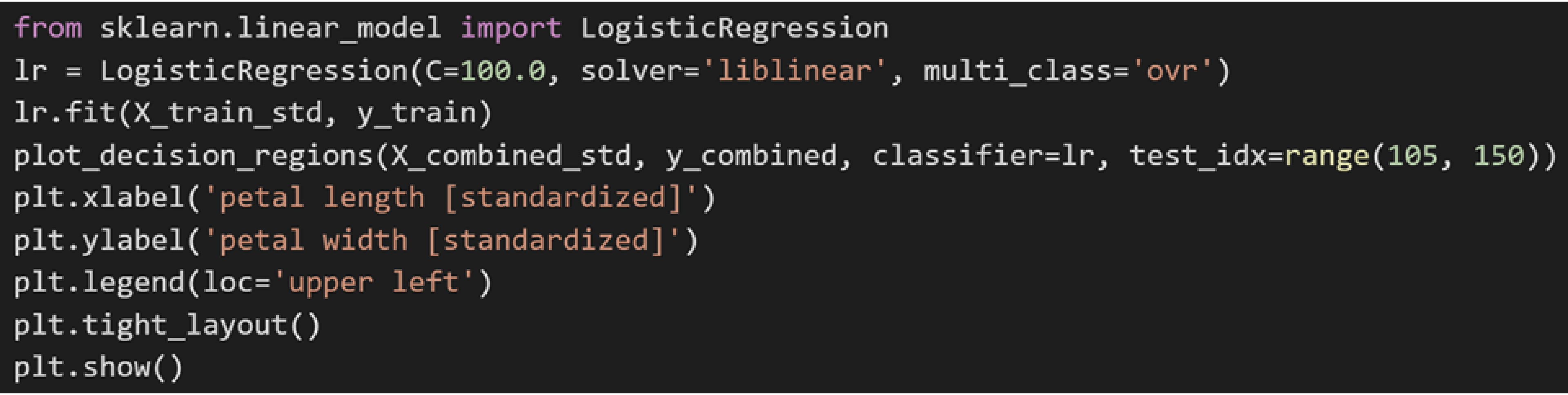
# Regularization - Updated Cost Function

- The cost function for logistic regression can be regularized by adding a simple regularization term, which will shrink the weights during model training

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

- Via the regularization parameter λ , we can then control how well we fit the training data while keeping the weights small
  - By increasing the value of λ , we increase the regularization strength

# C

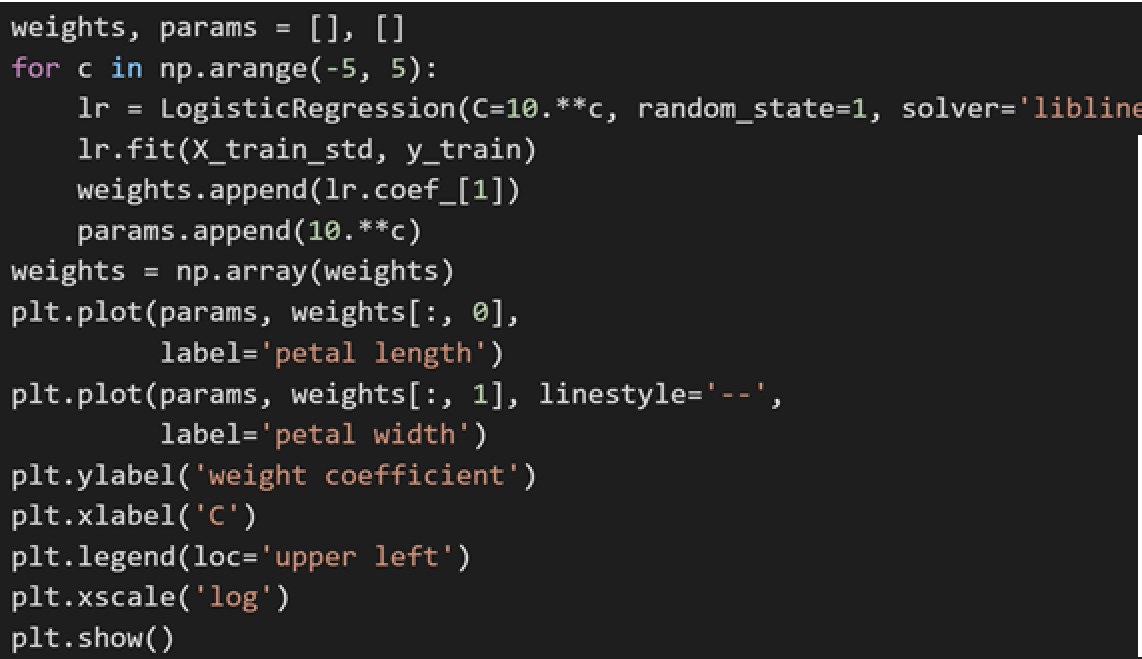


```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=100.0, solver='liblinear', multi_class='ovr')
lr.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std, y_combined, classifier=lr, test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal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

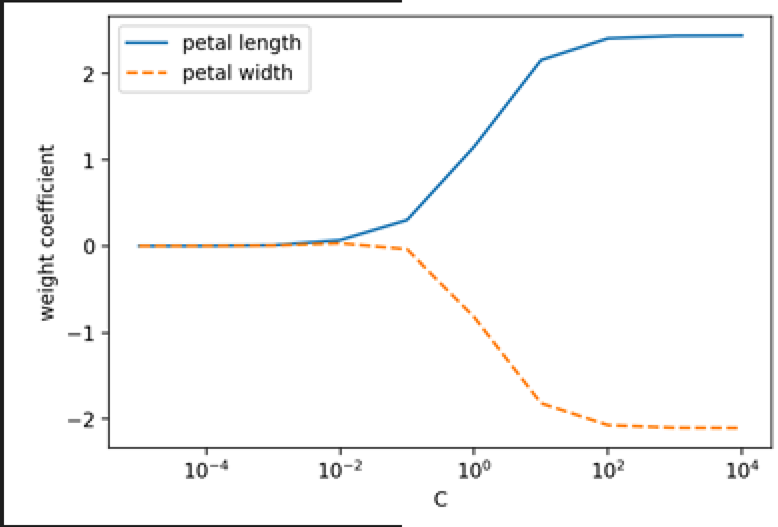
- The parameter C that is implemented for the LogisticRegression class in scikit-learn is directly related to the regularization parameter λ, which is its inverse
  - Decreasing the value of the inverse regularization parameter C means that we are increasing the regularization strength

# Regularization

- Let’s plot the L2-regularization path for the two weight coefficients
- We fit ten logistic regression models with different values for C
- For the purposes of illustration, consider only class 1



