SUPERVISED LEARNING

MODULE 7

CLASSIFICATION

Content

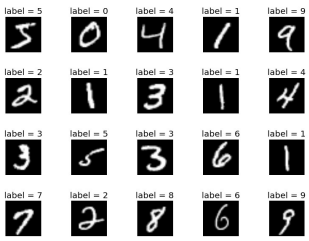
- The most common supervised learning tasks are regression (predicting values) and classification (predicting classes)
- In Module 2 we explored a regression task, predicting housing values, using various algorithms such as Linear Regression.
- In this module we study classification

MNIST

- In this module, we will be using the MNIST dataset
  - A set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau
  - Each image is labeled with the digit it represents

MNIST

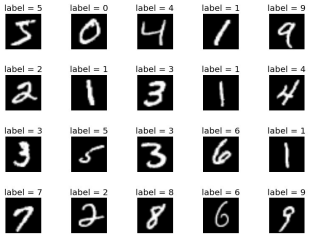
- In this module, we will be using the MNIST dataset
  - This set has been studied so much that it is often called the “Hello World” of Machine Learning
  - Whenever people come up with a new classification algorithm, they are curious to see how it will perform on MNIST.
  - Whenever someone learns Machine Learning, sooner or later they tackle MNIST.



MNIST

- Scikit-Learn provides many helper functions to download popular datasets.
  - MNIST is one of them

```
>>> from sklearn.datasets import fetch_openml
>>> mnist = fetch_openml('mnist_784', version=1)
>>> mnist.keys()
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details',
           'categories', 'url'])
```



MNIST

- Datasets loaded by Scikit-Learn generally have a similar dictionary structure including:
  - A `DESCR` key describing the dataset
  - A `data` key containing an array with one row per instance and one column per feature
  - A `target` key containing an array with the labels
  - The MNIST database contains a total of 70000 examples of handwritten digits of size 28x28 pixels, labeled from 0 to 9

MNIST

- Let's look at these arrays:

```
>>> X, y = mnist["data"], mnist["target"]
>>> X.shape
(70000, 784)
>>> y.shape
(70000,)
```

- There are 70,000 images
- Each image has 784 features
  - Each image is 28×28 pixels
- Each feature simply represents one pixel's intensity, from 0 (white) to 255 (black)

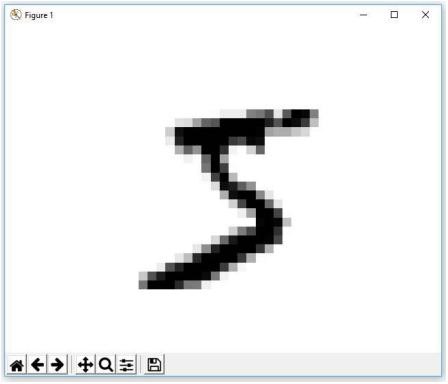
MNIST

> Let's take a peek at one digit from the dataset

```
import matplotlib as mpl
import matplotlib.pyplot as plt

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)

plt.imshow(some_digit_image, cmap = mpl.cm.binary, interpolation="nearest")
plt.axis("off")
plt.show()
```

MNIST

> This looks like a 5, and that's what the label tells us:




```
>>> y[0]
'5'
```

> Note that the label is a string.
  – We prefer numbers, so let's cast y to integers:

```
>>> y = y.astype(np.uint8)
```

MNIST

MNIST

> The MNIST dataset is already split into
  – a training set (the first 60,000 images)
  – a test set (the last 10,000 images)

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

Training a Binary Classifier

> Let's simplify the problem for now and only try to identify one digit—for example, the number 5.

> This "5-detector" will be an example of a *binary classifier*,

– capable of distinguishing between just two classes

- 5
- not-5

> Let's create the target vectors for this classification task:

```
y_train_5 = (y_train == 5)  # True for all 5s, False for all other digits
y_test_5 = (y_test == 5)
```

Training a Binary Classifier

> *Stochastic Gradient Descent* (SGD) classifier

– Available in Scikit-Learn

`SGDClassifier`

– Capable of handling very large datasets efficiently

– SGD deals with training instances independently

- Well suited for *online learning*

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

– Now you can use it to detect images of the number 5:

```
>>> sgd_clf.predict([some_digit])
array([ True])
```

Classification: "True vs. False" and "Positive vs. Negative"

> **An Aesop's Fable: The Boy Who Cried Wolf** (compressed)

A shepherd boy gets bored tending the town's flock. To have some fun, he cries out, "Wolf!" even though no wolf is in sight. The villagers run to protect the flock, but then get really mad when they realize the boy was playing a joke on them.

[Iterate previous paragraph N times.]

One night, the shepherd boy sees a real wolf approaching the flock and calls out, "Wolf!" The villagers refuse to be fooled again and stay in their houses. The hungry wolf turns the flock into lamb chops. The town goes hungry. Panic ensues.

Classification: «True vs. False» and «Positive vs. Negative»

> **An Aesop's Fable: The Boy Who Cried Wolf** (compressed)

– Let's make the following definitions:

- "Wolf" is a **positive class**.
- "No wolf" is a **negative class**.

Classification: «True vs. False» and «Positive vs. Negative»

> We can summarize our "wolf-prediction" model using a 2x2 confusion matrix that depicts all four possible outcomes

True Positive (TP): • Reality: A wolf threatened. • Shepherd said: "Wolf." • Outcome: Shepherd is a hero.	**False Positive (FP):** • Reality: No wolf threatened. • Shepherd said: "Wolf." • Outcome: Villagers are angry at shepherd for waking them up.
False Negative (FN): • Reality: A wolf threatened. • Shepherd said: "No wolf." • Outcome: The wolf ate all the sheep.	**True Negative (TN):** • Reality: No wolf threatened. • Shepherd said: "No wolf." • Outcome: Everyone is fine.

Classification: «True vs. False» and «Positive vs. Negative»

> A **True Positive** is an outcome where the model correctly predicts the positive class.

> Similarly, a **True Negative** is an outcome where the model correctly predicts the negative class.

True Positive (TP): • Reality: A wolf threatened. • Shepherd said: "Wolf." • Outcome: Shepherd is a hero.	**False Positive (FP):** • Reality: No wolf threatened. • Shepherd said: "Wolf." • Outcome: Villagers are angry at shepherd for waking them up.
False Negative (FN): • Reality: A wolf threatened. • Shepherd said: "No wolf." • Outcome: The wolf ate all the sheep.	**True Negative (TN):** • Reality: No wolf threatened. • Shepherd said: "No wolf." • Outcome: Everyone is fine.

Classification: «True vs. False» and «Positive vs. Negative»

> A **False Positive** is an outcome where the model incorrectly predicts the positive class.

> A **False Negative** is an outcome where the model incorrectly predicts the negative class.

True Positive (TP): • Reality: A wolf threatened. • Shepherd said: "Wolf." • Outcome: Shepherd is a hero.	**False Positive (FP):** • Reality: No wolf threatened. • Shepherd said: "Wolf." • Outcome: Villagers are angry at shepherd for waking them up.
False Negative (FN): • Reality: A wolf threatened. • Shepherd said: "No wolf." • Outcome: The wolf ate all the sheep.	**True Negative (TN):** • Reality: No wolf threatened. • Shepherd said: "No wolf." • Outcome: Everyone is fine.

Classification: «True vs. False» and «Positive vs. Negative»

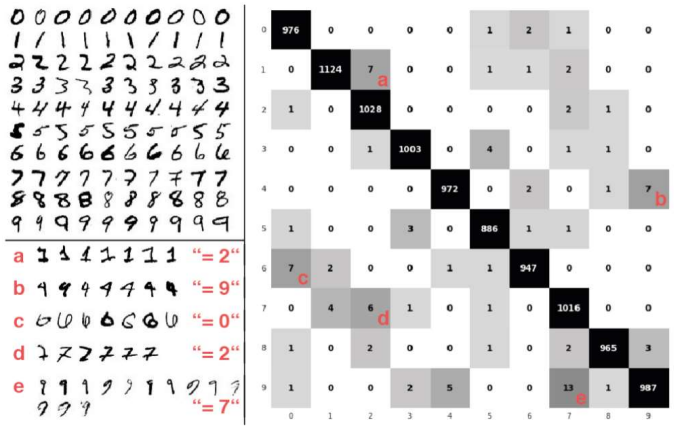
> Our new **KEYWORDS**:

- Confusion Matrix
- Positive Class
- Negative Class
- False Negative
- False Positive
- True Negative
- True Positive

Confusion Matrix

> A better way to evaluate the performance of a classifier is to look at the *confusion matrix*.

> The general idea is to count the number of times instances of class A are classified as class B.



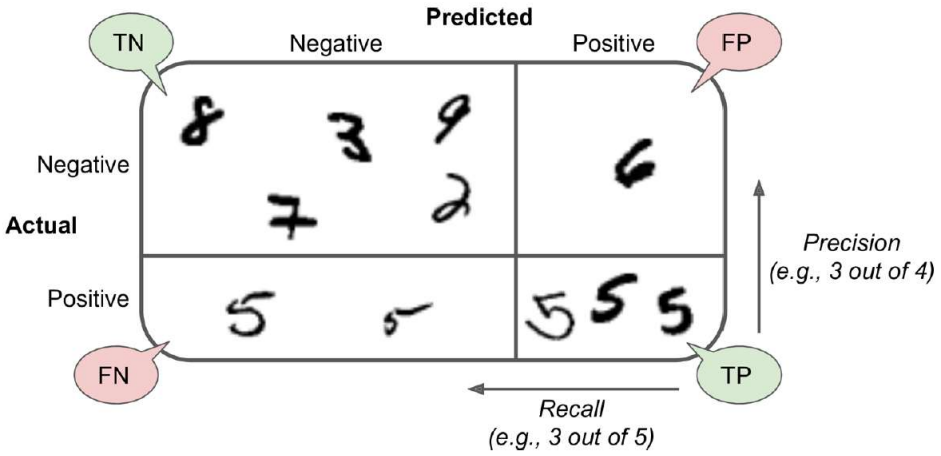
Confusion Matrix

> A better way to evaluate the performance of a classifier is to look at the *confusion matrix*.

> The general idea is to count the number of times instances of class A are classified as class B.

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53272,  1307],
       [ 1077,  4344]])
```

Confusion Matrix



Confusion Matrix

> A perfect classifier would have only true positives and true negatives, so its confusion matrix would have nonzero values only on its main diagonal:

```
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579,     0],
       [    0, 5421]])
```

Classification: Accuracy

> Accuracy is one metric for evaluating classification models.
Informally, **Accuracy** is the fraction of predictions our model got right.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

> For binary classification, accuracy can also be calculated in terms of positives and negatives as follows:

> *TP* = True Positives
> *TN* = True Negatives
> *FP* = False Positives
> *FN* = False Negatives.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Classification: Accuracy

> Let's try calculating accuracy for the following model that classified **100 tumors** as **Malignant** (the positive class) or **Benign** (the negative class):

True Positive (TP): • Reality: Malignant • ML model predicted: Malignant • **Number of TP results: 1**	**False Positive (FP):** • Reality: Benign • ML model predicted: Malignant • **Number of FP results: 1**
False Negative (FN): • Reality: Malignant • ML model predicted: Benign • **Number of FN results: 8**	**True Negative (TN):** • Reality: Benign • ML model predicted: Benign • **Number of TN results: 90**

Classification: Accuracy

> Let's try calculating accuracy for the following model that classified **100 tumors** as **Malignant** (the positive class) or **Benign** (the negative class):

True Positive (TP): • Reality: Malignant • ML model predicted: Malignant • **Number of TP results: 1**	**False Positive (FP):** • Reality: Benign • ML model predicted: Malignant • **Number of FP results: 1**
False Negative (FN): • Reality: Malignant • ML model predicted: Benign • **Number of FN results: 8**	**True Negative (TN):** • Reality: Benign • ML model predicted: Benign • **Number of TN results: 90**

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{1 + 90}{1 + 90 + 1 + 8} = 0.91$$

Classification: Accuracy

> Accuracy comes out to 0.91, or **91%** (**91 correct predictions out of 100 total examples**).

> That means **our tumor classifier is <u>doing a great job of identifying malignancies</u>**.

– Is it right?

• **NO !!!**

Classification: Accuracy

- Accuracy comes out to 0.91, or **91%** (**91 correct predictions out of 100 total examples**).
- Of the 100 tumor examples, 91 are benign (90 TNs and 1 FP) and 9 are malignant (1 TP and 8 FNs).
  - Of the 91 benign tumors, the model correctly identifies 90 as benign. **That's good.**
  - However, of the 9 malignant tumors, the model only correctly identifies 1 as malignant - **a terrible outcome**, as 8 out of 9 malignancies go undiagnosed!

Classification: Accuracy

- Accuracy comes out to 0.91, or **91%** (**91 correct predictions out of 100 total examples**).
- Accuracy alone doesn't tell the full story when you're working with a **class-imbalanced data set**, like this one, where there is a significant disparity between the number of positive and negative labels.
  - **91 Benign**
  - **9 Malignant**

Precision and Recall

> Precision

$$\text{precision} = \frac{TP}{TP + FP}$$

> Recall

$$\text{recall} = \frac{TP}{TP + FN}$$

Precision and Recall

> **Precision :** What proportion of positive identifications was actually correct?

$$\text{Precision} = \frac{TP}{TP + FP}$$

> A model that produces no false positives has a precision of 1.0

True Positives (TPs): 1	False Positives (FPs): 1
False Negatives (FNs): 8	True Negatives (TNs): 90

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{1}{1 + 1} = 0.5$$

Our model has a precision of 0.5—in other words, when it predicts a tumor is malignant, it is correct 50% of the time.

Precision and Recall

> **Recall :** What proportion of actual positives was identified correctly?

$$\text{Recall} = \frac{TP}{TP + FN}$$

> A model that produces no false negatives has a recall of 1.0.

True Positives (TPs): 1	False Positives (FPs): 1
False Negatives (FNs): 8	True Negatives (TNs): 90

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{1}{1 + 8} = 0.11$$

Our model has a recall of 0.11—in other words, it correctly identifies 11% of all malignant tumors.

Precision and Recall

> Scikit-Learn provides several functions to compute classifier metrics, including precision and recall

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_pred)     # == 4344 / (4344 + 1307)
0.76871350203503808
>>> recall_score(y_train_5, y_train_pred)  # == 4344 / (4344 + 1077)
0.79136690647482011
```

F_1 score

> It is often convenient to combine precision and recall into a single metric called the F_1 *score*, in particular if you need a simple way to compare two classifiers.

> The F_1 score is the *harmonic mean* of precision and recall

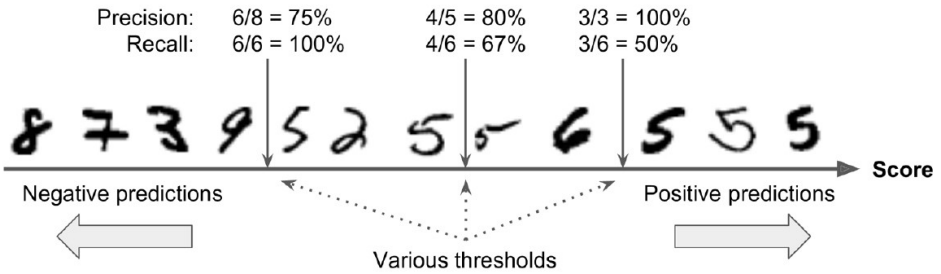
$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