```
weights, params = [], []
for c in np.arange(-5, 5):
    lr = LogisticRegression(C=10.**c, random_state=1, solver='liblinear', multi_class='ovr')
    lr.fit(X_train_std, y_train)
    weights.append(lr.coef_[1])
    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()
```



# Outline

- Introduction to more robust algorithms for classification
  - Logistic Regression
  - Support Vector Machines
  - Decision Trees
  - KNN
- Examples using the scikit-learn machine learning library
- Strengths and weaknesses of classifiers

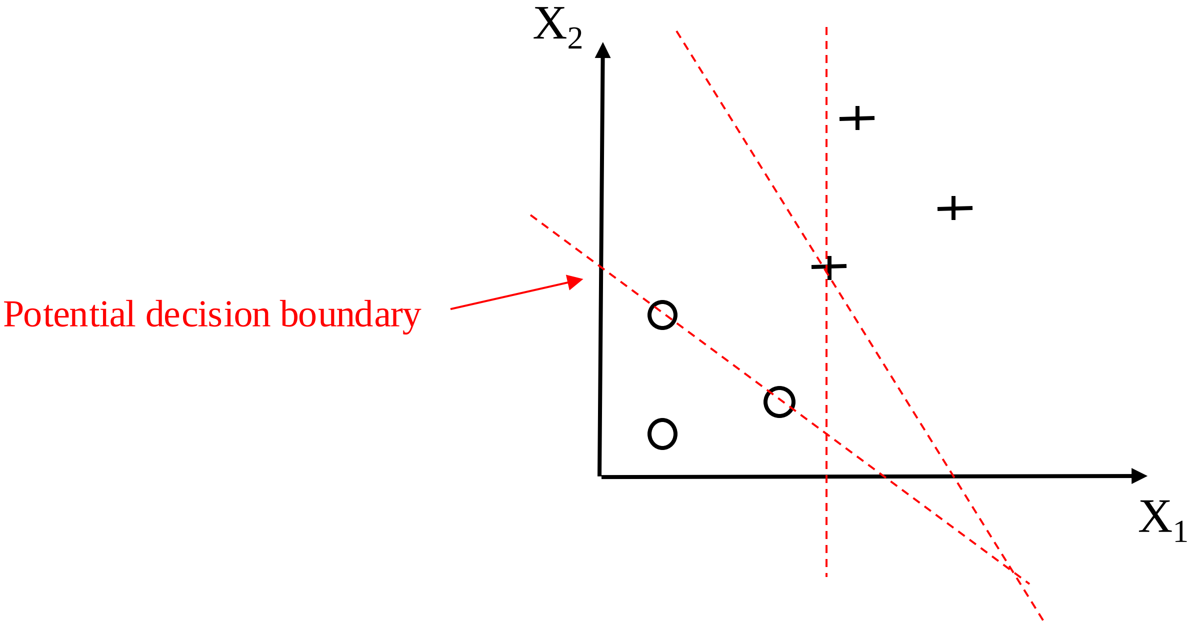
# Support Vector Machine (SVM)

- Powerful and widely used learning algorithm
    - Can be considered an extension of the perceptron
        - Perceptron: Minimized misclassification errors
        - SVM: Maximize margin
            - The margin is the distance between the separating hyperplane (decision boundary) and the training samples that are closest to this hyperplane, which are the so-called support vectors
- Original idea based on paper from 1963 by Vladimir Vapnik
    - Extended by Vladimir Vapnik in 1992 and 1995
- SVMs are used to solve various real-world problems
    - Text categorization, classification of images, handwritten character recognition, Protein classification, ...

# Which Hyperplane ?

+ Sample from positive class

○ Sample from negative class

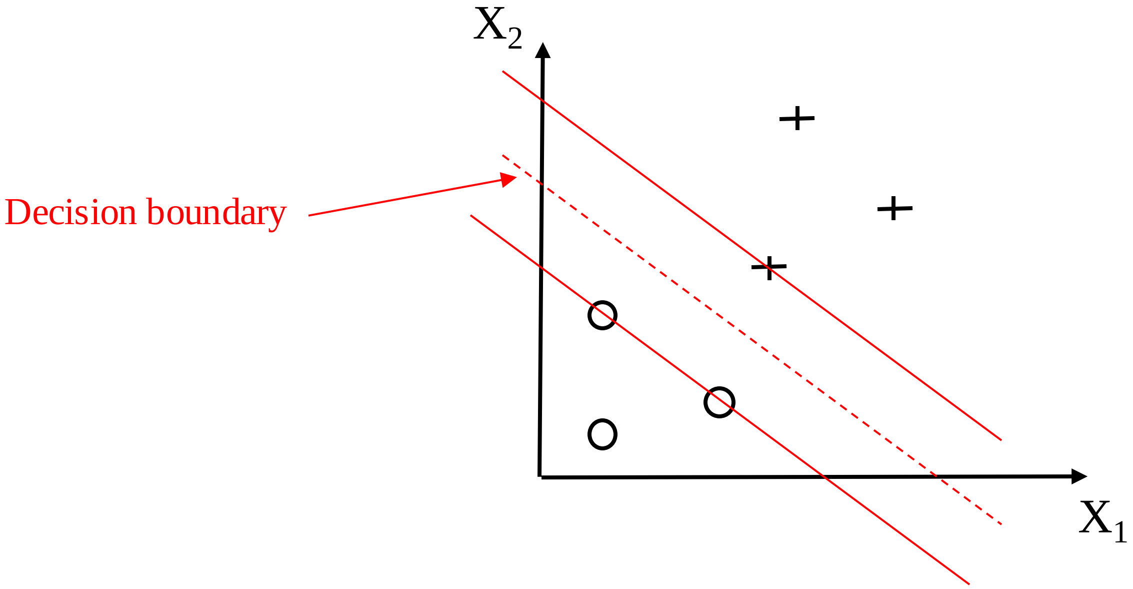


Potential decision boundary

# SVM: Maximize Margin

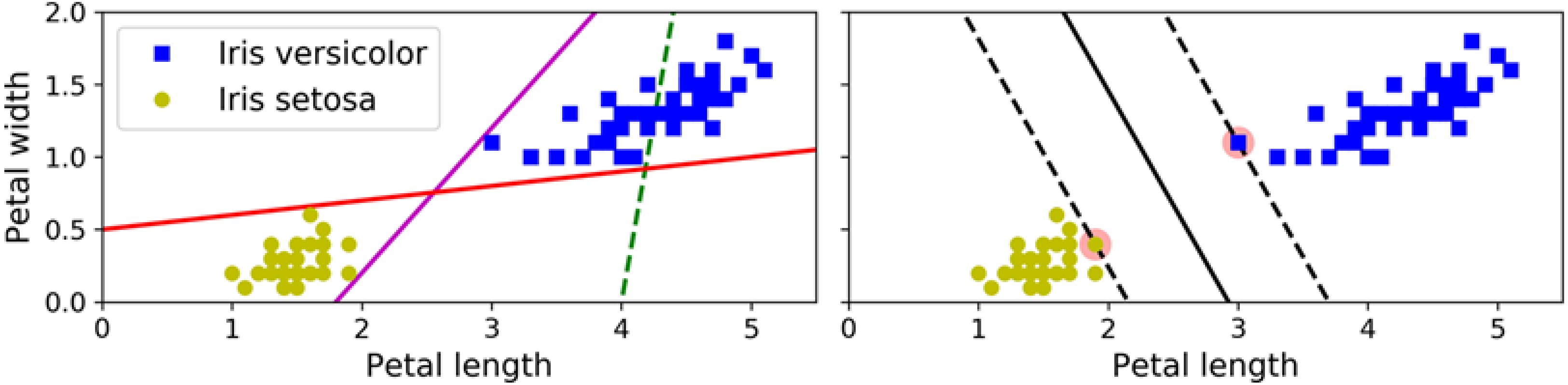
+ Sample from positive class

○ Sample from negative class



# Large Margin Classification

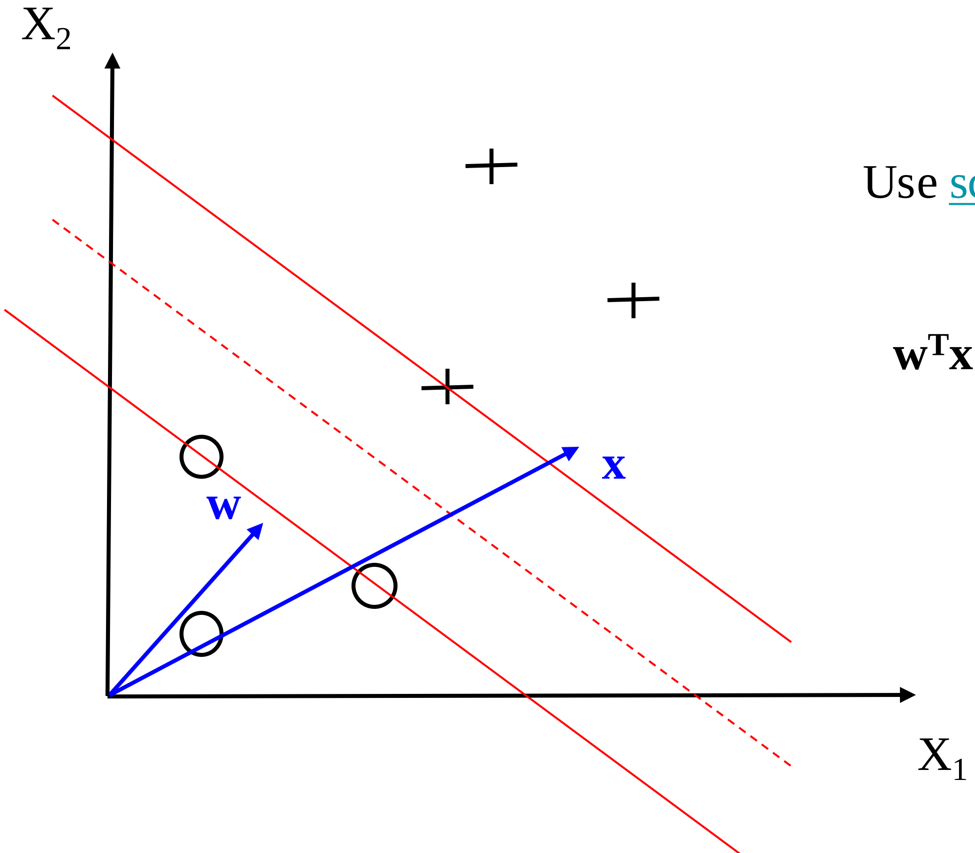
- Fitting the widest possible street is called large margin classification
- Notice that adding more training instances “off the street” will not affect the decision boundary at all
  - It is fully determined (or supported) by samples located on the edge of the street
  - These instances are called the support vectors



# SVM: Decision Rule

+ Sample from positive class

○ Sample from negative class

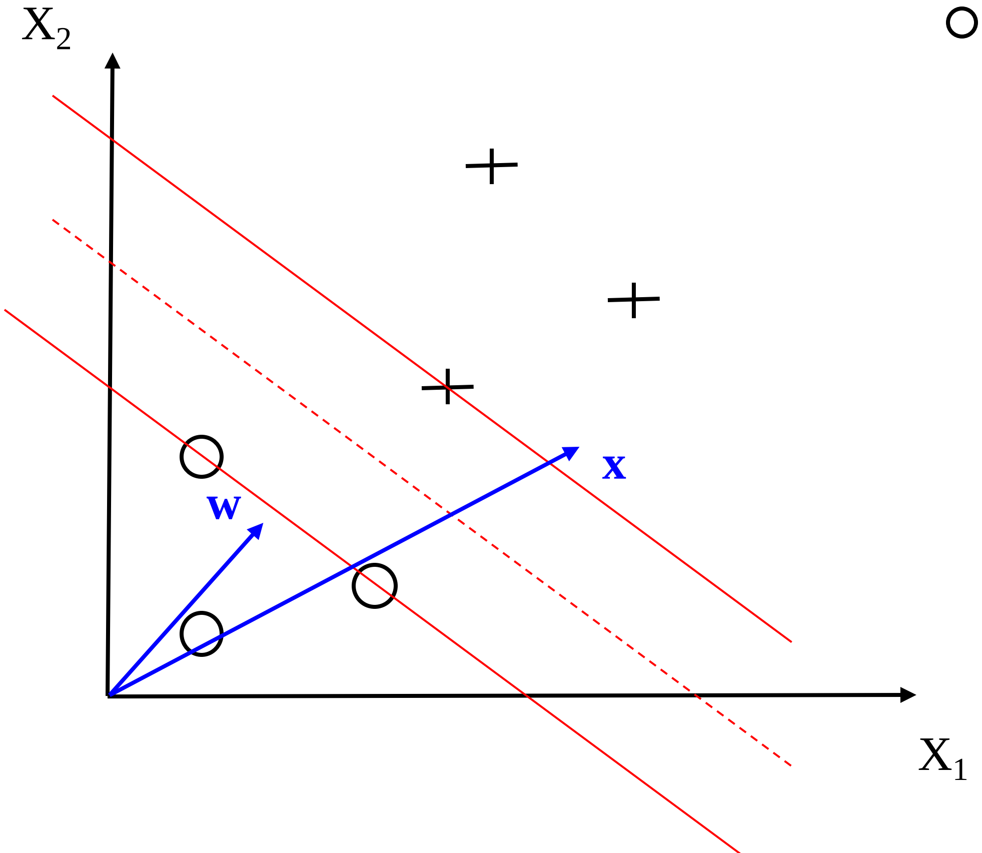


Use scalar projection: $\mathbf{w} \cdot \mathbf{x} \geq c$

Decision Rule:

$\mathbf{w}^T\mathbf{x} + w_0 \geq 0$ then positive class

# SVM: More Constraints



+ Sample from positive class $\mathbf{x}_{\mathbf{pos}}$

○ Sample from negative class $\mathbf{x}_{\mathbf{neg}}$

$$\mathbf{w}^{\mathbf{T}}\mathbf{x}_{\mathbf{pos}} + w_0 \geq 1$$
$$\mathbf{w}^{\mathbf{T}}\mathbf{x}_{\mathbf{neg}} + w_0 \leq \mathbf{-}1$$

$$\Leftrightarrow$$

$$y^{(i)}\,(\mathbf{w}^{\mathbf{T}}\mathbf{x}_{\mathbf{pos}} + w_0) \geq 1$$
$$y^{(i)}\,(\mathbf{w}^{\mathbf{T}}\mathbf{x}_{\mathbf{neg}} + w_0) \geq 1$$

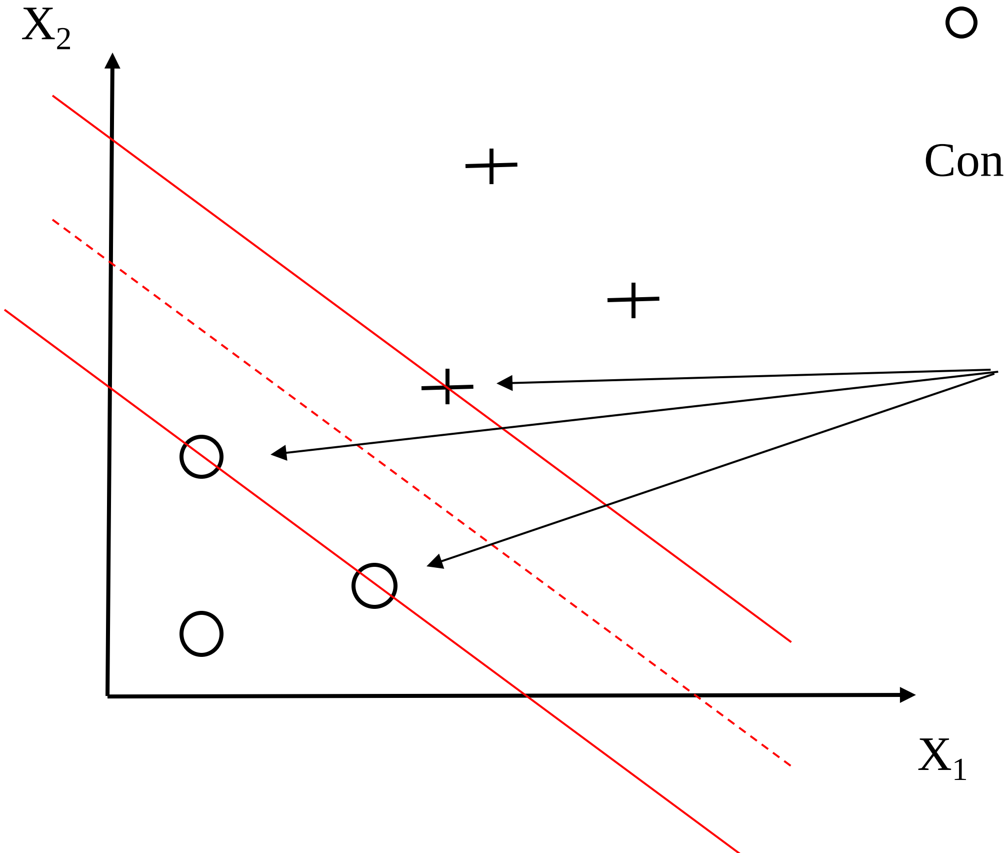
$$\Leftrightarrow$$

$$y^{(i)}\,(\mathbf{w}^{\mathbf{T}}\mathbf{x} + w_0)\ \mathbf{-}1 \geq 0$$

# SVM: More Constraints

+ Sample from positive class $\mathbf{x}_{\mathbf{pos}}$

○ Sample from negative class $\mathbf{x}_{\mathbf{neg}}$



Constraint from previous slide:

$$y^{(i)}\,(\mathbf{w}^{\mathbf{T}}\mathbf{x} + w_0) - 1 \geq 0$$

Additional constraint:

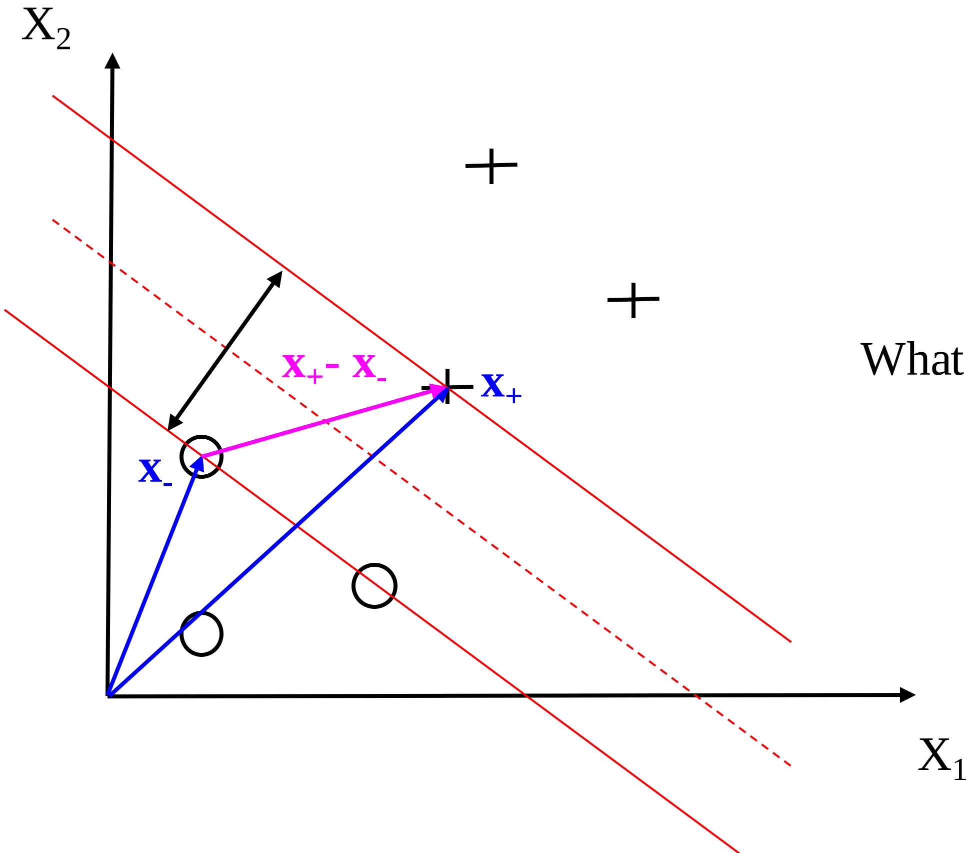
$$y^{(i)}\,(\mathbf{w}^{\mathbf{T}}\mathbf{x} + w_0) - 1 = 0$$

at the gutter

i.e.

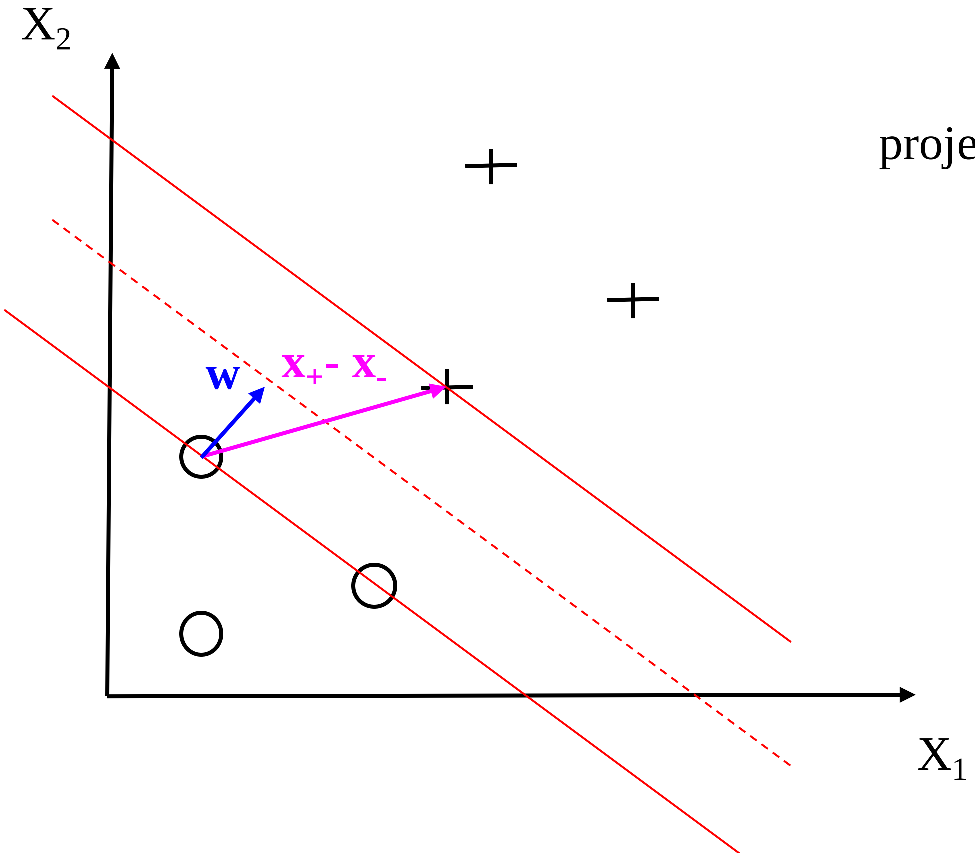
$$y^{(i)}\,(\mathbf{w}^{\mathbf{T}}\mathbf{x}_+ + w_0) - 1 = 0$$
$$y^{(i)}\,(\mathbf{w}^{\mathbf{T}}\mathbf{x}_- + w_0) - 1 = 0$$

# SVM: Margin



What is the width of the street, i.e., the margin?

# SVM: Margin

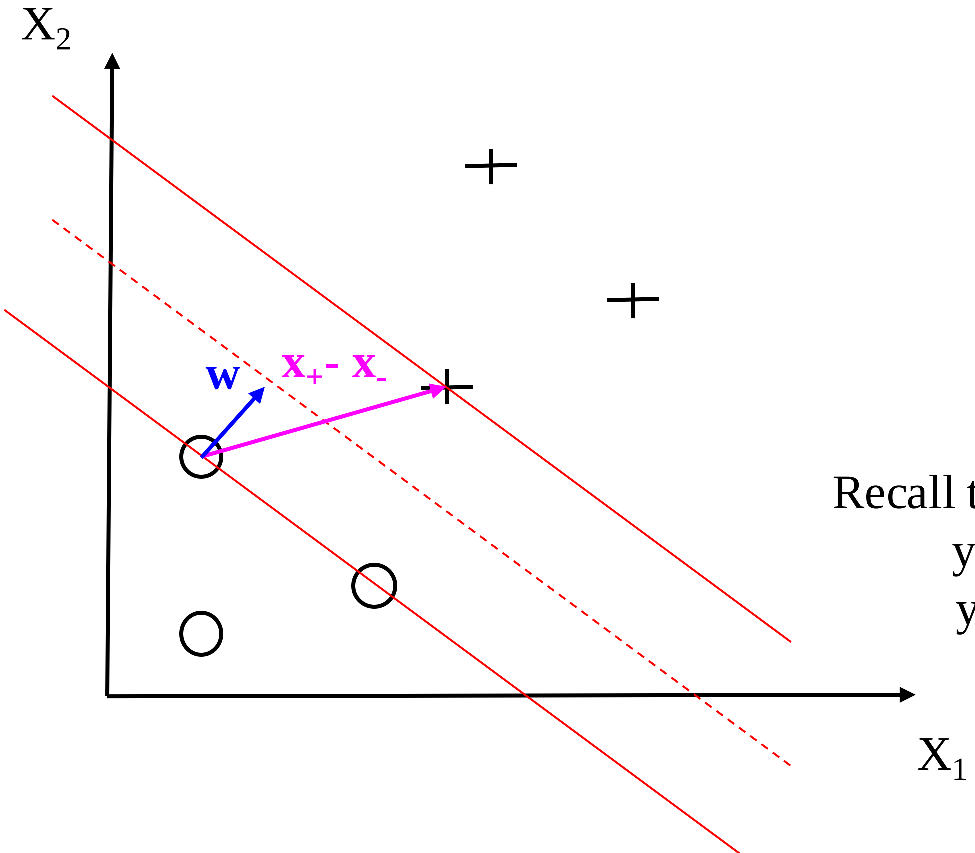


We can take the scalar projection with the unit vector **w** / ||**w**||

Width of the street:

$(\mathbf{x}_+ - \mathbf{x}_-)^T \mathbf{w} / ||\mathbf{w}||$

# SVM: Margin



Width of the street:

$(\mathbf{x}_+ - \mathbf{x}_-)^T \mathbf{w} \,/\, ||\mathbf{w}||$

$= (\mathbf{w}^T\mathbf{x}_+ - \mathbf{w}^T\mathbf{x}_-) \,/\, ||\mathbf{w}||$

$= (1 - w_0 + 1 + w_0) \,/\, ||\mathbf{w}||$

$= 2 \,/\, ||\mathbf{w}||$

Recall that we had the constraints:

$y^{(i)} (\mathbf{w}^T\mathbf{x}_+ + w_0) - 1 = 0$

$y^{(i)} (\mathbf{w}^T\mathbf{x}_- + w_0) - 1 = 0$

i.e.,

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

$\mathbf{w}^T\mathbf{x}_- = -1 - w_0$

# SVM: Optimization

- The objective function of the SVM is the maximization of the margin

$$2 / ||\mathbf{w}||$$

- Equivalently we can minimize the reciprocal term

$$||\mathbf{w}||$$

# SVM: Optimization

- For mathematical convenience we solve the following

$$\underset{\boldsymbol{w}}{\arg\min} \quad \frac{1}{2}\|\boldsymbol{w}\|^2$$

subject to the constraints

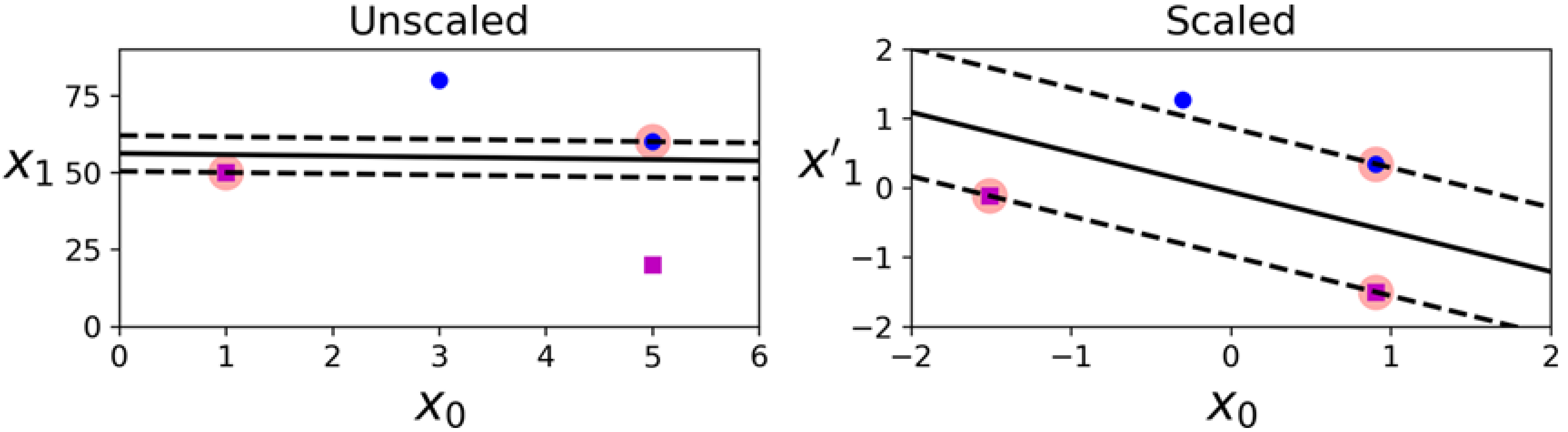
$$y^{(i)}\left(w_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)}\right) \geq 1 \ \forall_i$$

- This can be minimized efficiently by quadratic programming
  - More details can be found here
    - The Nature of Statistical Learning Theory

  Vladimir Vapnik, 2000
    - A Tutorial on Support Vector Machines for Pattern Recognition
      Data Mining and Knowledge Discovery, 1998
  - It can be shown that the search space is convex
    - The optimization will not get stuck in a local minima

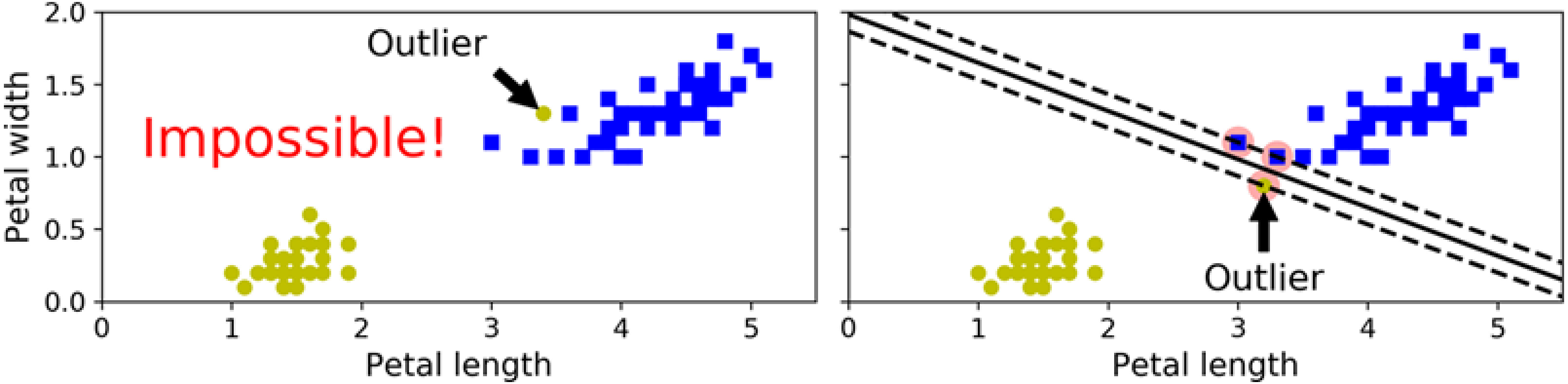
# Scaling

- SVMs are sensitive to the feature scales



# Hard vs. Soft

- If we strictly impose that all instances must be off the street and on the right side, this is called hard margin classification
- There are two main issues with hard margin classification
    - First, it only works if the data is linearly separable
    - Second, it is sensitive to outliers



# Hard vs. Soft

- To avoid these issues, use a more flexible model
- The objective is to find a good balance between keeping the street as large as possible and limiting the margin violations
    - I.e., instances that end up in the middle of the street or even on the wrong side
- This is called soft margin classification.

# Slack Variables

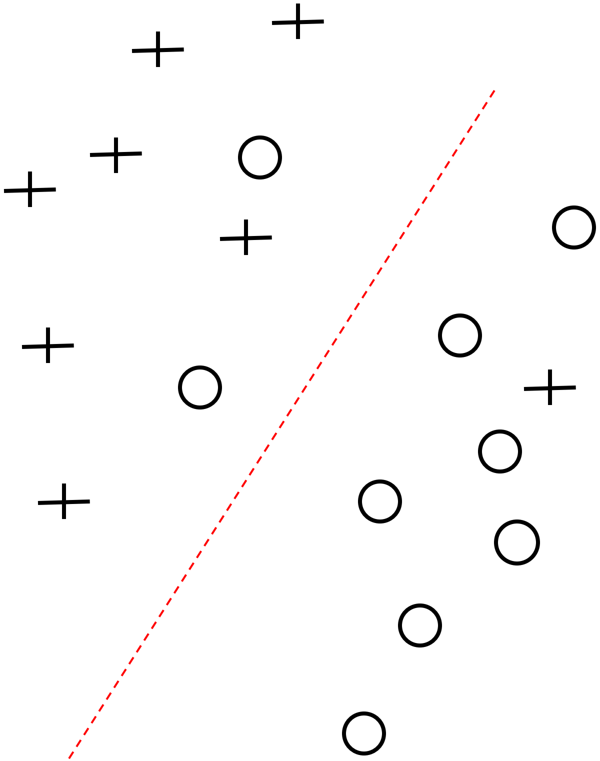
- A slack variable $\xi$ can be introduced
  - Positive (or zero)
- Allows for convergence in the presence of misclassifications
  - For nonlinearly separable data
  - Soft margin

# If Data is not Linearly Separable Introduce Penalty

How to penalize?

Idea: $\underset{\boldsymbol{w}}{\arg\min}\ \frac{1}{2}\,||\boldsymbol{w}||^2 + C\ (\#\ \text{mistakes})$

Not all mistakes are equally bad.
Use margin to penalize the mistakes.



# If Data is not Linearly Separable Introduce Penalty

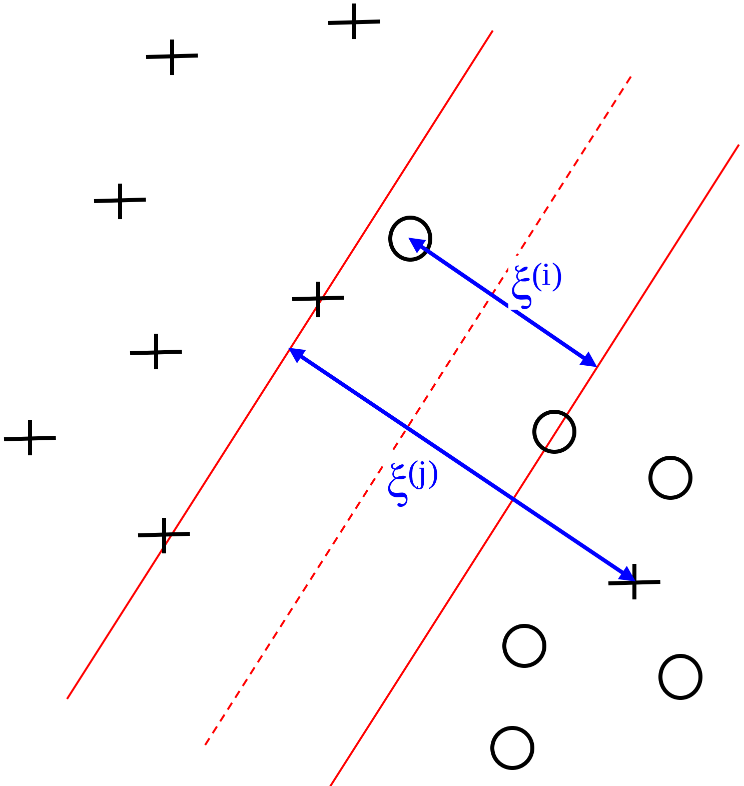
Introduce slack variable $\xi^{(i)}$

If point $x^{(i)}$ is on the wrong side
then get penalty $\xi^{(i)}$

Idea v2: $$\underset{\boldsymbol{w},\ \xi^{(i)}}{\arg\min}\ \frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

Under the following constraints

$$y^{(i)}\,(\mathbf{w}^T\mathbf{x}^{(i)} + w_0) \geq 1 - \xi^{(i)}$$



# Slack Penalty C

- New optimization problem:

$$\underset{\boldsymbol{w},\ \xi^{(i)}}{\arg\min} \quad \frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

- Via the variable C, we can control the penalty for misclassification
  - Large values of C correspond to large error penalties, whereas we are less strict about misclassification errors if we choose smaller values for C
  - This concept is related to regularization
    - Decreasing the value of C increases the bias and lowers the variance of the model

# Hinge Loss Function

- The preceding formulation of the SVM cost function was in the so called QP form
- An equivalent formulation uses the hinge loss function

$$\max(\ 0,\ 1 - y^{(i)}\ (\ \boldsymbol{w}^{T}\mathbf{x}^{(i)} + w_0\ )\ )$$

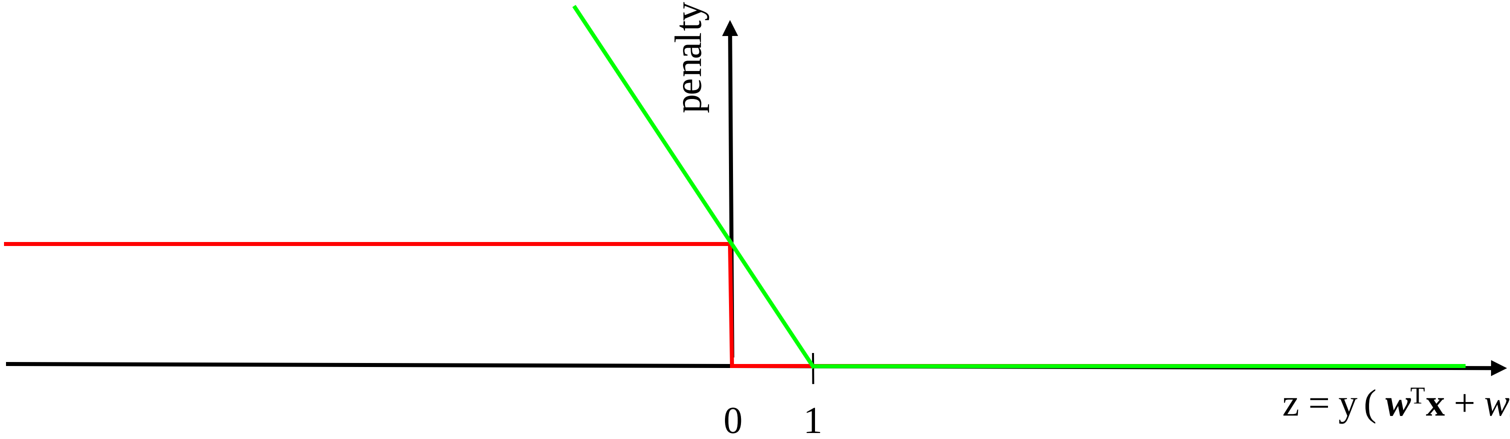
- The hinge loss function returns 0 if the point is correctly classified
- If the point is on the wrong side of the margin, the function's value is proportional to the distance from the margin

# SVM 'Natural' Form

- SVM in the 'natural' form:

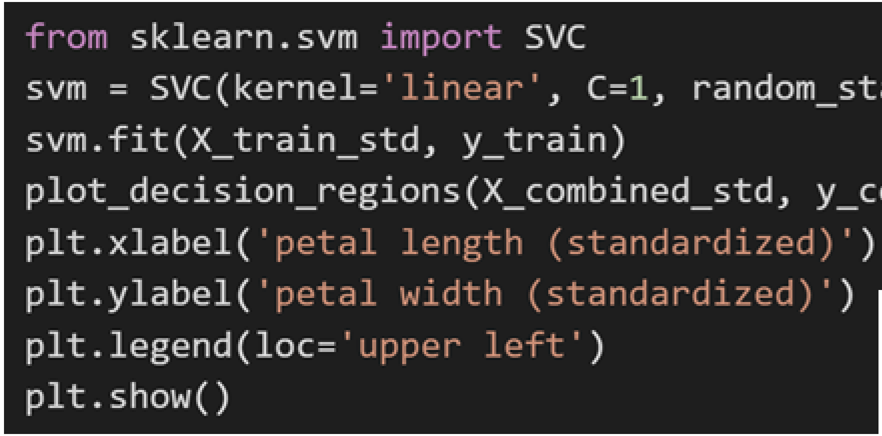
$$\underset{\boldsymbol{w}, w_0}{\arg\min} \quad \frac{1}{2}||\mathbf{w}||^2 + C \sum_{i=1}^{n} \max(0, 1 - y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + w_0))$$

- Hinge Loss vs. 0\1 Loss

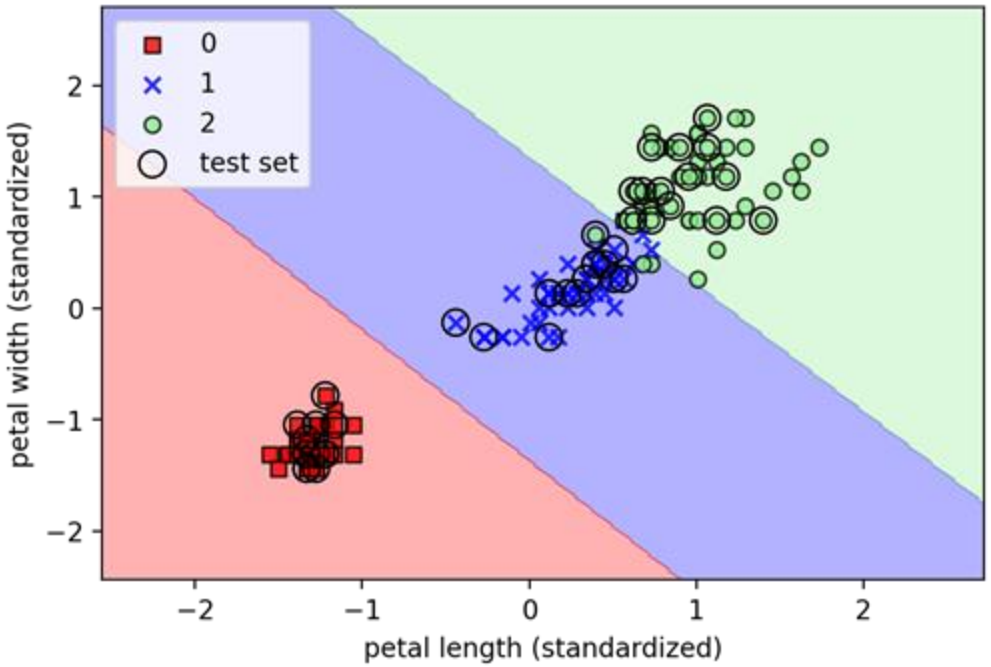


# Code - SVM.ipynb

- Available here on CoLab