> To compute the F_1 score, simply call the **`f1_score()`** function

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_pred)
0.78468208092485547
```

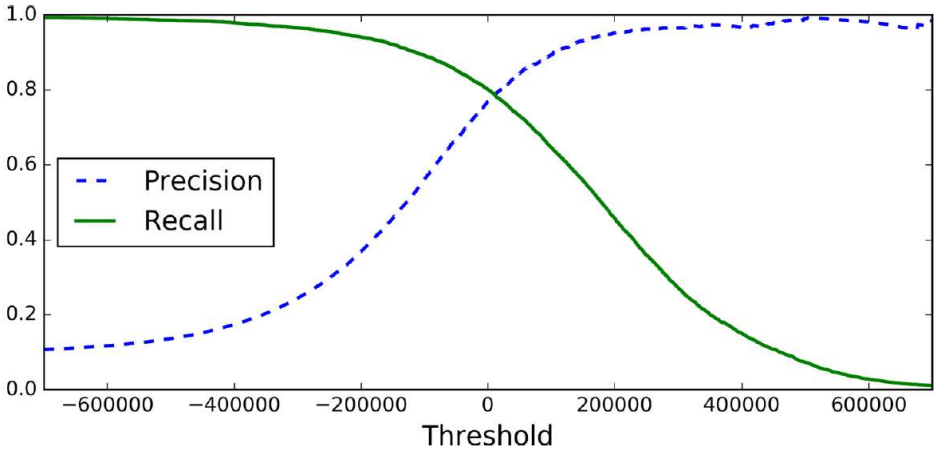
Precision/Recall Tradeoff

> Decision threshold and precision/recall tradeoff

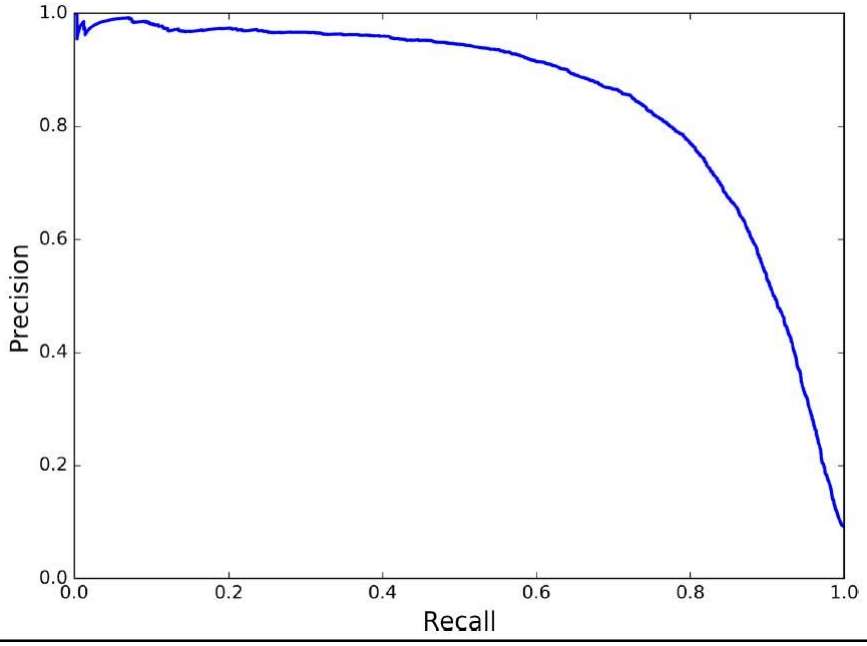


Precision/Recall Tradeoff

> Precision and recall versus the decision threshold



Precision versus Recall

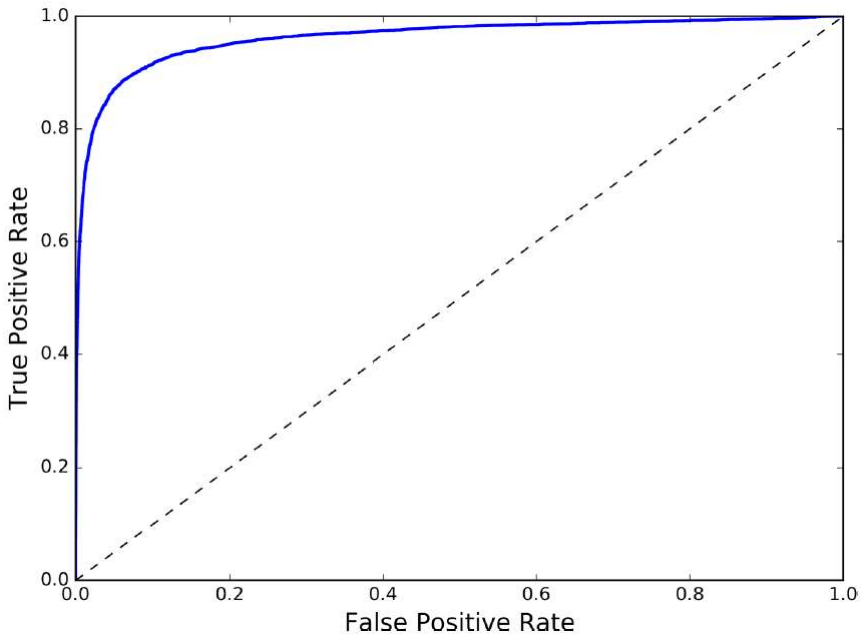


The ROC Curve

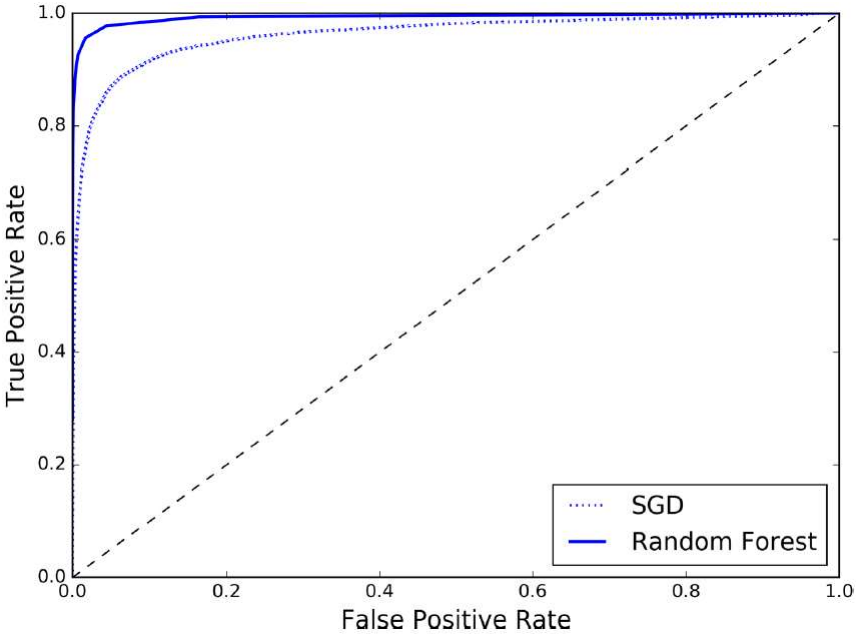
- The *receiver operating characteristic* (ROC) curve is another common tool used with binary classifiers
- It is very similar to the precision/recall curve
- The ROC curve plots the *true positive rate* (another name for recall) against the *false positive rate*.
- The FPR is the ratio of negative instances that are incorrectly classified as positive.

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

The ROC Curve



Comparing ROC curves



TRAINING LINEAR MODELS

Content

> In this module, we will start by looking at the **Linear Regression** model, one of the simplest models there is.
> We will discuss two very different ways to train it:
  - Using a direct "closed-form" equation
    - Directly computes the model parameters that best fit the model to the training set
      - Minimize the cost function over the training set
  - Using an iterative optimization approach
    - Gradient Descent (GD)
      - Gradually tweaks the model parameters to minimize the cost function over the training set
      - Eventually converging to the same set of parameters as the first method.

Content

> Next we will look at **Polynomial Regression**
  - More complex model that can fit **nonlinear** datasets.
  - It is more prone to over-fitting the training data
  - How to detect whether or not this is the case
  - Regularization techniques that can reduce the risk of over-fitting the training set
> Finally, we will look at two more models that are commonly used for classification tasks
  - Logistic Regression
  - Softmax Regression.

Linear Regression

> A linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the *bias term*

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- $\hat{y}$ is the predicted value.
- n is the number of features.
- x_i is the i^{th} feature value.
- θ_j is the j^{th} model parameter
  including the bias term θ_0 and the feature weights $(\theta_1, \theta_2, \cdots, \theta_n)$

Linear Regression

> This can be written much more concisely using a vectorized form

$$\hat{y} = h_\theta(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

- θ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n.
- θ^T is the transpose of θ (a row vector instead of a column vector).
- $\mathbf{x}$ is the instance's *feature vector*, containing x_0 to x_n, with x_0 always equal to 1.
- $\theta^T \cdot \mathbf{x}$ is the dot product of θ^T and $\mathbf{x}$.
- h_θ is the hypothesis function, using the model parameters θ.

Linear Regression

> You know the Linear Regression model

$$\hat{y} = h_\theta(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

> So now how do we train it?

Linear Regression

> You know the Linear Regression model

$$\hat{y} = h_\theta(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

> So now how do we train it?

> Mean Square Error (MSE)

> MSE cost function for a Linear Regression model

$$\mathrm{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

Linear Regression

> You know the Linear Regression model

$$\hat{y} = h_\theta(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

> So now how do we train it?

> Mean Square Error (MSE)

> MSE cost function for a Linear Regression model

$$\text{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

> You need to find θ that minimizes the **RMSE**

The Normal Equation

> To find the value of θ that minimizes the cost function, there is a ***closed-form solution***

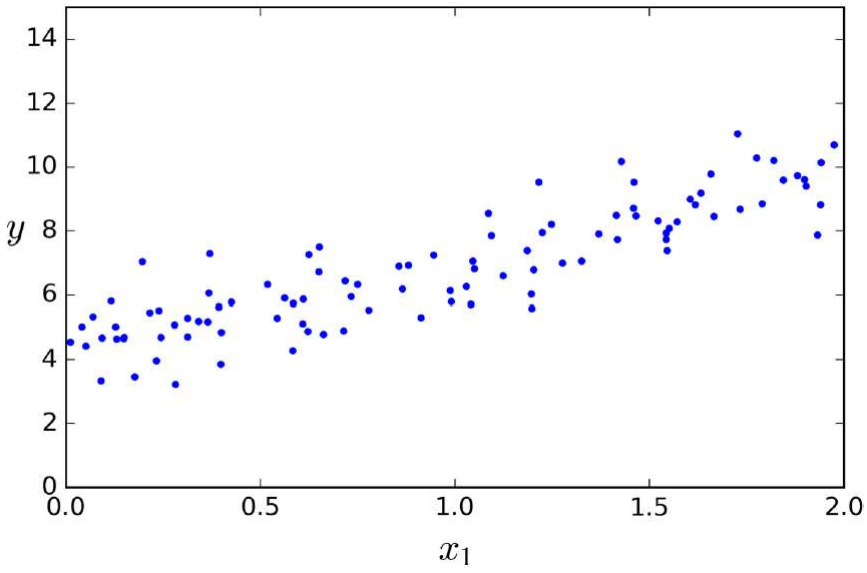
– A mathematical equation that gives the result directly

• It is called the ***Normal Equation***

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

- $\hat{\theta}$ is the value of θ that minimizes the cost function.
- $\mathbf{y}$ is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

The Normal Equation



The Normal Equation

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

The Normal Equation

> We will use the inv() function from NumPy's Linear Algebra module (np.linalg) to compute the inverse of a matrix, and the dot() method for matrix multiplication:

```
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

The Normal Equation

> The actual function that we used to generate the data

$y = 4 + 3x_0 + \textit{Gaussian noise}$

> Let's see what the equation found:

```
>>> theta_best
array([[ 4.21509616],
       [ 2.77011339]])
```

Linear Regression model predictions

> Now you can make predictions

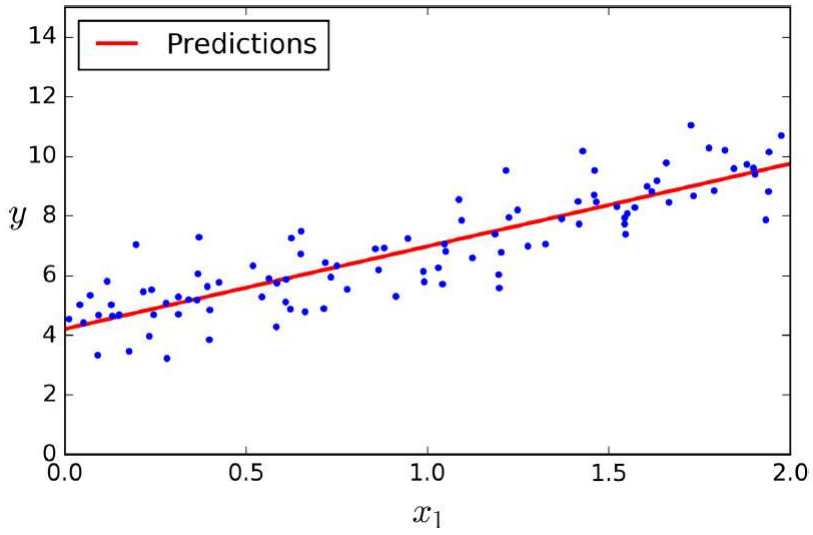
```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new]
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[ 4.21509616],
       [ 9.75532293]])
```

Linear Regression model predictions

> Let's plot this model's predictions

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

Linear Regression model predictions



Linear Regression model predictions

> The equivalent code using Scikit-Learn

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 4.21509616]), array([[ 2.77011339]]))
>>> lin_reg.predict(X_new)
array([[ 4.21509616],
       [ 9.75532293]])
```

Computational Complexity

- The Normal Equation computes the inverse of $\mathbf{X}^T \cdot \mathbf{X}$
  - It is an $n \times n$ matrix
  - n is the number of features
- The *computational complexity* of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^3)$
- If you double the number of features, you multiply the computation time by roughly $2^{2.4} = \mathbf{5.3}$ to $2^3 = \mathbf{8}$

Computational Complexity

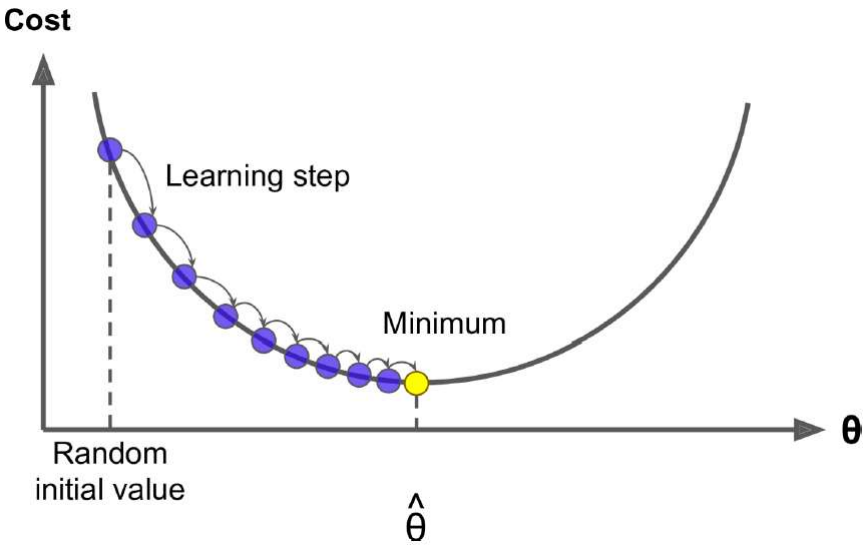
- On the positive side, this equation is linear with regards to the number of instances in the training set
  - it is $O(m)$
  - it handles large training sets efficiently, provided they can fit in memory.
- Once you have trained your Linear Regression model, predictions are very fast
  - the computational complexity is linear with regards to both
    - The number of instances you want to make predictions on
    - The number of features.

Gradient Descent

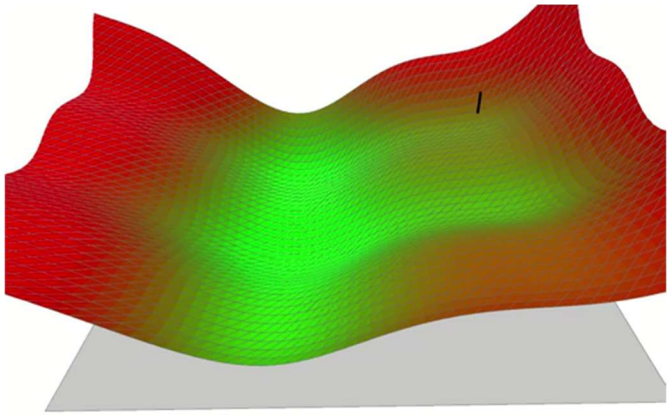
> *Gradient Descent* is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems.

> The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function.

Gradient Descent



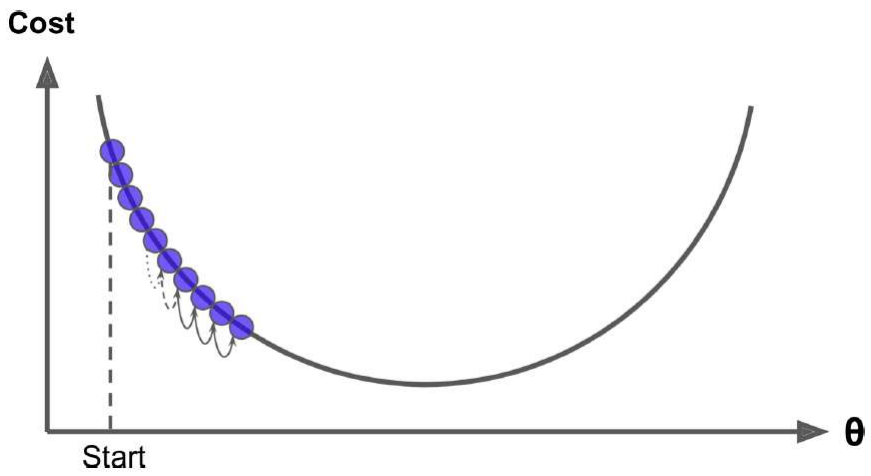
Gradient Descent



Gradient Descent

- An important parameter in Gradient Descent is the size of the steps, determined by the *learning rate* hyperparameter.
- If the learning rate is too small
  - The algorithm will have to go through many iterations to converge
  - It takes a long time

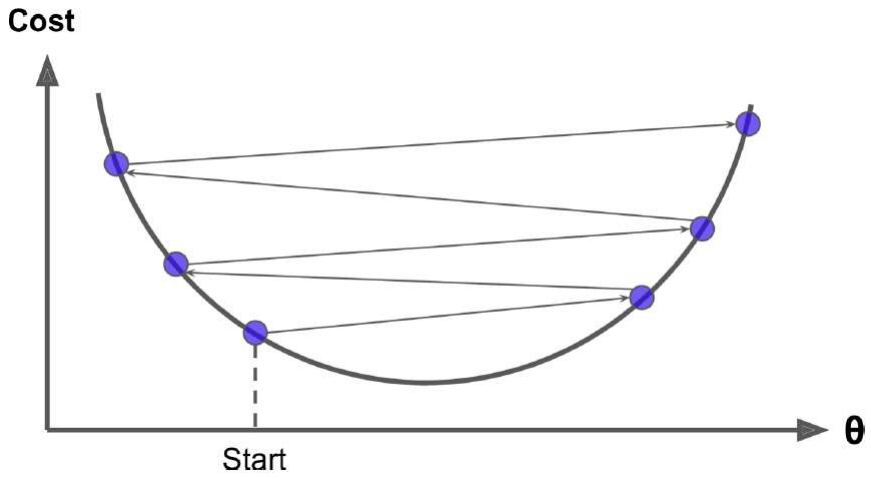
Gradient Descent



Gradient Descent

> If the learning rate is too high
  – It makes the algorithm diverge, with larger and larger values
  – It fails to find a good solution

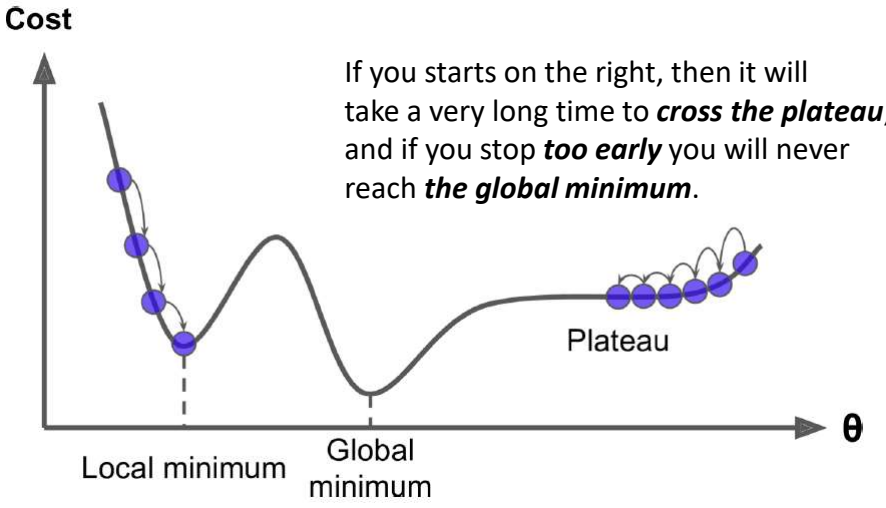
Gradient Descent



Gradient Descent

> Cost functions are not regular
  – there may be holes, ridges, plateaus, and all sorts of irregular terrains
  – makes convergence to the minimum very difficult

Gradient Descent pitfalls



Gradient Descent

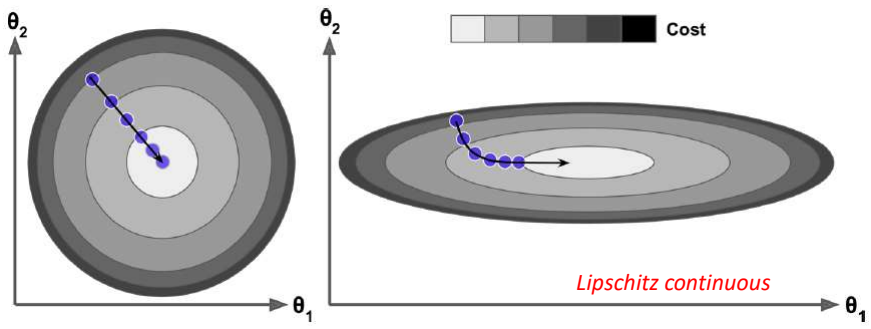
> Good news
  - MSE cost function for a Linear Regression model happens to be a *convex function*
  - if you pick any two points on the curve, the line segment joining them never crosses the curve
  - there are no local minima, just one global minimum
  - It is also a continuous function with a slope that never changes abruptly

Gradient Descent

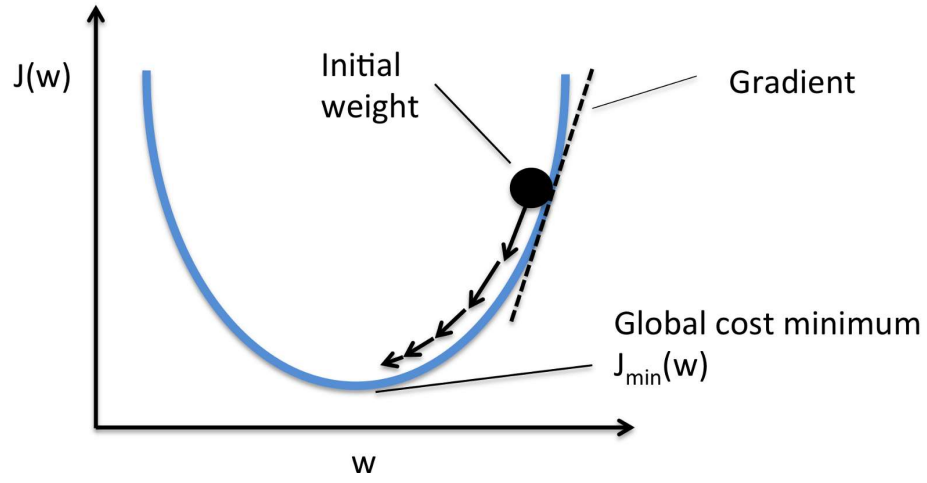
> These two facts have a great consequence

– Gradient Descent is guaranteed to approach arbitrarily close the global minimum

- if you wait long enough
- if the learning rate is not too high



Gradient Descent pitfalls



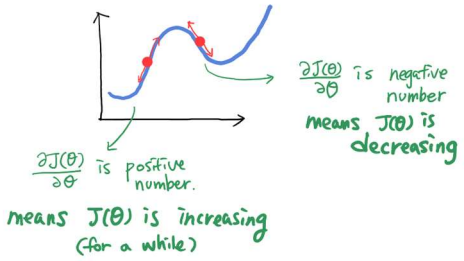
Derivative $loss = (\hat{y} - y)^2 = (x * w - y)^2$

Gradient Descent pitfalls

$$\theta := \theta - \alpha \cdot \frac{\partial J(\theta)}{\partial \theta}$$

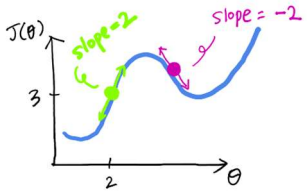
WHY IS THIS MINUS ?

BECAUSE $\frac{\partial J(\theta)}{\partial \theta}$ IS THE DIRECTION OF CHANGE.



We try to MINIMIZE **J(θ)**

Gradient Descent pitfalls



if you set $\theta = \theta - \alpha \cdot \text{slope}$

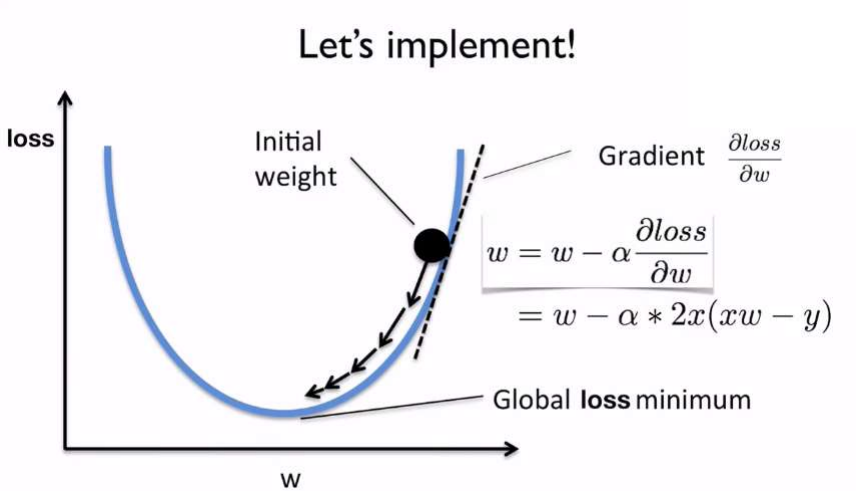
at above points, J(θ) will DECREASE.

if you set $\theta = \theta + \alpha \cdot \text{slope}$

J(θ) will INCREASE.

We try to MINIMIZE **J(θ)**

Gradient Descent pitfalls



Go: https://www.derivative-calculator.net/

Write: (wx-y)^2 for gradiend d/dw

Batch Gradient Descent

- To implement Gradient Descent, you need to compute the gradient of the cost function with regards to each model parameter θ_j.
- You need to calculate how much the cost function will change if you change θ_j just a little bit
- This is called a *partial derivative*.
- It is like asking
  - "what is the slope of the mountain under my feet if I face east?"
  - then asking the same question facing north

$$\frac{\partial}{\partial \theta_j} \mathrm{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^{m} \left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

Batch Gradient Descent

> Gradient vector of the cost function

$$\nabla_\theta \mathrm{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0}\mathrm{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1}\mathrm{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n}\mathrm{MSE}(\theta) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

> Notice that this formula involves calculations over the full training set **X**, at each Gradient Descent step!

> This is why the algorithm is called *Batch Gradient Descent*: it uses the whole batch of training data at every step.

Batch Gradient Descent

> Gradient Descent **is terribly slow** on very **large training sets**

> However, Gradient Descent scales well with the number of features

> Training a Linear Regression model with hundreds of thousands of features

– Gradient Descent is ***much faster*** than using the Normal Equation

Gradient Descent step

> Once you have the gradient vector, just go in the opposite direction

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta \text{MSE}(\theta)$$

Gradient Descent

> Let's look at a quick implementation of this algorithm

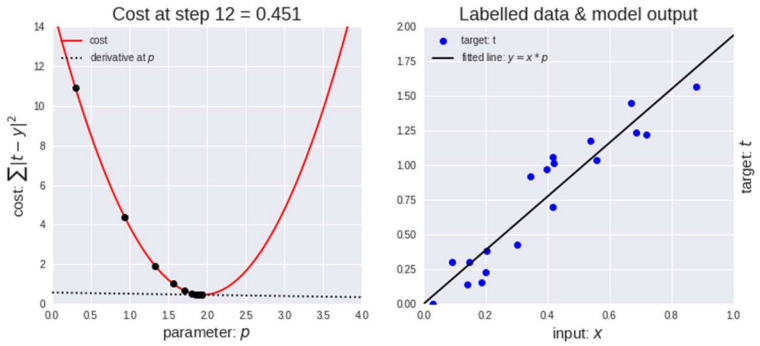
```
eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