```
from sklearn.svm import SVC
svm = SVC(kernel='linear', C=1, random_state=1)
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.show()
```

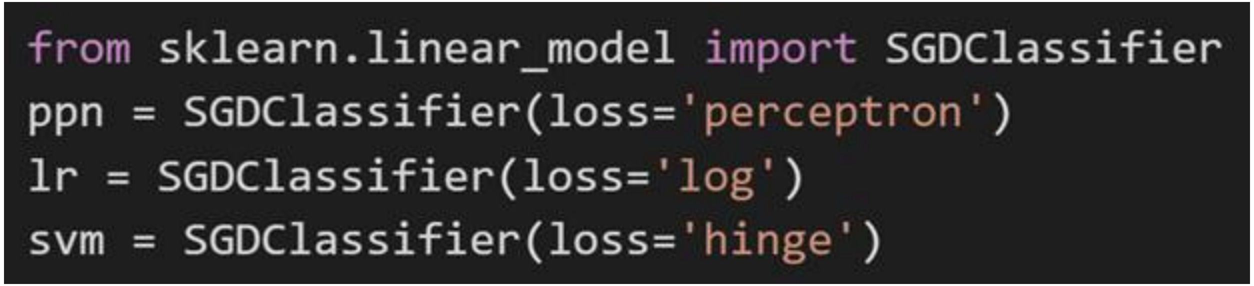


# Logistic Regression vs. SVM

- Linear logistic regression and linear SVMs often yield similar results
- Logistic regression tries to maximize the conditional likelihoods of the training data, which makes it more prone to outliers than SVMs, which mostly care about the points that are closest to the decision boundary (support vectors)
- On the other hand, logistic regression has the advantage that it is a simpler model and can be implemented more easily
- Furthermore, logistic regression models can be easily updated, which is attractive when working with streaming data

# Alternative Scikit-learn Implementations

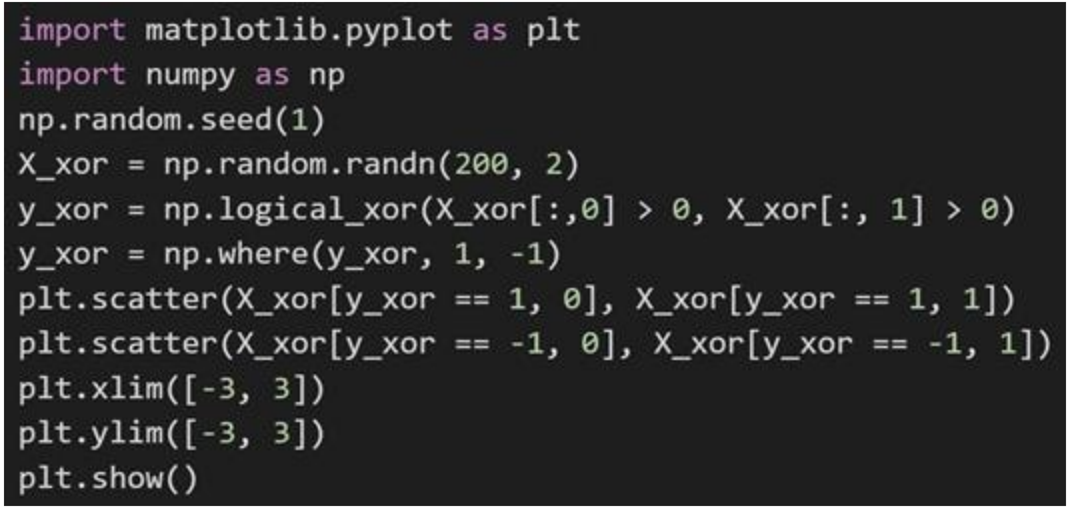
- We used the LIBLINEAR library (via scikit-learn library's Perceptron and LogisticRegression classes) in the previous section
  - A highly optimized C/C++ library developed at NTU
- Similarly, the SVC makes use of LIBSVM, which is an equivalent C/C++ library specialized for SVMs
- The above libraries are very fast, however, if your dataset cannot fit into memory you may use an alternative implementation available in SGDClassifier
  - Supports online learning via the partial_fit method



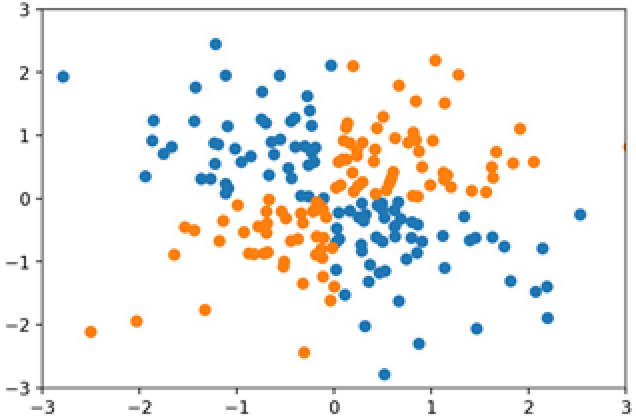
```
from sklearn.linear_model import SGDClassifier
ppn = SGDClassifier(loss='perceptron')
lr = SGDClassifier(loss='log')
svm = SGDClassifier(loss='hinge')
```

# Kernel SVM

- SVM can be kernelized to solve nonlinear classification problems
- Let’s create a small data set of linearly inseparable data

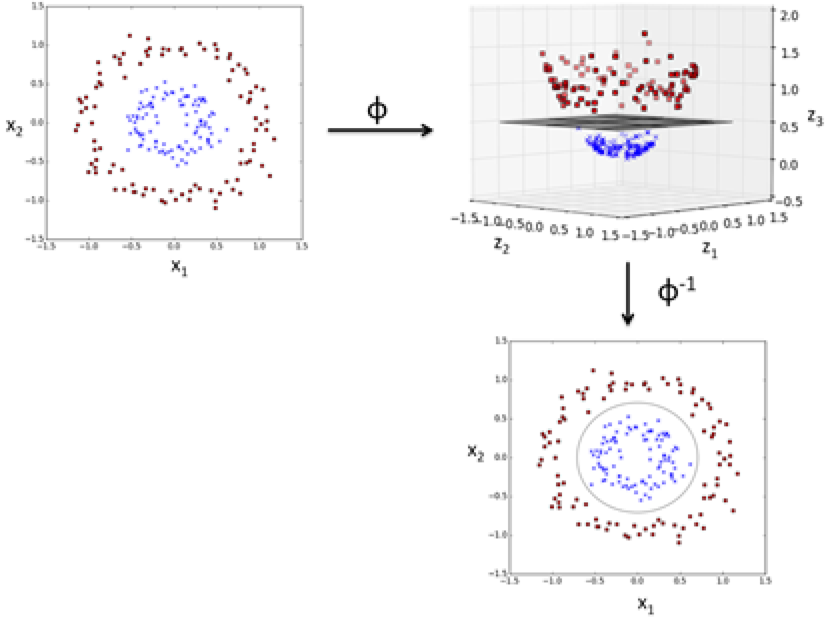


```
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(1)
X_xor = np.random.randn(200, 2)
y_xor = np.logical_xor(X_xor[:,0] > 0, X_xor[:, 1] > 0)
y_xor = np.where(y_xor, 1, -1)
plt.scatter(X_xor[y_xor == 1, 0], X_xor[y_xor == 1, 1])
plt.scatter(X_xor[y_xor == -1, 0], X_xor[y_xor == -1, 1])
plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.show()
```



# Kernel SVM - Mapping Function $\phi$

- Basic idea
  - Deal with linearly inseparable data by creating nonlinear combinations of the original features by projecting them onto a higher-dimensional space where it becomes linearly separable
- Example
  - We can transform a 2D dataset onto a new 3D feature space where the classes become separable via the following projection



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

# Kernel SVM

- One problem with this mapping approach is that the construction of the new features is computationally very expensive, especially if we are dealing with high-dimensional data
- This is where the so-called kernel trick comes into play
  - In practice all we need is to replace the dot product

$$\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)} \text{ by } \phi\left(\boldsymbol{x}^{(i)}\right)^T \phi\left(\boldsymbol{x}^{(j)}\right)$$

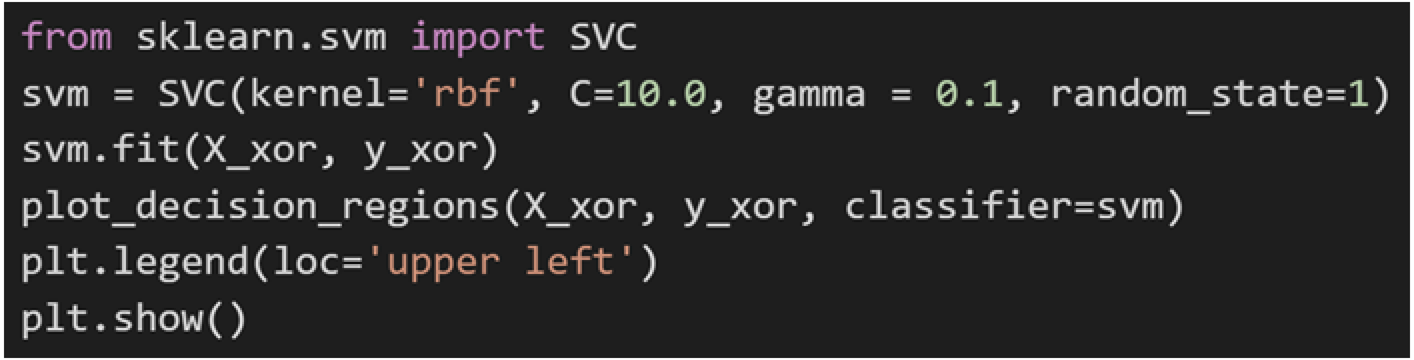
  - In order to save the expensive step of calculating the dot product between two points explicitly, we define a so-called kernel function

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

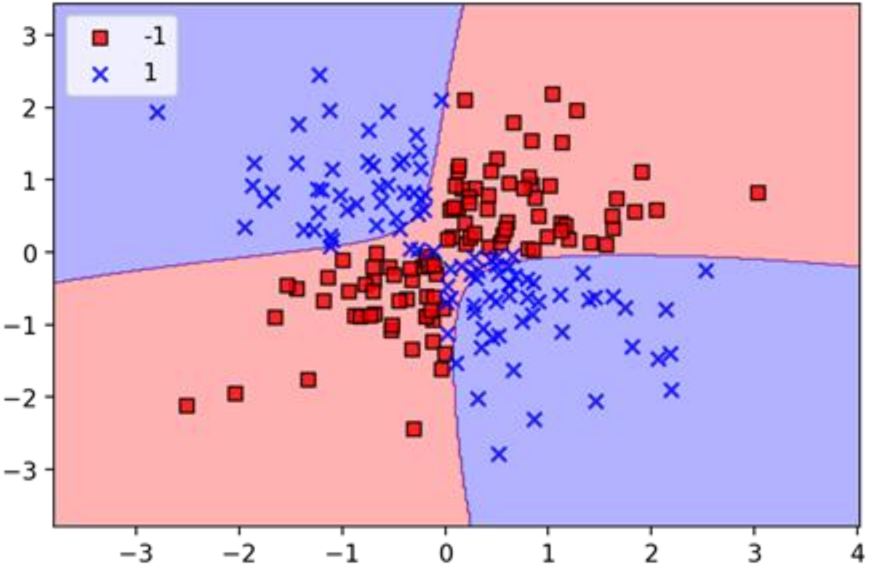
  - On of the most widely used kernels is the Gaussian kernel (aka radial basis function)

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

  - where $\gamma = \frac{1}{2\sigma^2}$ is a free parameter

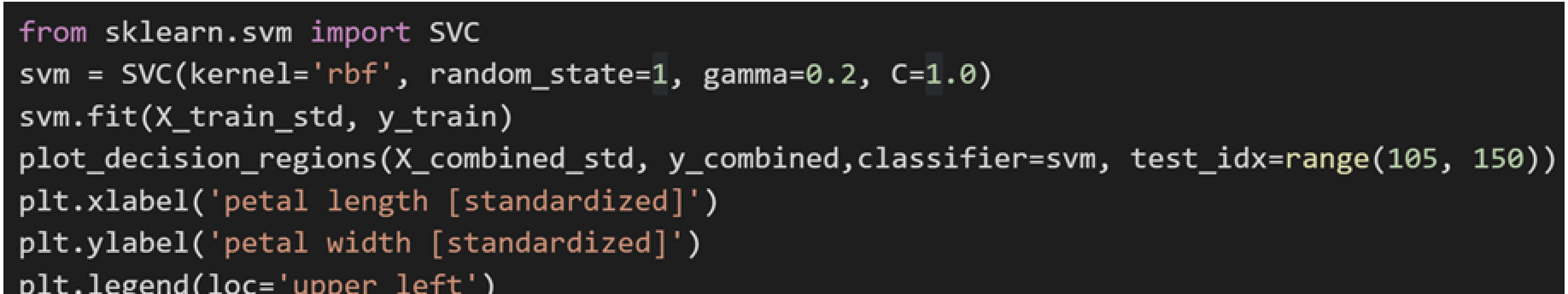


```python
from sklearn.svm import SVC
svm = SVC(kernel='rbf', C=10.0, gamma = 0.1, random_state=1)
svm.fit(X_xor, y_xor)
plot_decision_regions(X_xor, y_xor, classifier=svm)
plt.legend(loc='upper left')
plt.show()
```

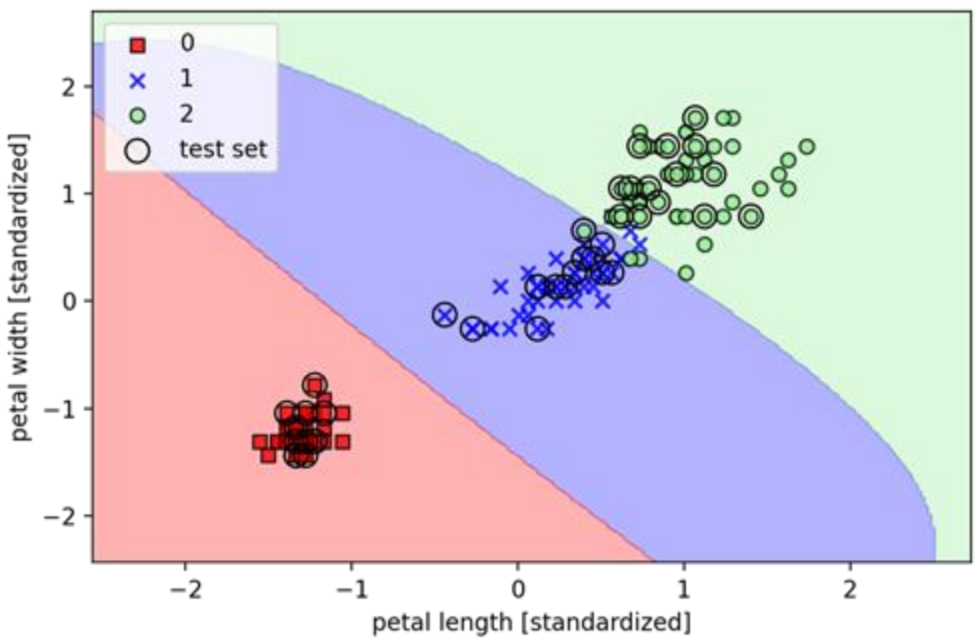


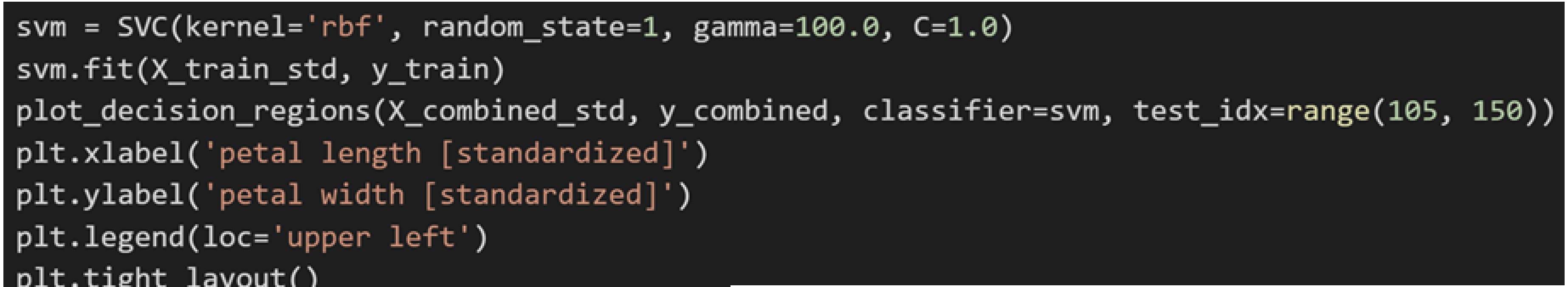
# Gamma

- The gamma parameter can be understood as a cut-off parameter for the Gaussian
- If we increase the value for gamma, we increase the influence or reach of the training samples, which leads to a tighter and bumpier decision boundary
- To get a better intuition for gamma, let us apply an RBF kernel SVM to our Iris flower dataset

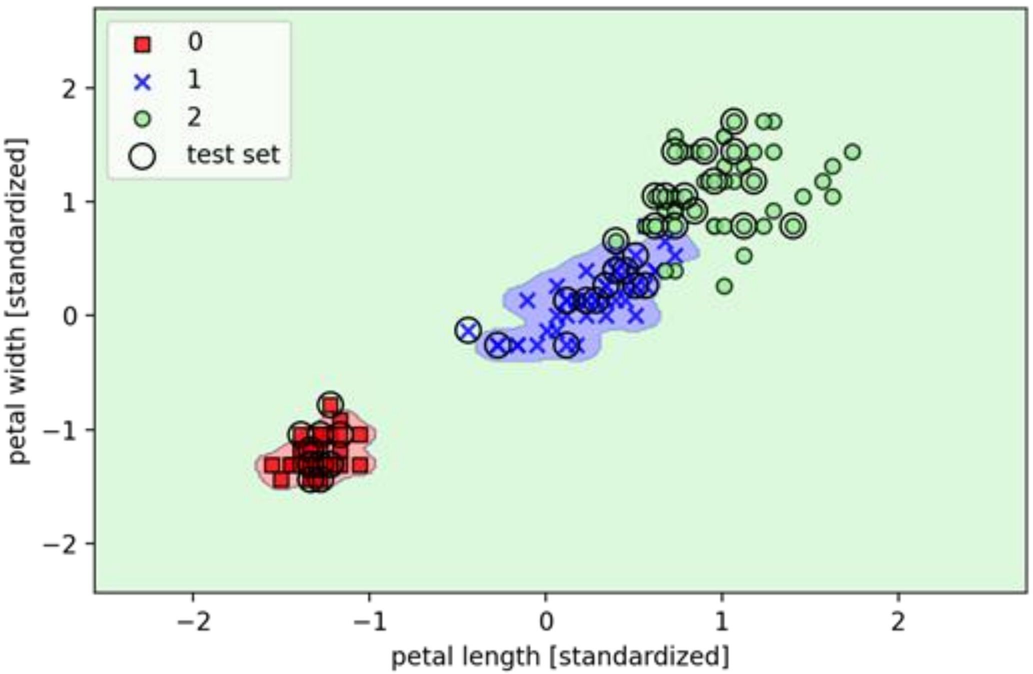