> Let's look at the resulting theta:

```
>>> theta
array([[ 4.21509616],
       [ 2.77011339]])
```

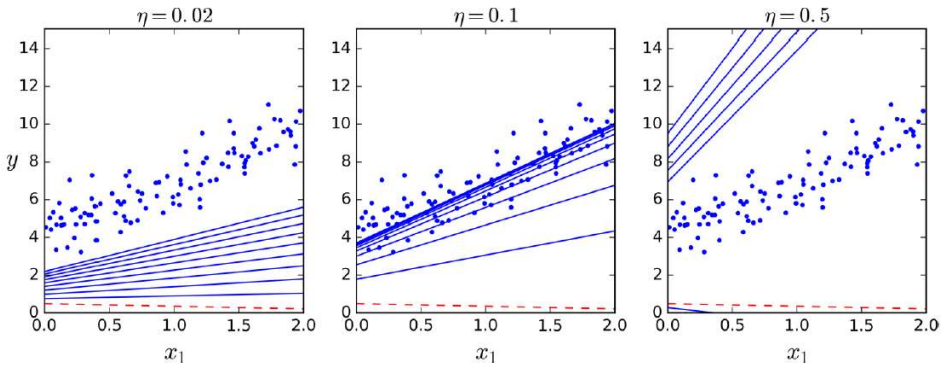
Gradient Descent

> Gradient Descent Minimization and Model Output



Gradient Descent

> Gradient Descent with various learning rates



Gradient Descent

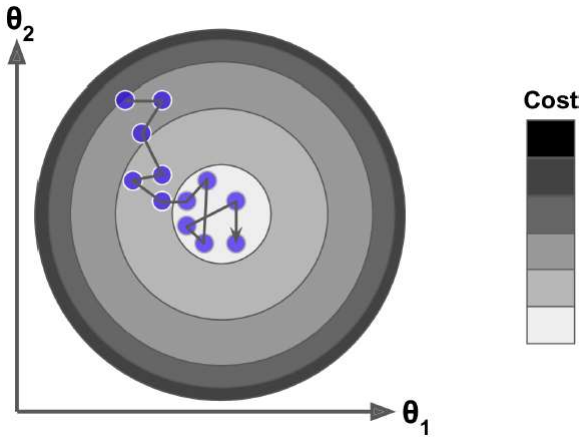
- How to set the number of iterations
  - If it is too low, you will still be far away from the optimal solution when the algorithm stops
  - if it is too high, you will waste time while the model parameters do not change anymore.
- A simple solution is to set a very large number of iterations but to interrupt the algorithm when the gradient vector becomes tiny
  - its norm becomes smaller than a tiny number ϵ (called the *tolerance*)

Stochastic Gradient Descent

- The main problem with Batch Gradient Descent
  - it uses the whole training set to compute the gradients at every step
  - Makes it very slow when the training set is large
- *Stochastic Gradient Descent* just picks a random instance in the training set at every step and computes the gradients based only on that single instance
  - makes the algorithm much faster since it has very little data to manipulate at every iteration.
  - makes it possible to train on huge training sets
    - only one instance needs to be in memory at each iteration

Stochastic Gradient Descent

> Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does



Simulated Annealing

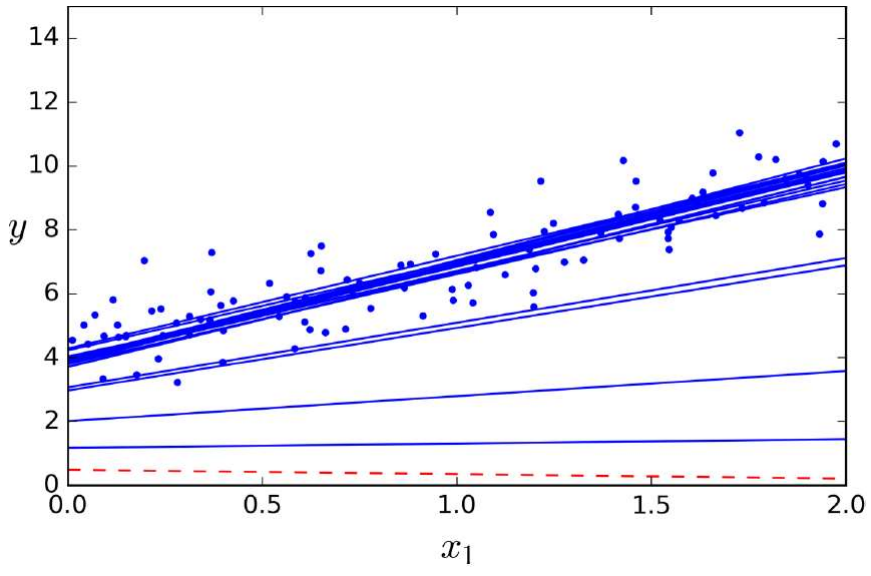
> Gradually reduce the learning rate
> The steps start out large, then get smaller and smaller, allowing the algorithm to settle at the global minimum
> This process is called *simulated annealing*

Implementing Stochastic Gradient Descent

```
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)  # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

```
>>> theta
array([[ 4.21076011],
       [ 2.74856079]])
```

Implementing Stochastic Gradient Descent



SUPPORT VECTOR MACHINES

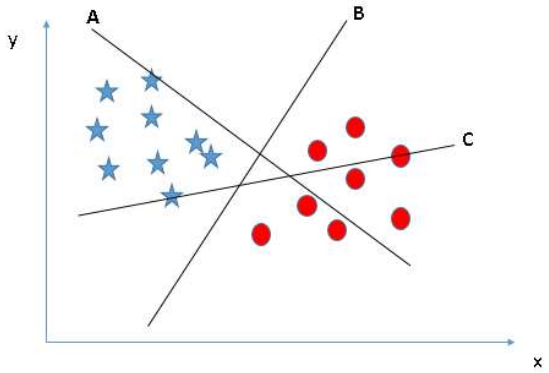
Content

- A *Support Vector Machine* (SVM) is a very powerful and versatile Machine Learning model
  - capable of performing
    - linear or nonlinear
    - classification, regression, and even outlier detection
- It is one of the most popular models in Machine Learning
  - Anyone interested in ML should have it in their toolbox
- SVMs are particularly well suited for classification of complex but small- or medium-sized datasets.
- This module will explain the core concepts of SVMs, how to use them, and how they work.

Linear Classification

> Find possible linear classifiers

– Identify the best hyper-plane: (A?, B? C?)

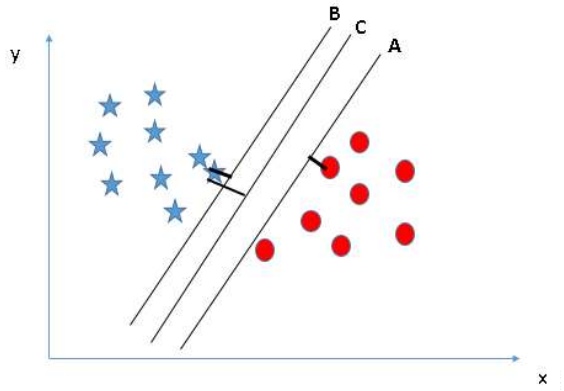


Rule 1: *Select the hyper-plane which separates the two classes better*

Linear Classification

> Find possible linear classifiers

– Identify the best hyper-plane: (A?, B? C?)

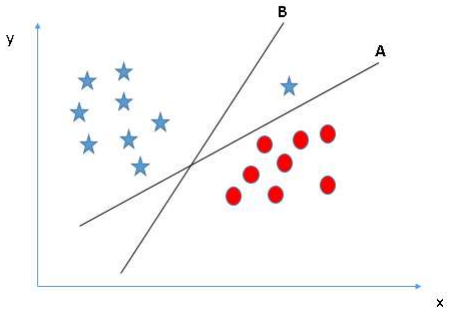


Rule 2: *Maximize the distances between nearest data point and hyper-plane.*

Linear Classification

> Find possible linear classifiers

– Identify the best hyper-plane: (A?, B?)



argin

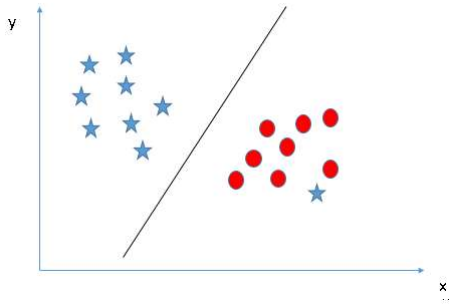
the classes accurately

Rule 3: *Selects the hyper-plane which classifies the classes accurately prior to maximizing margin*

Linear Classification

> Find possible linear classifiers

– Can we classify two classes using straight line?

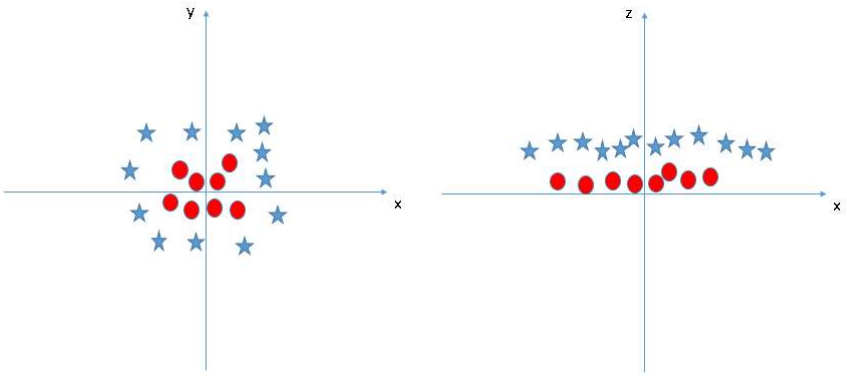


assifier

Rule 4: *Ignore outlier and find the hyper-plane that has maximum margin*

Linear Classification

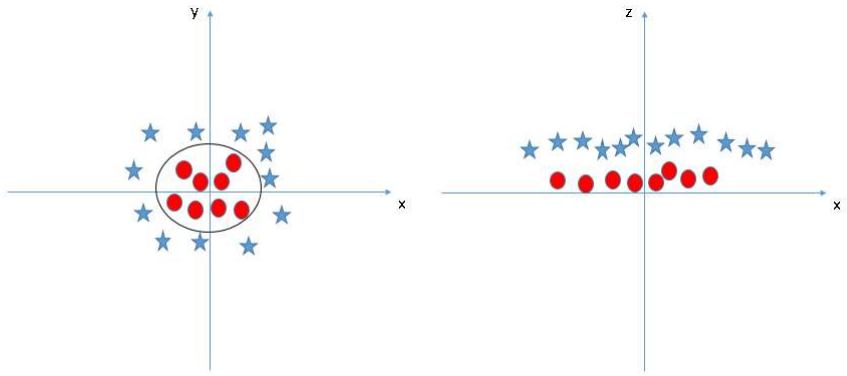
> Find possible linear classifiers
  – Can we classify two classes using straight line? No!



> Add a New Feature: z=x^2+y^2.
  – Now, let's plot the data points on axis x and z

Linear Classification & Kernel Trick

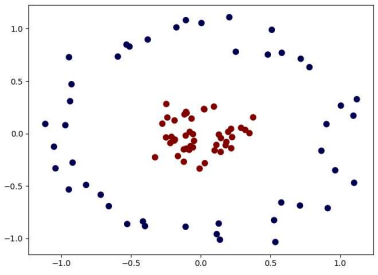
> These are functions which takes low dimensional input space and transform it to

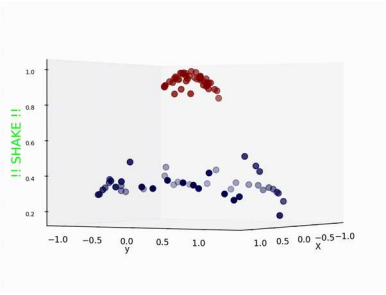


> Add a New Feature: z=x^2+y^2. – KERNEL TRICK !!!
  – It converts not separable problem to separable problem, these functions are called kernels.

Linear Classification & Kernel Trick

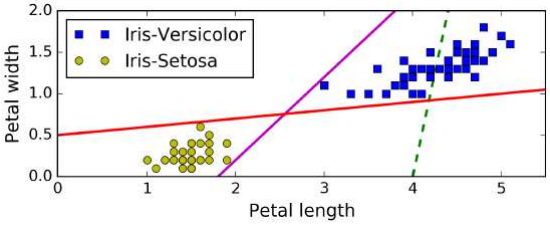
> These are functions which takes low dimensional input space and transform it to

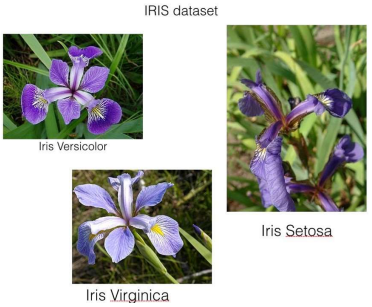




Linear SVM Classification

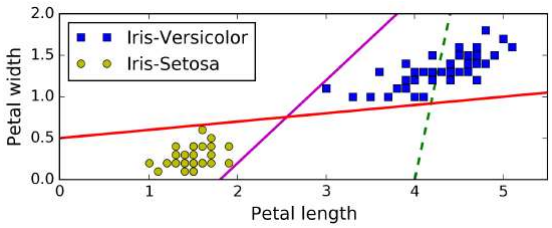
> Three possible linear classifiers



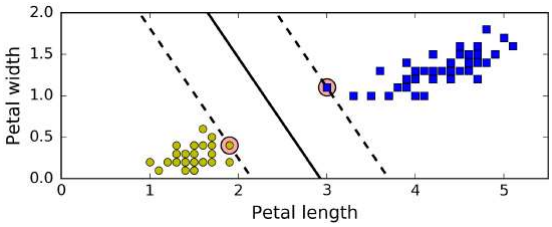


Linear SVM Classification

> Three possible linear classifiers

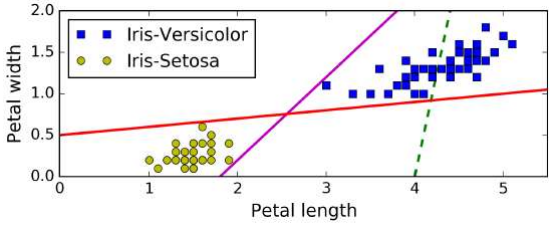


> Large margin classification

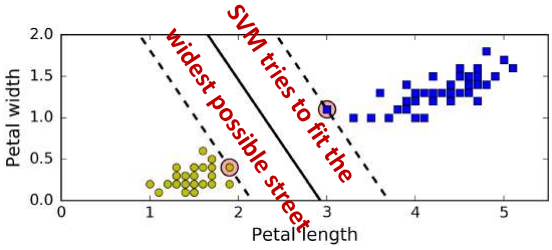


Linear SVM Classification

> Three possible linear classifiers



> Large margin classification

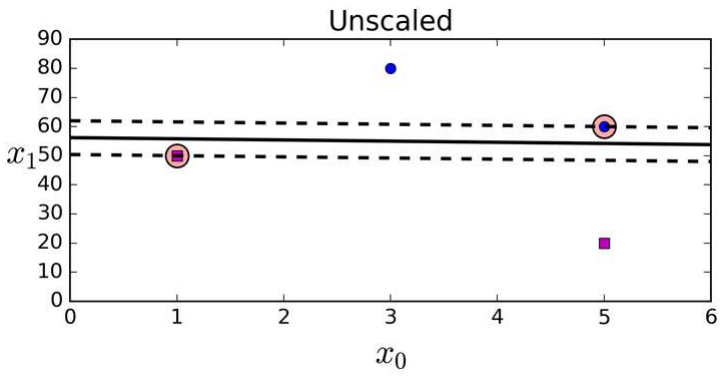


Linear SVM Classification

> Adding more training instances "***off the street***" will not affect the decision boundary at all
  – It is fully determined (or "***supported***") by the instances located on the edge of the street
  – These instances are called the ***support vectors***

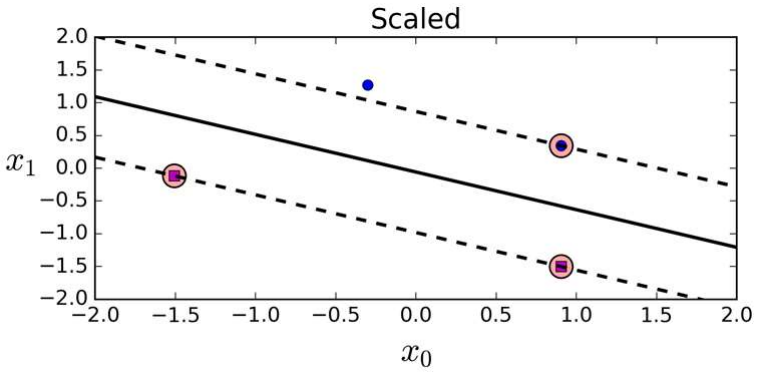
Linear SVM Classification

> SVMs are sensitive to the feature scales



Linear SVM Classification

> SVMs are sensitive to the feature scales



using Scikit-Learn's **`StandardScaler`**

Soft Margin Classification

> If we strictly impose that all instances be off the street and on the right side, this is called ***hard margin classification***.

> There are two main issues with hard margin classification

– It only works if the data is linearly separable

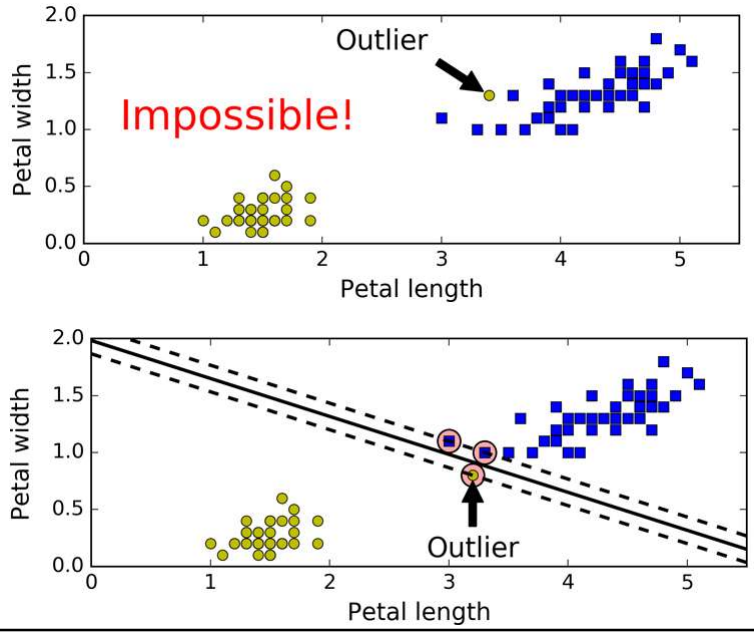
– It is quite sensitive to outliers

– Hard margin given by Boser et al.

– Soft margin given by Vapnik et al.

- Soft margin is extended version of hard margin SVM

Soft Margin Classification



Soft Margin Classification

> To avoid these issues it is preferable to use a more flexible model.

> The objective is to find a good **balance** between

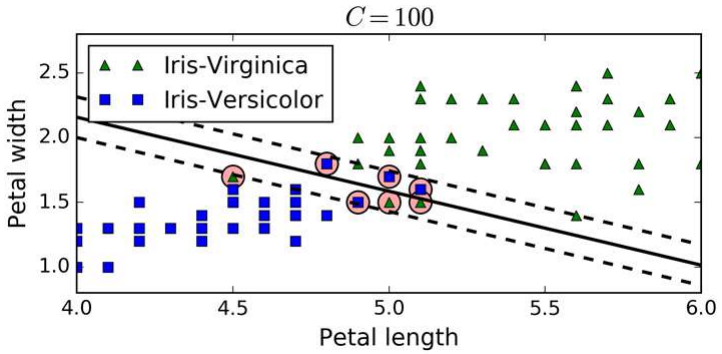
  – keeping the street as large as possible

  – limiting the *margin violations*

> This is called ***soft margin classification***.

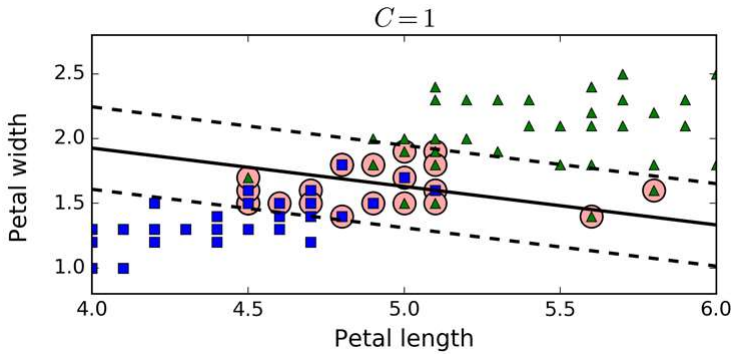
Soft Margin Classification

> In Scikit-Learn's SVM classes, you can control this balance using the C hyperparameter:
  – a smaller C (Cost or Penalty parameter) value leads to a wider street but more margin violations



Soft Margin Classification

> In Scikit-Learn's SVM classes, you can control this balance using the C hyperparameter:
  – a smaller C (Cost or Penalty parameter) value leads to a wider street but more margin violations



Soft Margin Classification

> The following code loads the iris dataset, scales the features, and then trains a linear SVM model

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.float64)  # Iris-Virginica

svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge")),
    ))
svm_clf.fit(X_scaled, y)
```