```
from sklearn.svm import SVC
svm = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
svm.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std, y_combined,classifier=svm, test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```





```
svm = SVC(kernel='rbf', random_state=1, gamma=100.0, C=1.0)
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()
```

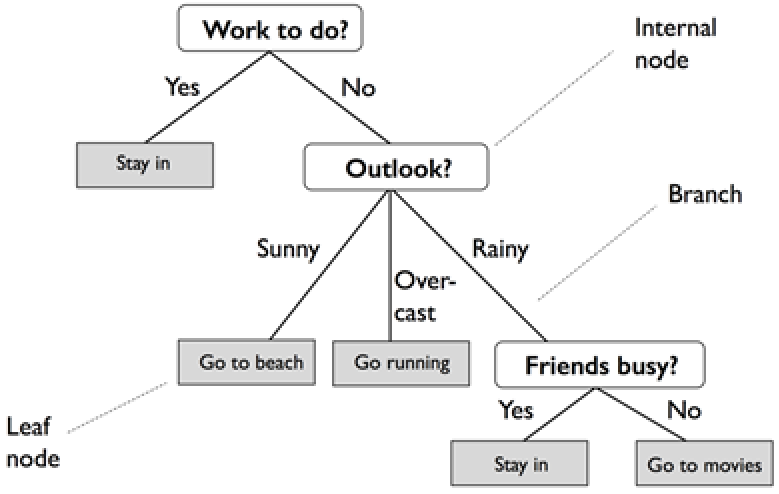


# Outline

- Introduction to more robust algorithms for classification
    - Logistic Regression
    - Support Vector Machines
    - Decision Trees
    - KNN
- Examples using the scikit-learn machine learning library
- Strengths and weaknesses of classifiers

# Decision Tree Learning

- Decision tree classifiers are attractive models if we care about interpretability
- This model breaks down our data by making a decision based on asking a series of questions
- Example of decision tree



# Decision Tree Learning

- Model learns to ask a series of questions
    - E.g.: Is sepal width ≥ 2.8 cm?
- Prediction is based on answers to questions
- What questions to ask?
    - Split data on feature that results in largest Information Gain (IG)
    - Iterative repeat splitting until leaves are pure
        - I.e., samples all belong to same class
        - This can result in a very deep tree with many nodes, which can easily lead to overfitting
        - Thus, we typically want to prune the tree by setting a limit for the maximal depth of the tree

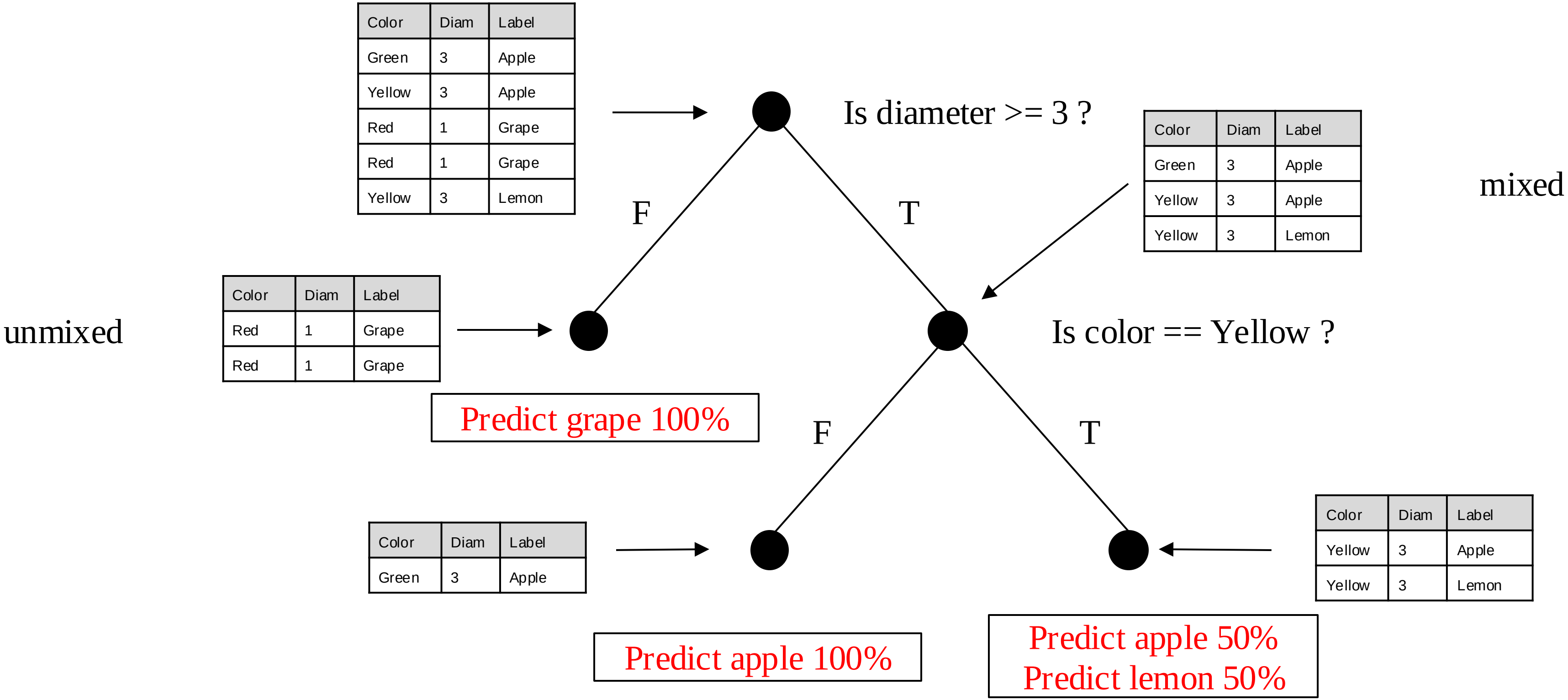
# Decision Tree Learning - Intuition

- Training data:

| Color | Diameter | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |
| Yellow | 3 | Lemon |

features

# Decision Tree Learning - Intuition



| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |
| Yellow | 3 | Lemon |

Is diameter >= 3 ?

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Yellow | 3 | Lemon |

mixed

unmixed

| Color | Diam | Label |
|---|---|---|
| Red | 1 | Grape |
| Red | 1 | Grape |

Is color == Yellow ?

Predict grape 100%

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |

| Color | Diam | Label |
|---|---|---|
| Yellow | 3 | Apple |
| Yellow | 3 | Lemon |

Predict apple 100%

Predict apple 50%
Predict lemon 50%

# Which questions to ask and when?

- Quantify how much a question helps to unmix the labels
  - 1. Quantify the amount of uncertainty at a single node
    - E.g. Gini Impurity

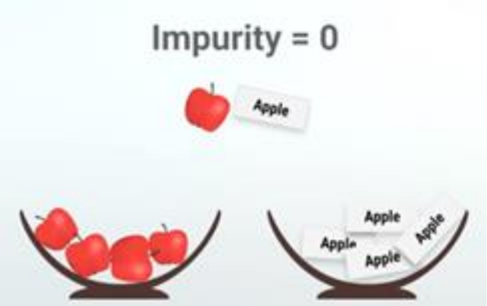
$$I_G(t)$$

  - 2. How much does a question reduce the uncertainty
    - We aim to maximize Information Gain

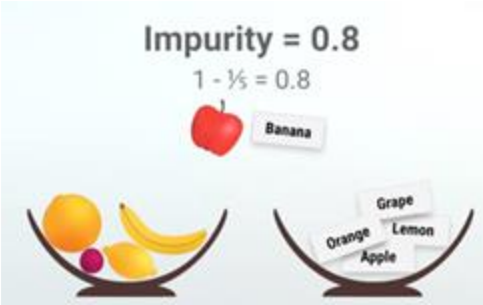
$$IG(D_p, f)$$

# 1. Gini Impurity

- Chance of being incorrect if you randomly assign a label to an example in the same set



Low Impurity
High Impurity



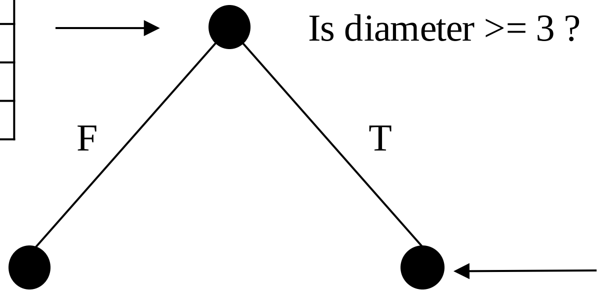
$$I_G(t) = \sum_{i=1}^{c} p(i|t)\left(1 - p(i|t)\right) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

p( i | t ) is the proportion of the samples that belong to class i for a particular node t

# 1. Gini Impurity

Gini Impurity:
$I_G = 0.64$

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |
| Yellow | 3 | Lemon |



Gini Impurity:
$I_G = 0$

| Color | Diam | Label |
|---|---|---|
| Red | 1 | Grape |
| Red | 1 | Grape |

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Yellow | 3 | Lemon |

Gini Impurity:
$I_G = 0.44$

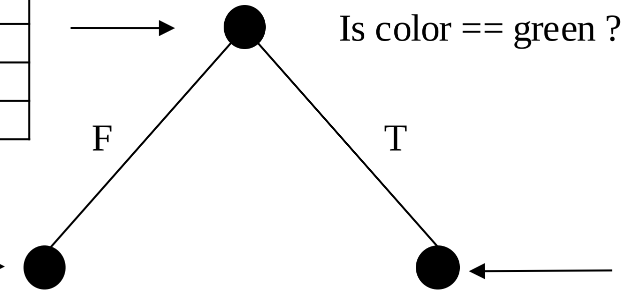
$$I_G(t) = \sum_{i=1}^{c} p(i|t)\left(1 - p(i|t)\right) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

p( i | t ) is the proportion of the samples that belong to class i for a particular node t

# 1. Gini Impurity

Gini Impurity:
$I_G = 0.64$

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |
| Yellow | 3 | Lemon |



Gini Impurity:
$I_G = 0.625$

| Color | Diam | Label |
|---|---|---|
| Yellow | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |
| Yellow | 3 | Lemon |

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |

Gini Impurity:
$I_G = 0$

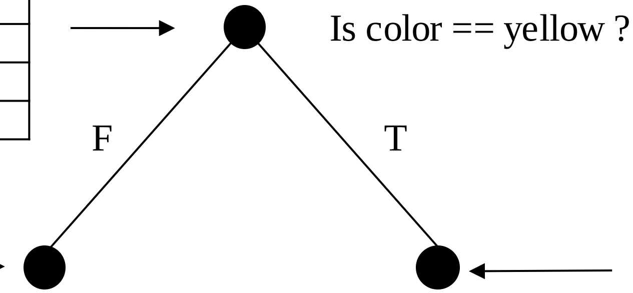
$$I_G(t)=\sum_{i=1}^{c} p(i|t)\left(1-p(i|t)\right)=1-\sum_{i=1}^{c} p(i|t)^2$$

p( i | t ) is the proportion of the samples that belong to class i for a particular node t

# 1. Gini Impurity

Gini Impurity:
$I_G = 0.64$

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |
| Yellow | 3 | Lemon |



Is color == yellow ?

Gini Impurity:
$I_G = 0.44$

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |

| Color | Diam | Label |
|---|---|---|
| Yellow | 3 | Apple |
| Yellow | 3 | Lemon |

Gini Impurity:
$I_G = 0.5$

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

p( i | t ) is the proportion of the samples that belong to class i for a particular node t

# Which questions to ask and when?

- Quantify how much a question helps to unmix the labels
  - 1. Quantify the amount of uncertainty at a single node
    - E.g. Gini Impurity

$$I_G(t)$$

  - 2. How much does a question reduce the uncertainty
    - We aim to maximize Information Gain

$$IG(D_p, f)$$

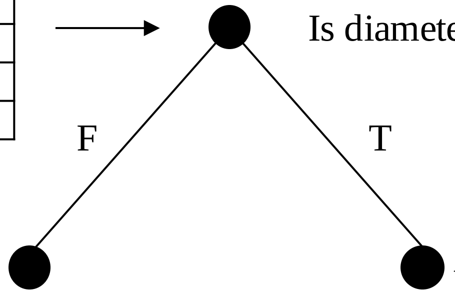
# 2. Information Gain

Gini Impurity:
$I_G = 0.64$

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |
| Yellow | 3 | Lemon |

Information Gain:
IG = 0.376

Is diameter >= 3 ?



Gini Impurity:
$I_G = 0$

| Color | Diam | Label |
|---|---|---|
| Red | 1 | Grape |
| Red | 1 | Grape |

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Yellow | 3 | Lemon |

Gini Impurity:
$I_G = 0.44$

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

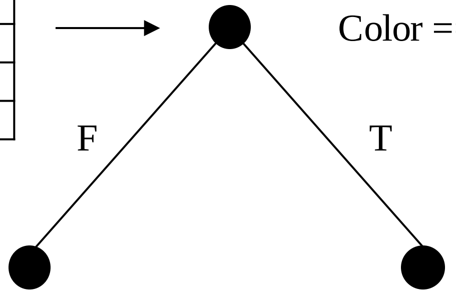
# 2. Information Gain - Task

Gini Impurity:
$I_G = 0.64$

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |
| Yellow | 3 | Lemon |

Information Gain:
IG = ?

Color == green ?



Gini Impurity:
$I_G = ?$

Gini Impurity:
$I_G = ?$

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

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

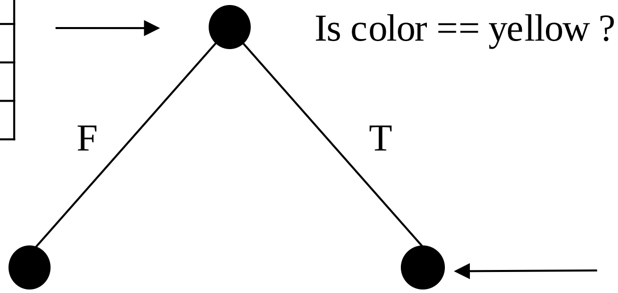
# 2. Information Gain

Gini Impurity:
$I_G = 0.64$

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |
| Yellow | 3 | Lemon |

Information Gain:
IG = 0.176

Is color == yellow ?



Gini Impurity:
$I_G = 0.44$

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |

| Color | Diam | Label |
|---|---|---|
| Yellow | 3 | Apple |
| Yellow | 3 | Lemon |

Gini Impurity:
$I_G = 0.5$

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

# Example

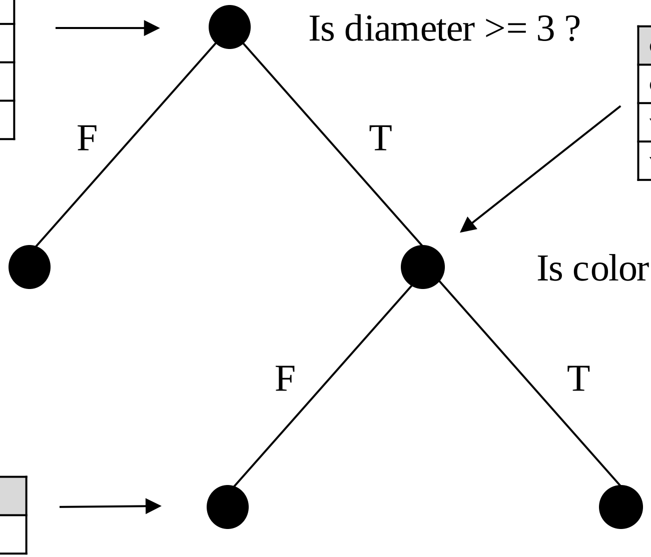
$$I_G(t)=\sum_{i=1}^{c} p(i|t)(1-p(i|t))=1-\sum_{i=1}^{c} p(i|t)^2$$

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

Gini Impurity:
$I_G = 0.64$

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |
| Yellow | 3 | Lemon |

Information Gain:
IG = 0.376



Is diameter >= 3 ?