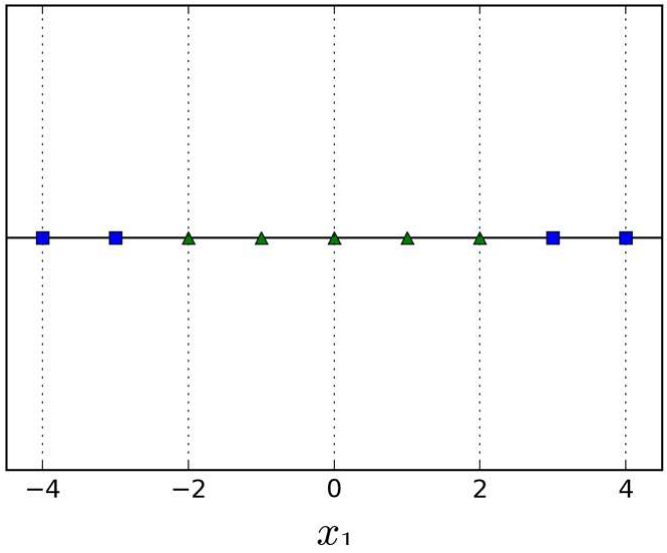
Nonlinear SVM Classification

> Linear SVM classifiers are efficient and work surprisingly well in many cases

> But many datasets are not even close to being linearly separable

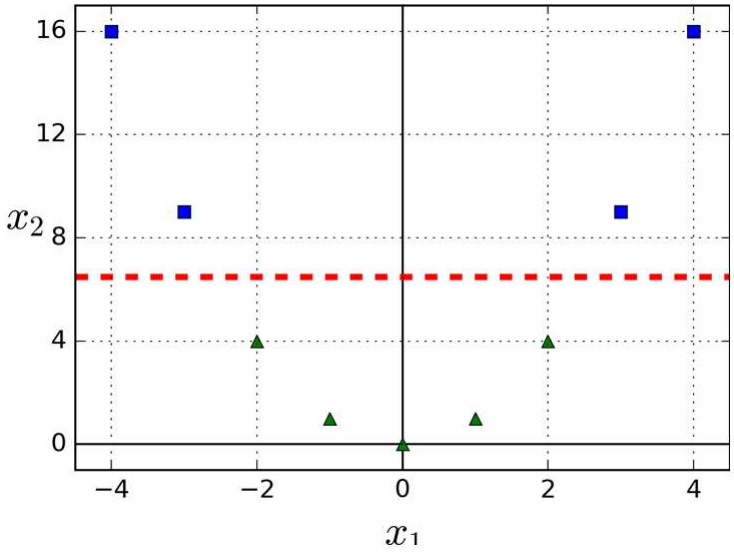
– One approach to handling nonlinear datasets is to add more features, such as polynomial features