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Yellow | 3 | Lemon |

Gini Impurity:
$I_G = 0.44$

Gini Impurity:
$I_G = 0$

| Color | Diam | Label |
|---|---|---|
| Red | 1 | Grape |
| Red | 1 | Grape |

Is color == Yellow ?

Information Gain:
IG = 0.11

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |

Gini Impurity:
$I_G= 0$

| Color | Diam | Label |
|---|---|---|
| Yellow | 3 | Apple |
| Yellow | 3 | Lemon |

Gini Impurity:
$I_G = 0.5$

# Maximizing Information Gain

- Split nodes at most informative features
  - In our tree learning algorithm we optimize an objective function
    - E.g., maximize information gain at each split

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

- f is the feature to perform the split
- $D_p$ and $D_j$ are the dataset of the parent and jth child node
- I is our impurity measure
- $N_p$ is the total number of samples at the parent node, and
- $N_j$ is the number of samples in the jth child node.

# Binary Decision Tree

- For binary decision trees each parent node is split into two child nodes
  - $D_{left}$ and $D_{right}$

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

- The following impurity measures or splitting criteria are commonly used in binary decision trees
  - Gini impurity ( $I_G$ ),
  - Entropy ( $I_H$ ), and
  - Classification error ( $I_E$ )

# Entropy

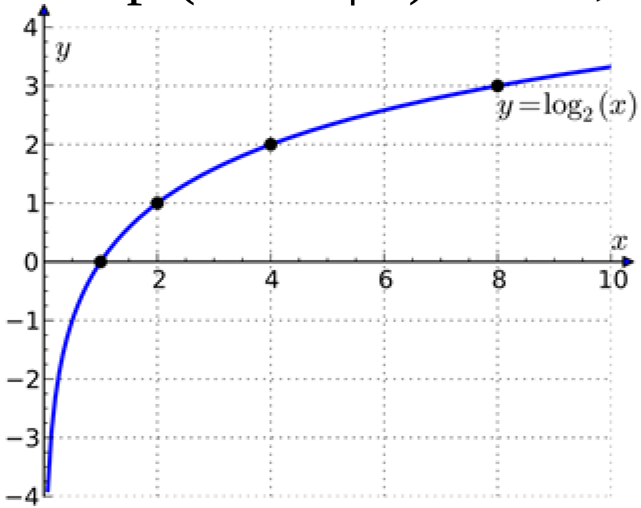
$$I_H(t) = -\sum_{i=1}^{c} p(i|t)\log_2 p(i|t)$$

- p( i | t ) is the proportion of the samples that belong to class i for a particular node t
- The entropy is therefore 0 if all samples at a node belong to the same class, and the entropy is maximal if we have a uniform class distribution
- The entropy criterion attempts to maximize the mutual information in the tree

p( i | t ) is the proportion of the samples that belong to class i for a particular node t

# Entropy

- For example, in a binary class setting ( c = 2 )
  - The entropy is 0 if p ( i = 1| t ) = 1 or p ( i = 0 | t ) = 0
  - If the classes are distributed uniformly with p ( i = 1| t ) = 0.5 and p ( i = 0 | t ) = 0.5 , the entropy is 1



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

p( i | t ) is the proportion of the samples that belong to class i for a particular node t

# Gini

- The Gini impurity can be understood as a criterion to minimize the probability of misclassification

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

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

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

p( i | t ) is the proportion of the samples that belong to class i for a particular node t

# Classification Error

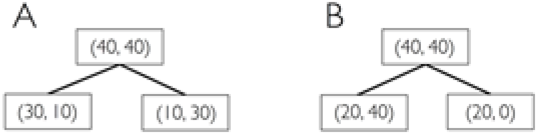
- Another impurity measure is the classification error

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

- This is 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

p( i | t ) is the proportion of the samples that belong to class i for a particular node t

# Example $IG_E$



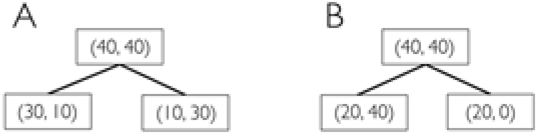
$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

$$I_E = 1 - \max\{p(i|t)\}$$

p( i | t ) is the proportion of the samples that belong to class i for a particular node t

- Consider the two possible splitting scenarios shown in the figure
- Let’s calculate the information gain using the classification error $I_E$ as a splitting criterion
- $I_E(D_p) = 0.5$
  - Case A:
    - $I_E(D_{left}) = 0.25$
    - $I_E(D_{right}) = 0.25$
    - $IG_E = 0.25$
  - Case B:
    - $I_E(D_{left}) = 0.33$
    - $I_E(D_{right}) = 0$
    - $IG_E = 0.25$

# Example $IG_G$



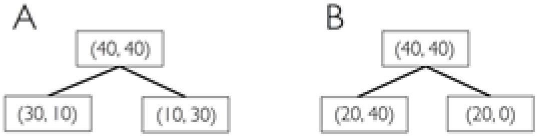
$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

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

p( i | t ) is the proportion of the samples that belong to class i for a particular node t

- Consider the two possible splitting scenarios shown in the figure
- Let’s calculate the information gain using the gini impurity $I_G$ as a splitting criterion
- $I_G(D_p) = 0.5$
  - Case A:
    - $I_G(D_{left}) = 0.375$
    - $I_G(D_{right}) = 0.375$
    - $IG_G = 0.125$
  - Case B:
    - $I_G(D_{left}) = 0.44$
    - $I_G(D_{right}) = 0$
    - $IG_G = 0.17$

# Example $IG_H$



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

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

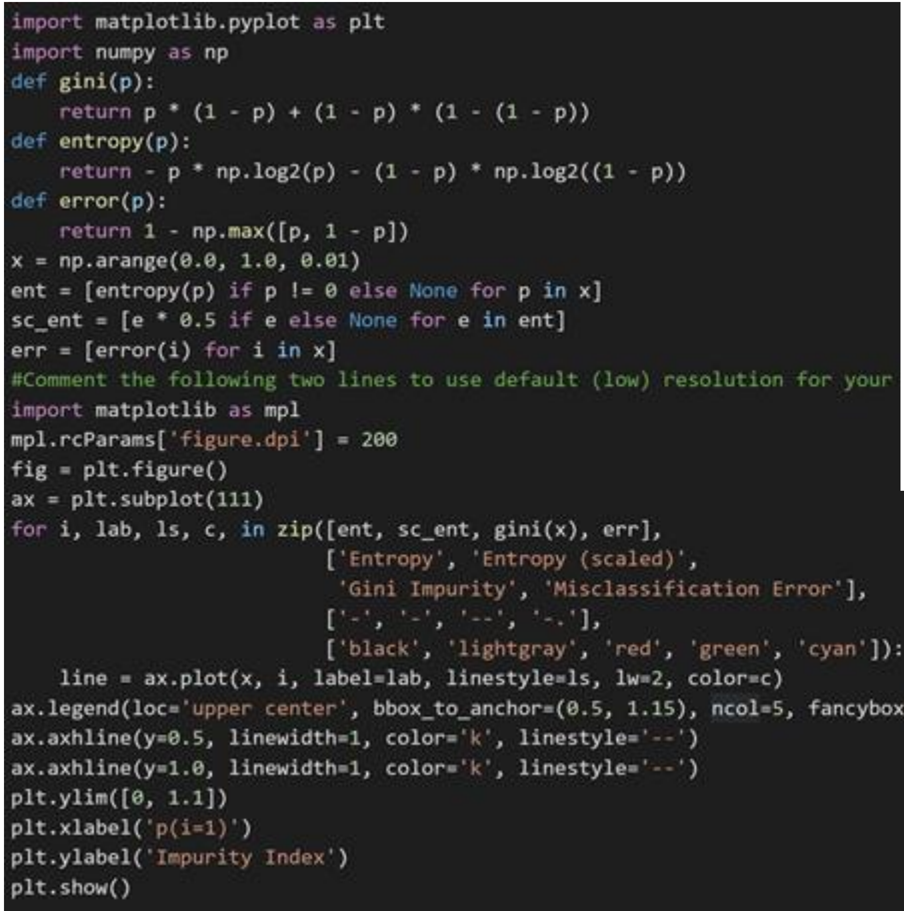
p( i | t ) is the proportion of the samples that belong to class i for a particular node t

- Consider the two possible splitting scenarios shown in the figure
- Let’s calculate the information gain using the entropy $I_H$ as a splitting criterion
- $I_H(D_p) = 1$
  - Case A:
    - $I_H(D_{left}) = 0.81$
    - $I_H(D_{right}) = 0.81$
    - $IG_H = 0.19$
  - Case B:
    - $I_H(D_{left}) = 0.92$
    - $I_H(D_{right}) = 0$
    - $IG_H = 0.31$

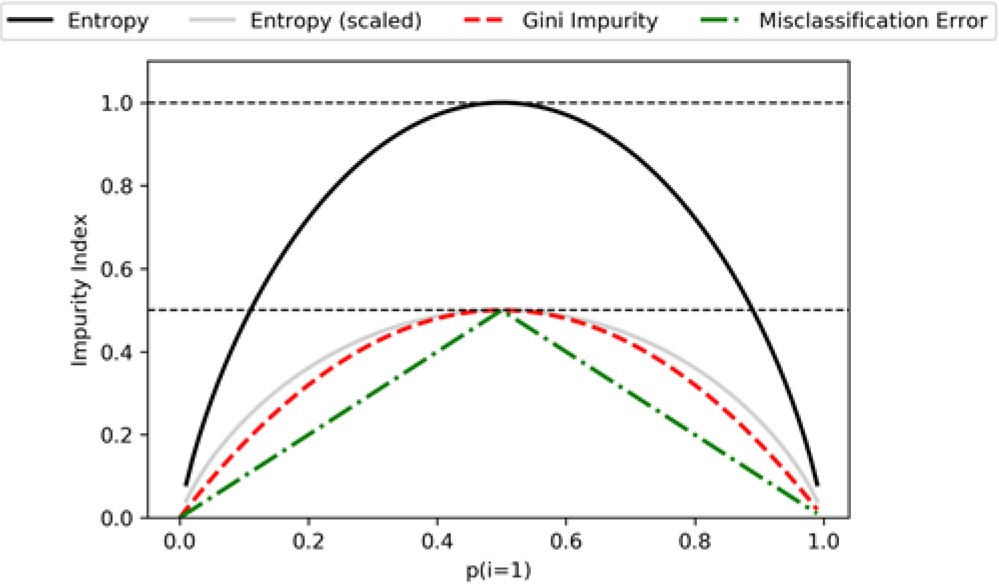
# Code - DecisionTrees.ipynb

- Available here on CoLab

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



```
import matplotlib.pyplot as plt
import numpy as np
def gini(p):
    return p * (1 - p) + (1 - p) * (1 - (1 - p))
def entropy(p):
    return - p * np.log2(p) - (1 - p) * np.log2((1 - p))
def error(p):
    return 1 - np.max([p, 1 - p])
x = np.arange(0.0, 1.0, 0.01)
ent = [entropy(p) if p != 0 else None for p in x]
sc_ent = [e * 0.5 if e else None for e in ent]
err = [error(i) for i in x]
#Comment the following two lines to use default (low) resolution for your
import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 200
fig = plt.figure()
ax = plt.subplot(111)
for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
                          ['Entropy', 'Entropy (scaled)',
                           'Gini Impurity', 'Misclassification Error'],
                          ['-', '-', '--', '-.'],
                          ['black', 'lightgray', 'red', 'green', 'cyan']):
    line = ax.plot(x, i, label=lab, linestyle=ls, lw=2, color=c)
ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15), ncol=5, fancybox=True, shadow=False)
ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
plt.ylim([0, 1.1])
plt.xlabel('p(i=1)')
plt.ylabel('Impurity Index')
plt.show()
```

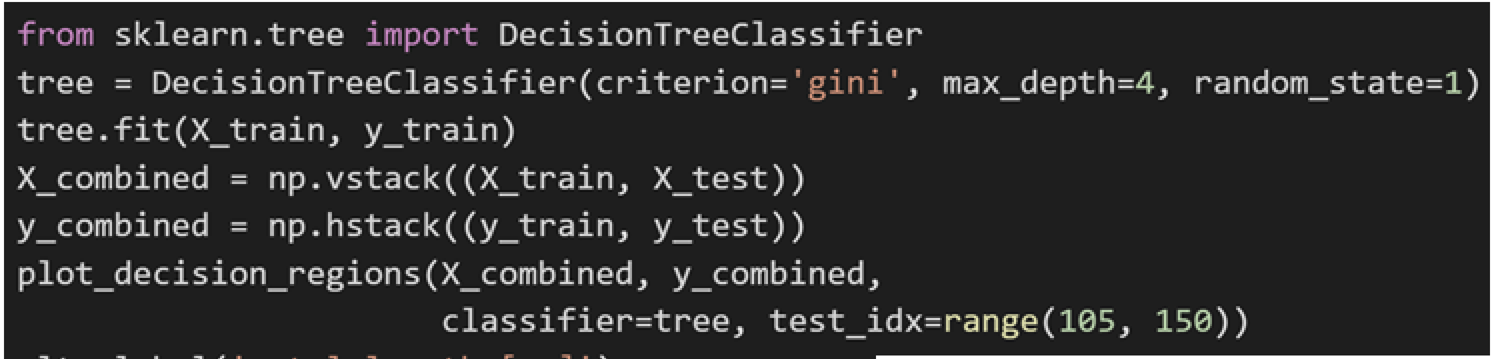


$$I_H(t) = -\sum_{i=1}^{c} p(i|t)\log_2 p(i|t)$$

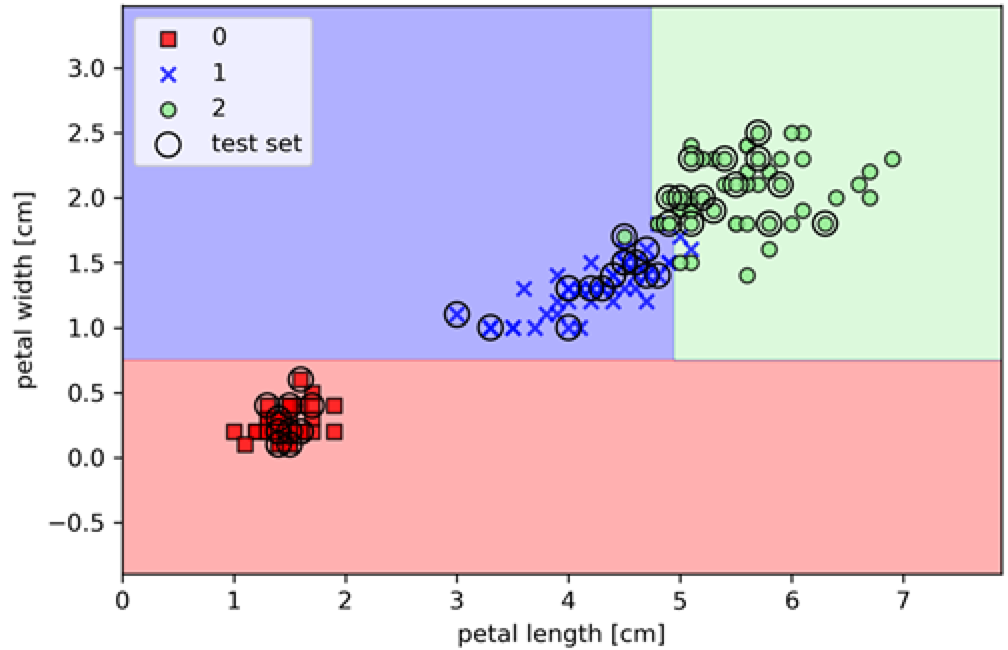
$$I_E = 1 - \max\left\{p(i|t)\right\}$$

# Building a Decision Tree

- Decision trees can build complex decision boundaries by dividing the feature space into rectangles
    - However, we have to be careful since the deeper the decision tree, the more complex the decision boundary becomes, which can easily result in overfitting
- Using scikit-learn, we will now train a decision tree with a maximum depth of 4, using Gini Impurity as a criterion for impurity
- Although feature scaling may be desired for visualization purposes, note that feature scaling is not a requirement for decision tree algorithms

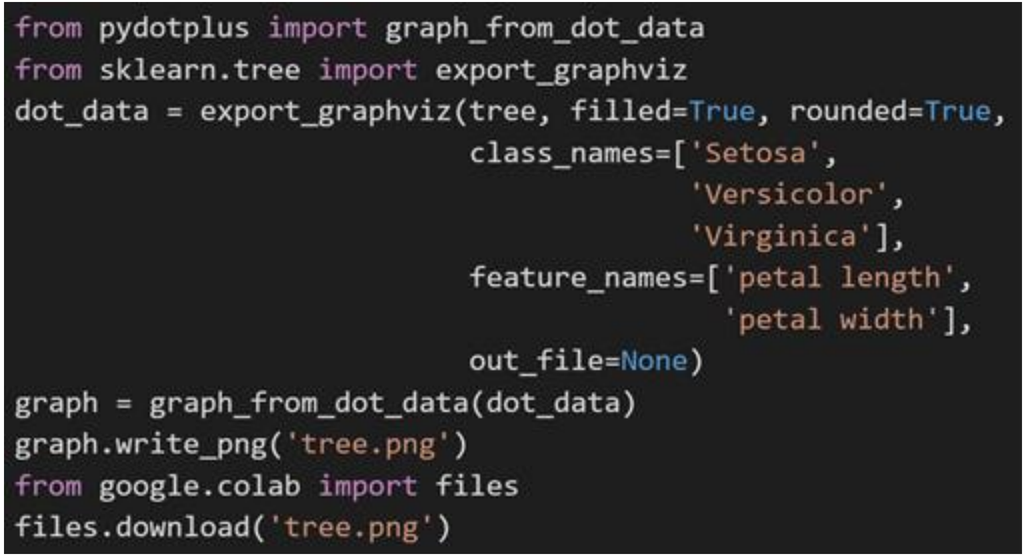


```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(criterion='gini', max_depth=4, random_state=1)
tree.fit(X_train, y_train)
X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))
plot_decision_regions(X_combined, y_combined,
                      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()
```

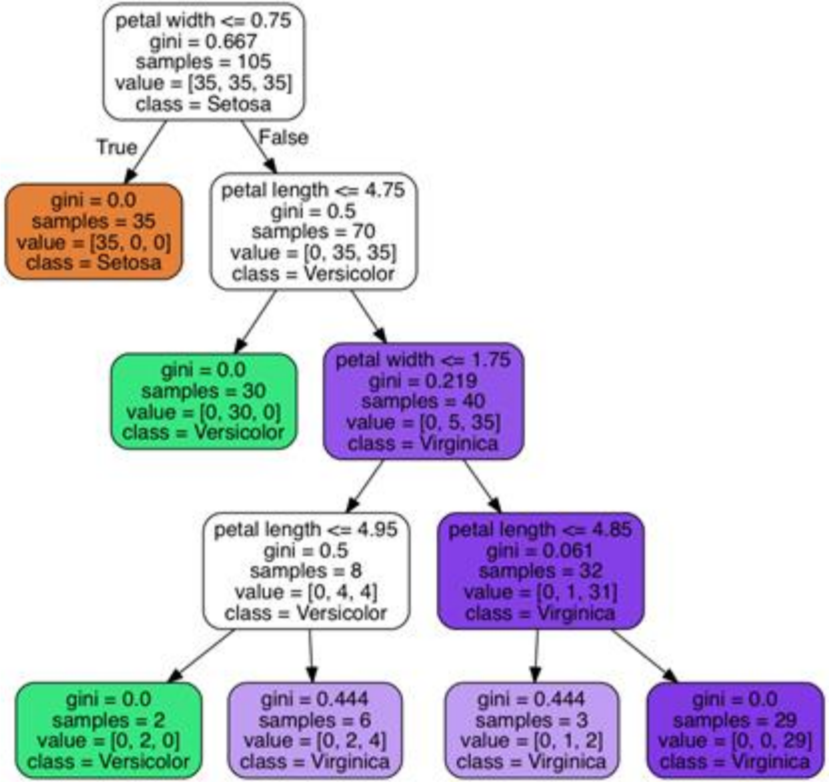


# Visualizing Decision Tree

- The following code will create an image of our decision tree in PNG format
- For this to work you may have to install
  - sudo apt-get install graphviz
  - pip install pydotplus



```
from pydotplus import graph_from_dot_data
from sklearn.tree import export_graphviz
dot_data = export_graphviz(tree, filled=True, rounded=True,
                           class_names=['Setosa',
                                        'Versicolor',
                                        'Virginica'],
                           feature_names=['petal length',
                                          'petal width'],
                           out_file=None)
graph = graph_from_dot_data(dot_data)
graph.write_png('tree.png')
from google.colab import files
files.download('tree.png')
```



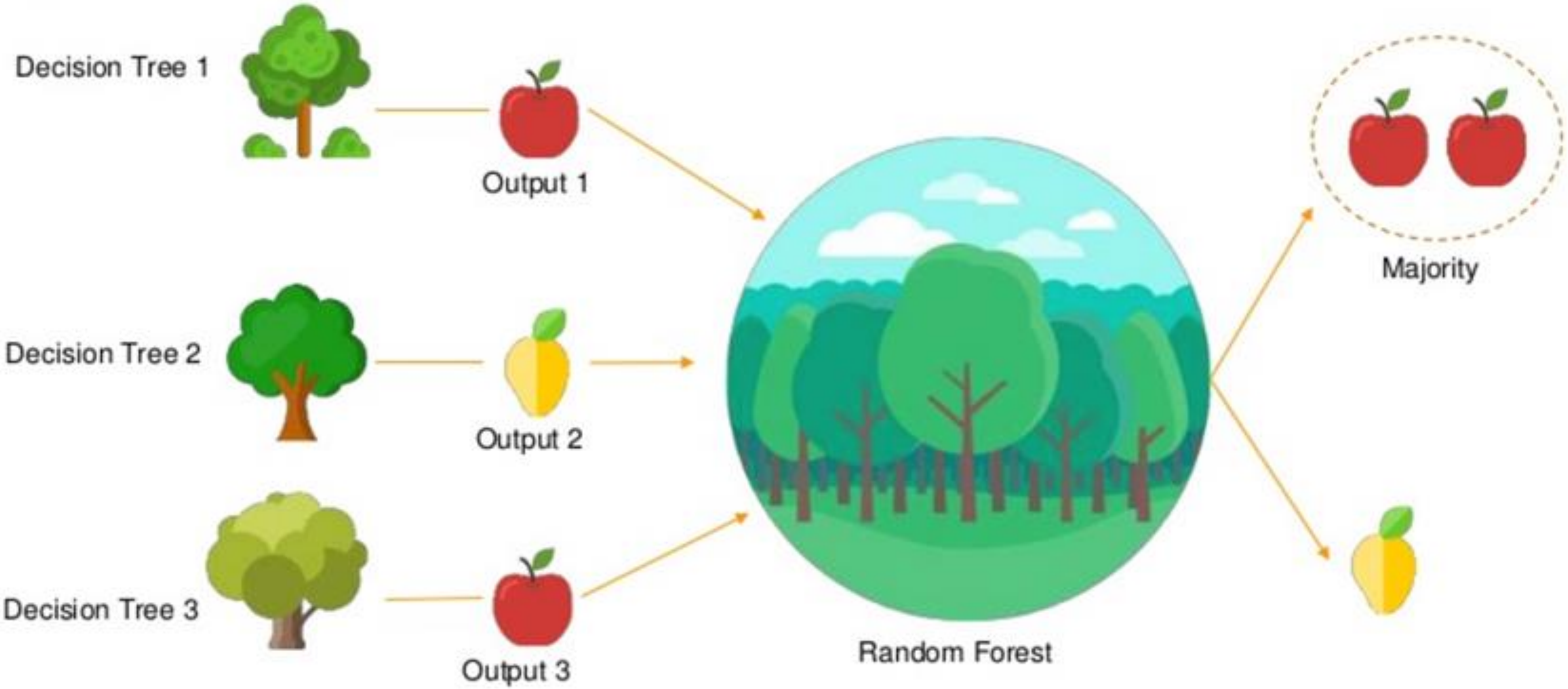
# Random Forests

- A random forest can be considered as an ensemble of decision trees
- The idea behind a random forest is to average multiple (deep) decision trees that individually suffer from high variance, to build a more robust model that has a better generalization performance and is less susceptible to overfitting

# Random Forests Creation

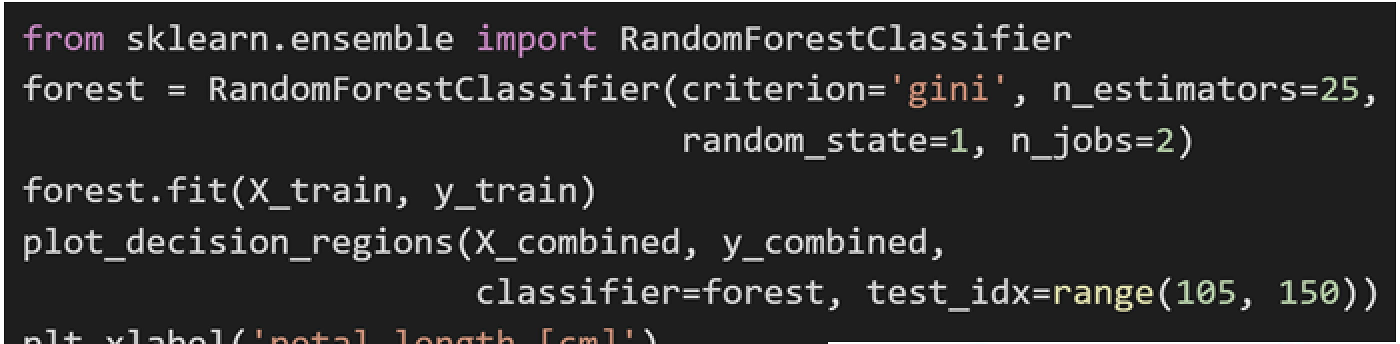
1. Draw a random bootstrap sample of size n (i.e., randomly choose n samples from the training set with replacement)
2. Grow a decision tree from the bootstrap sample. At each node:
   a. Randomly select d features without replacement
   b. Split the node using the feature that provides the best split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

# Random Forests Intuition

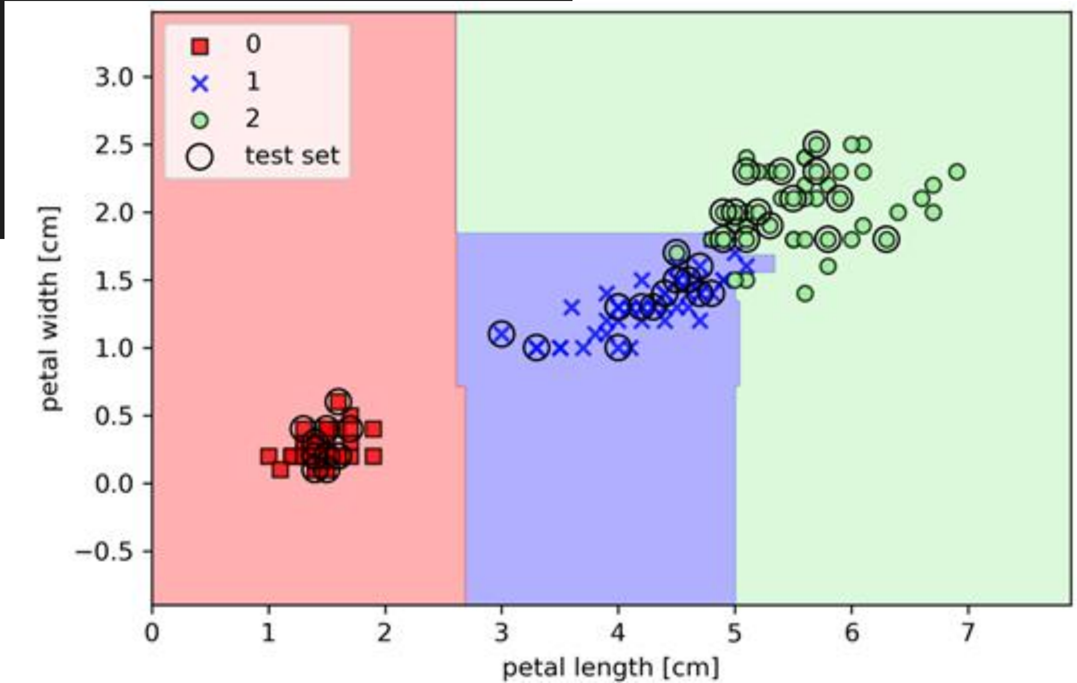


# Bootstrap Sample Size

- Decreasing the size of the bootstrap samples (i.e., decreasing n) increases the diversity among the individual trees
    - The probability that a particular training sample is included in the bootstrap sample is lower
    - This increases the randomness of the random forest
        - Helps to reduce the effect of overfitting
- However, smaller bootstrap samples typically result in a lower overall performance of the random forest, a small gap between training and test performance, but a low test performance overall
- Conversely, increasing the size of the bootstrap sample may increase the degree of overfitting
    - The bootstrap samples, and consequently the individual decision trees, become more similar to each other, they learn to fit the original training dataset more closely



```python
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()
```

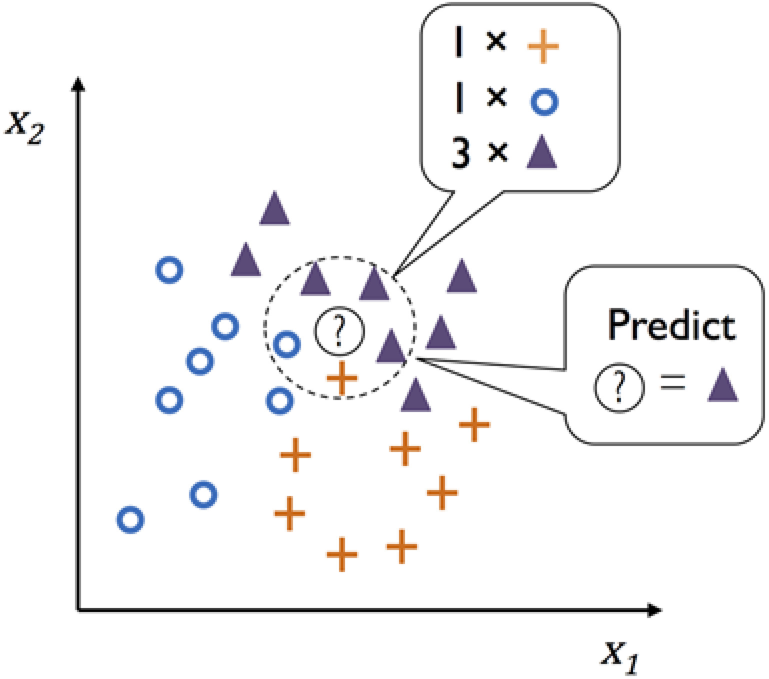


# Outline

- Introduction to more robust algorithms for classification
  - Logistic Regression
  - Support Vector Machines
  - Decision Trees
  - KNN
- Examples using the scikit-learn machine learning library
- Strengths and weaknesses of classifiers

# KNN Algorithm

- The k-nearest neighbor (KNN) classifier is fairly straightforward and can be summarized by the following steps
  - Choose the number of k and a distance metric
  - Find the k-nearest neighbors of the sample that we want to classify
  - Assign the class label by majority vote



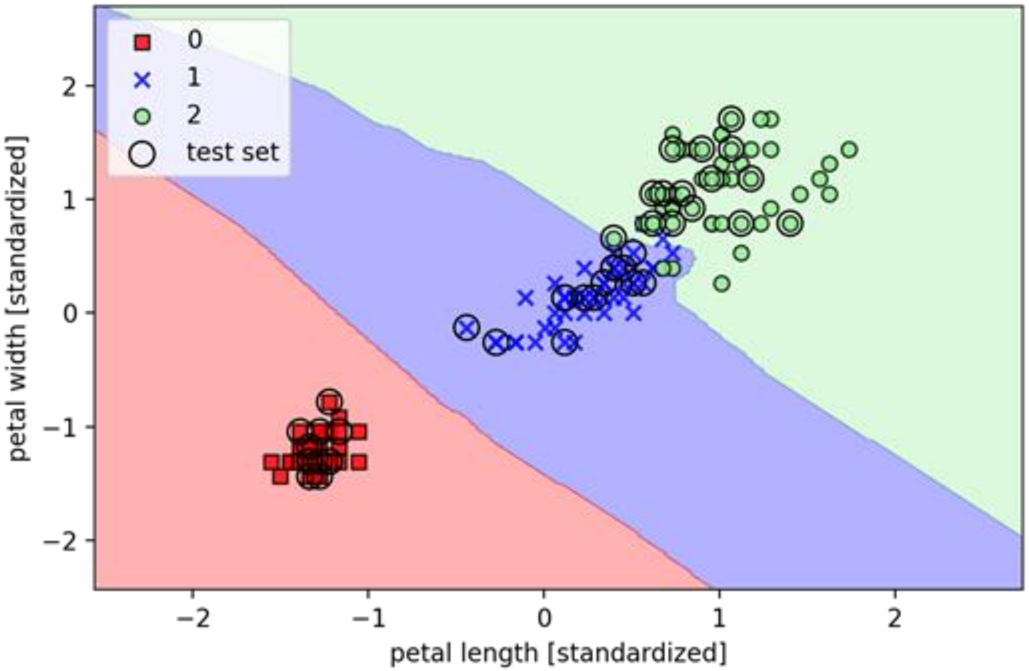
# KNN

- The main advantage of such a memory-based approach is that the classifier immediately adapts as we collect new training data
- However, the downside is that the computational complexity for classifying new samples grows linearly with the number of samples in the training dataset in the worst-case scenario
  - Unless the dataset has very few dimensions (features) and the algorithm has been implemented using efficient data structures such as KD-trees
- Furthermore, we can't discard training samples since no training step is involved
  - Storage space can become a challenge if we are working with large datasets

# Code - KNN.ipynb

- Available here on CoLab

```
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()
```



# KNN

- The right choice of k is crucial to find a good balance between overfitting and underfitting
- We also have to make sure that we choose a distance metric that is appropriate for the features in the dataset
    - Often, a simple Euclidean distance measure is used for real-value samples, for example, the flowers in our Iris dataset, which have features measured in centimeters. However, if we are using a Euclidean distance measure, it is also important to standardize the data so that each feature contributes equally to the distance

# KNN

- The Minkowski distance that we used in the previous code is just a generalization of the Euclidean and Manhattan distance, which can be written as follows

$$d\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \sqrt[p]{\sum_k \left|x_k^{(i)} - x_k^{(j)}\right|^p}$$

- It becomes the Euclidean distance if we set the parameter p=2 or the Manhattan distance at p=1
- Many other distance metrics are available in scikit-learn and can be provided to the metric parameter

# Curse of Dimensionality

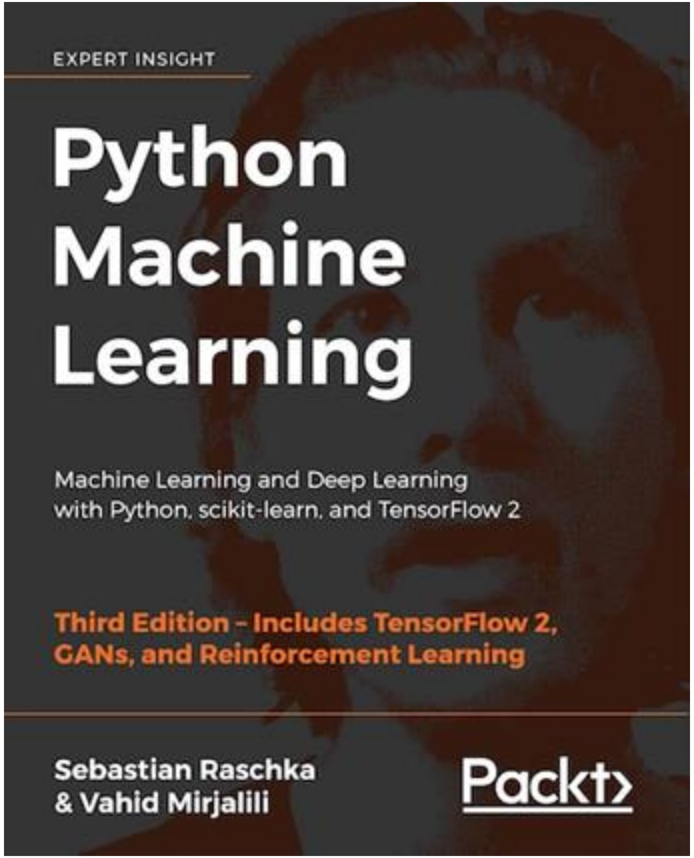
- KNN is susceptible to overfitting due to the curse of dimensionality
    - A phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed- size training dataset
        - Even the closest neighbors being too far away in a high-dimensional space to give a good estimate
- We can use feature selection and dimensionality reduction techniques to help us avoid the curse of dimensionality

# Outline

- Introduction to more robust algorithms for classification
  - Logistic Regression
  - Support Vector Machines
  - Decision Trees
  - KNN
- Examples using the scikit-learn machine learning library
- Strengths and weaknesses of classifiers

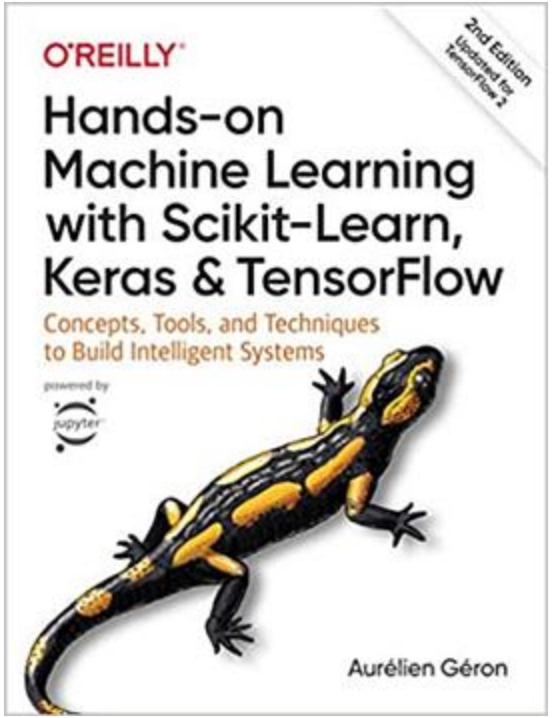
# References

- Most materials in this chapter are based on
  - Book
  - Code


EXPERT INSIGHT
Python Machine Learning
Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2
Third Edition – Includes TensorFlow 2, GANs, and Reinforcement Learning
Sebastian Raschka & Vahid Mirjalili
Packt

# References

- Some materials in this chapter are based on
  - Book
  - Code



O'REILLY

Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow

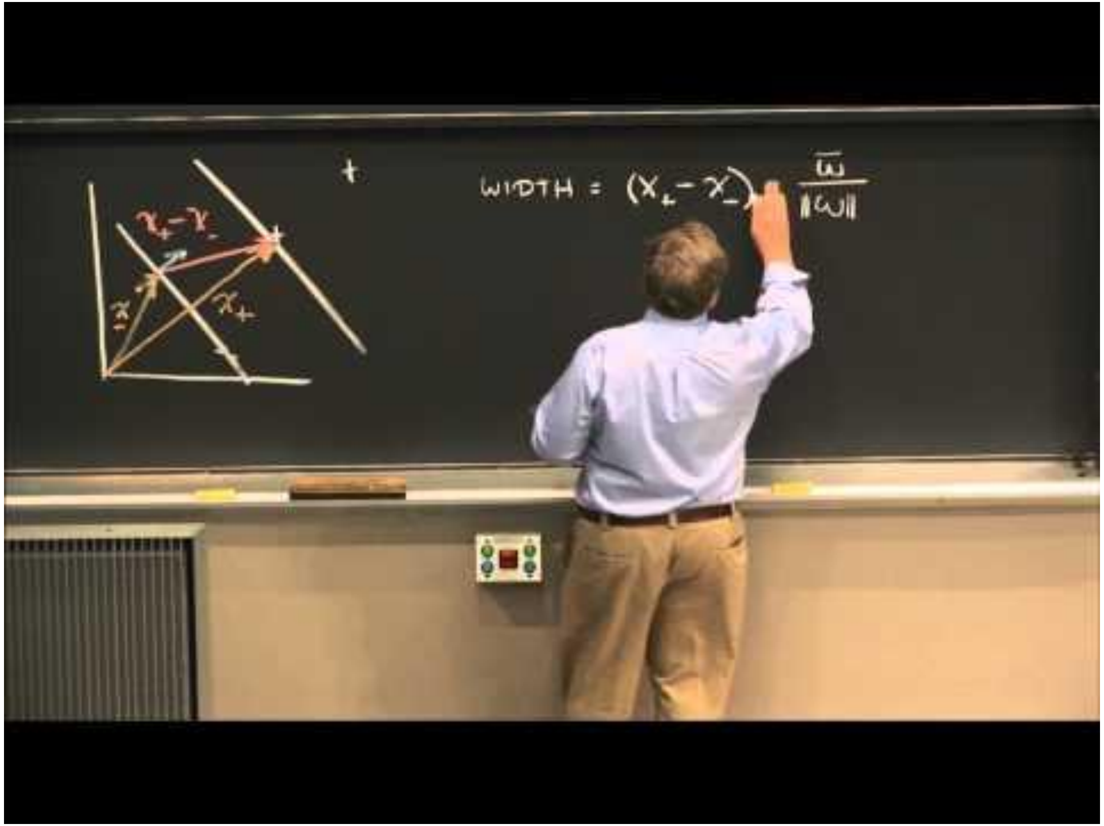
Concepts, Tools, and Techniques to Build Intelligent Systems

2nd Edition Updated for TensorFlow 2

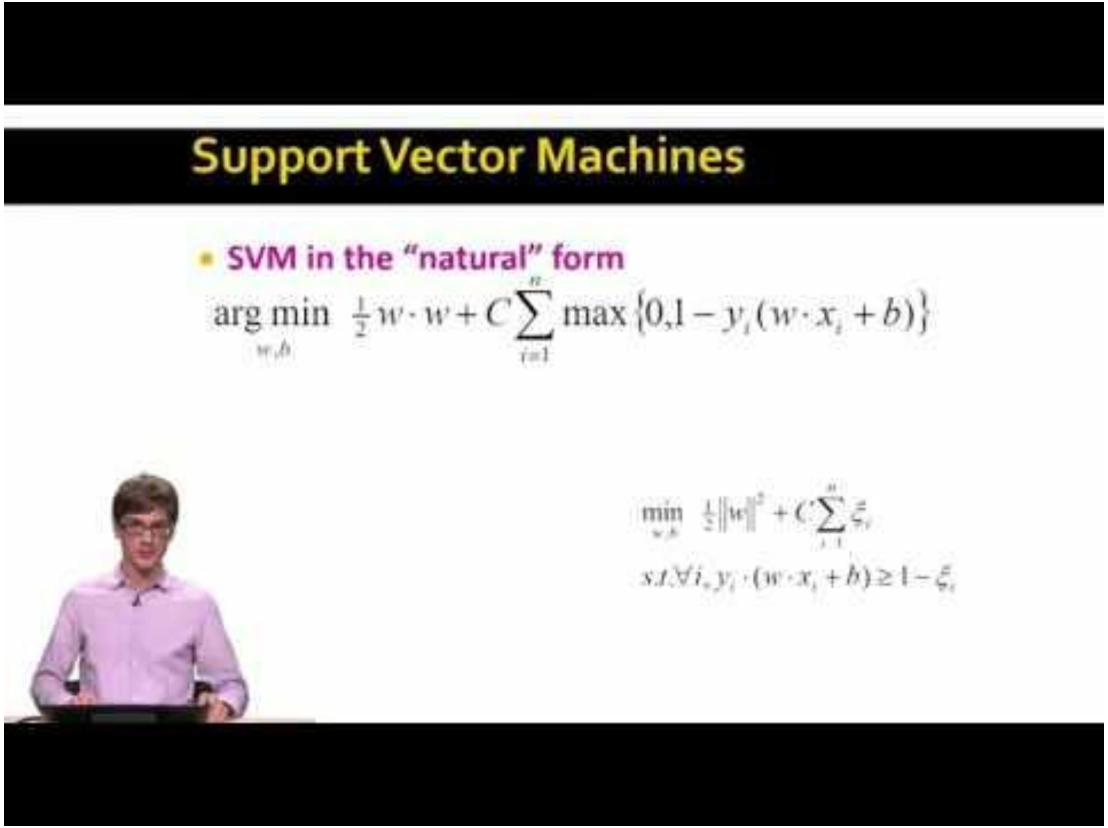
powered by jupyter

Aurélien Géron

# References



# References



Support Vector Machines

- SVM in the “natural” form

$$\arg\min_{w,b} \ \tfrac{1}{2} w \cdot w + C \sum_{i=1}^{n} \max\{0, 1 - y_i (w \cdot x_i + b)\}$$

$$\min_{w,b} \ \tfrac{1}{2}\|w\|^2 + C \sum_{i=1}^{n} \xi_i$$

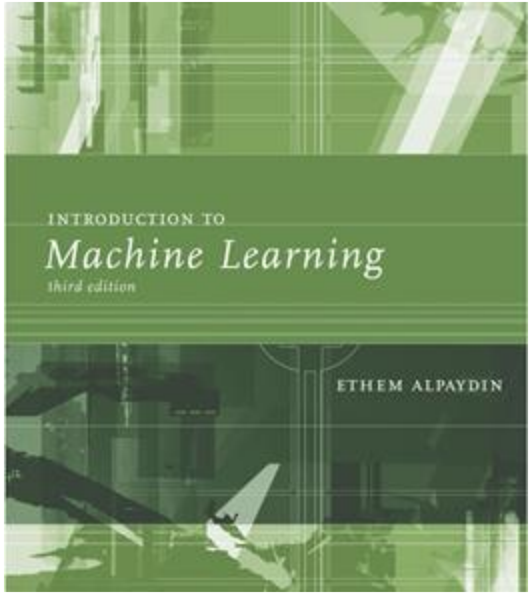
$$s.t. \forall i, y_i \cdot (w \cdot x_i + b) \ge 1 - \xi_i$$

# References

- CS229 Lecture notes

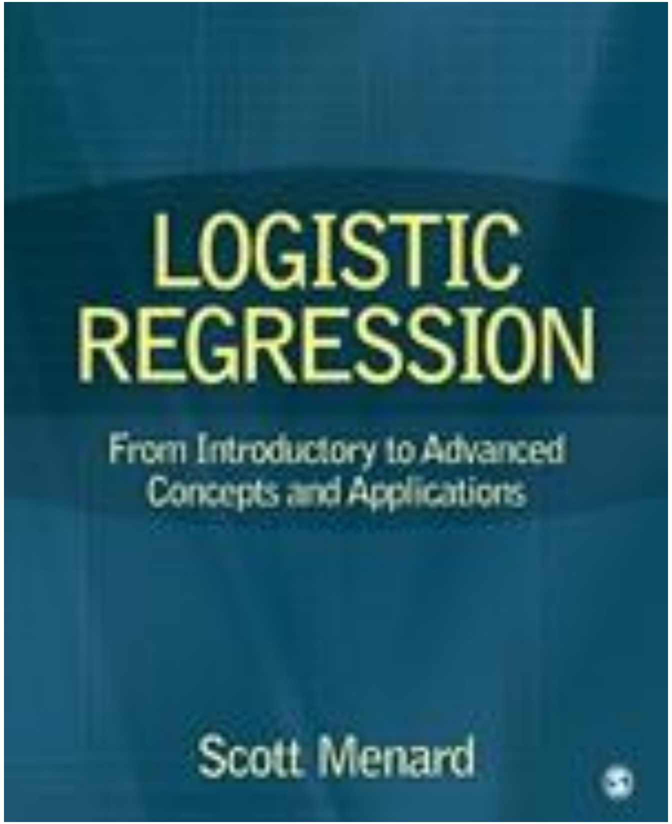
# References

- Introduction to Machine Learning, Third Edition
  - Ethem Alpaydin



# References

- Logistic Regression: From Introductory to Advanced Concepts and Applications
- Scott Menard, 2010



# References

| Color | Diam | Label |
|---|---|---|
| Green | 3 | Apple |
| Yellow | 3 | Apple |
| Red | 1 | Grape |
| Red | 1 | Grape |
| Yellow | 3 | Lemon |

# References

# Exercise 1

- What is the fundamental idea behind Support Vector Machines?
- What is a support vector?
- Why is it important to scale the inputs when using SVMs?
- Can an SVM classifier output a confidence score when it classifies an instance? What about a probability?
- Say you’ve trained an SVM classifier with an RBF kernel, but it seems to underfit the training set. Should you increase or decrease γ (gamma)? What about C?

# Exercise 2

- Train a LinearSVC on a linearly separable dataset. Then train an SVC and a SGDClassifier on the same dataset. See if you can get them to produce roughly the same model.
- Train an SVM classifier on the MNIST dataset. Since SVM classifiers are binary classifiers, you will need to use one-versus-the-rest to classify all 10 digits. You may want to tune the hyperparameters using small validation sets to speed up the process. What accuracy can you reach?

# Exercise 3

- What is the approximate depth of a Decision Tree trained (without restrictions) on a training set with one million instances?
- Is a node's Gini impurity generally lower or greater than its parent's? Is it generally lower/greater, or always lower/greater?
- If a Decision Tree is overfitting the training set, is it a good idea to try decreasing max_depth?
- If a Decision Tree is underfitting the training set, is it a good idea to try scaling the input features?
- If it takes one hour to train a Decision Tree on a training set containing 1 million instances, roughly how much time will it take to train another Decision Tree on a training set containing 10 million instances?

# Exercise 4

- Try to build a classifier for the MNIST dataset that achieves over 97% accuracy on the test set
    - Hint: the KNeighborsClassifier works quite well for this task; you just need to find good hyperparameter values (try a grid search on the weights and n_neighbors hyperparameters)

# Exercise 5

- Write a function that can shift an MNIST image in any direction (left, right, up, or down) by one pixel.
- Then, for each image in the training set, create four shifted copies (one per direction) and add them to the training set
- Finally, train your best model on this expanded training set and measure its accuracy on the test set
- You should observe that your model performs even better now
    - This technique of artificially growing the training set is called data augmentation or training set expansion

# Exercise 6

- Tackle the Titanic dataset
    - A great place to start is on Kaggle