Nonlinear SVM Classification



Nonlinear SVM Classification

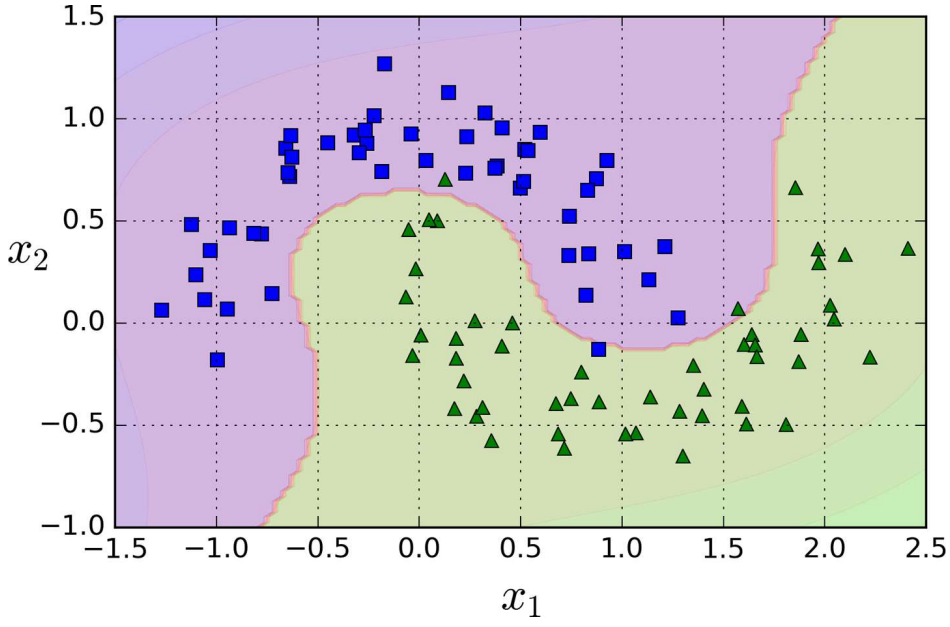
Adding features to make a dataset linearly separable



Linear SVM classifier using polynomial features

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline((
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge"))
    ))

polynomial_svm_clf.fit(X, y)
```

Linear SVM classifier using polynomial features



Polynomial Kernel

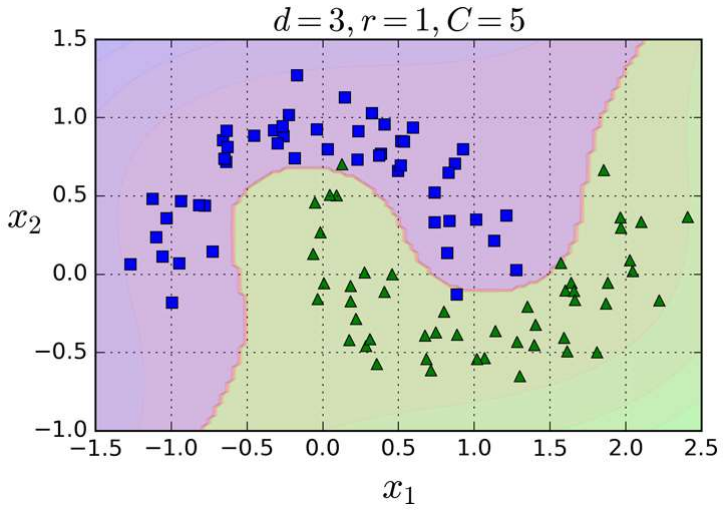
- Adding polynomial features
  - Simple to implement
  - Can work great with all sorts of Machine Learning algorithms (not just SVMs)
- A low polynomial degree it cannot deal with very complex datasets
- A high polynomial degree it creates a huge number of features
  - makes the model too slow.

Kernel Trick

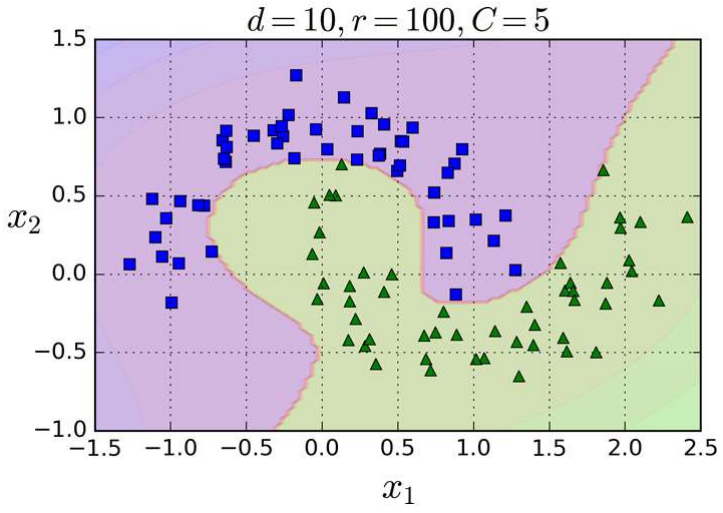
- Kernel Trick
  - it possible to get the same result as if you added many polynomial features
  - There is no combinatorial explosion of the number of features since you don't actually add any features

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ))
poly_kernel_svm_clf.fit(X, y)
```

Kernel Trick



Kernel Trick



Adding Similarity Features

> Another technique to tackle nonlinear problems is to add features computed using a *similarity function*

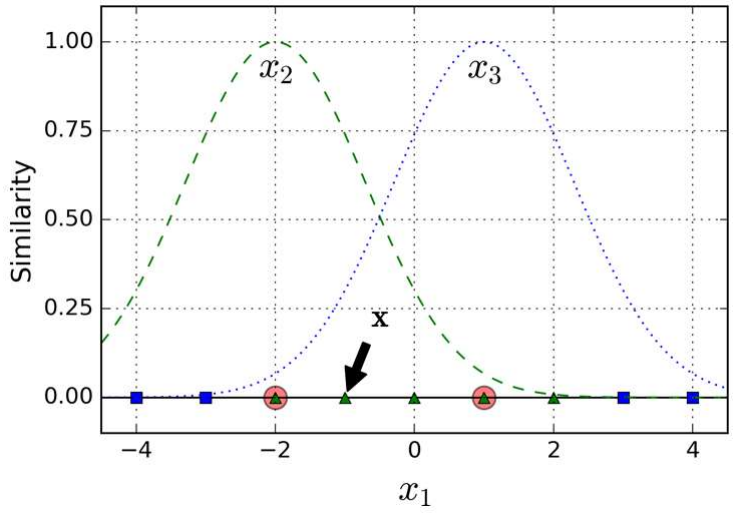
– It measures how much each instance resembles a particular landmark

– Example:

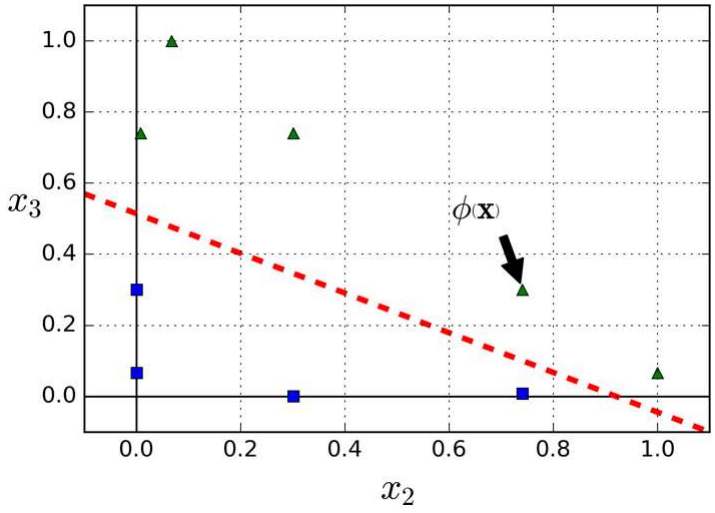
Gaussian Radial Basis Function (RBF) with $\gamma = 0.3$

$$\phi\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$

Similarity features using the Gaussian RBF



Similarity features using the Gaussian RBF



Adding Similarity Features

- You may wonder how to select the landmarks.
  - The simplest approach is to create a landmark at the location of each and every instance in the dataset.
  - This creates many dimensions and thus increases the chances that the transformed training set will be linearly separable.
- The downside is that a training set with *m* instances and *n* features gets transformed into a training set with *m* instances and *m* features
  - **If your training set is very large**, you end up with an equally large number of features.

Gaussian RBF Kernel

> Just like the polynomial features method, the similarity features method can be useful with any Machine Learning algorithm

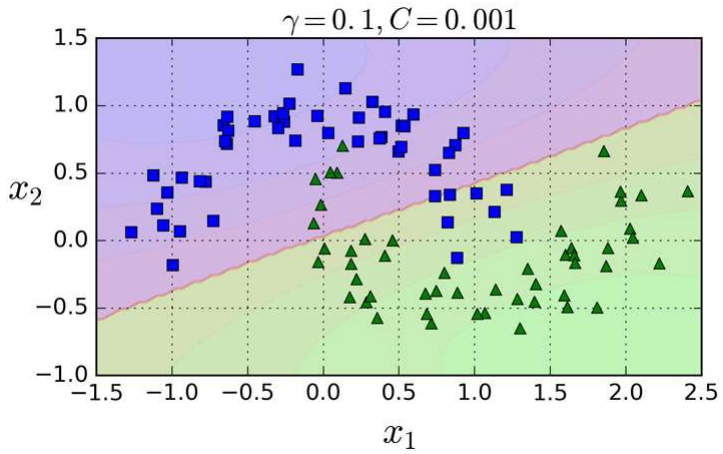
> But it may be computationally expensive to compute all the additional features, especially on large training sets

> However, once again the kernel trick does its SVM magic:

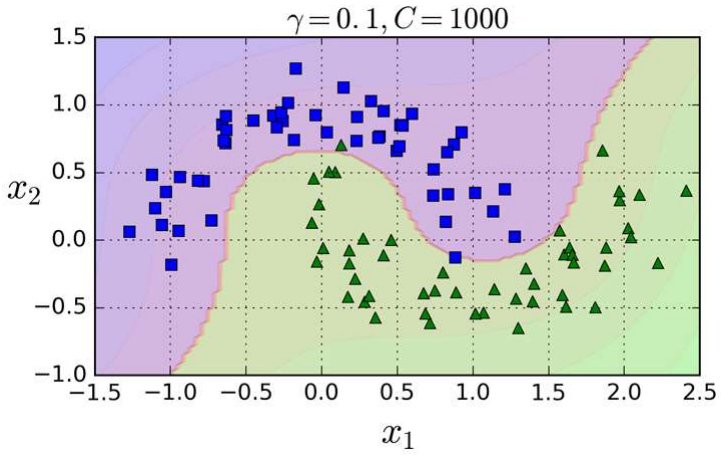
– it makes it possible to obtain a similar result as if you had added many similarity features without actually having to add them.

```
rbf_kernel_svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ))
rbf_kernel_svm_clf.fit(X, y)
```

Gaussian RBF Kernel

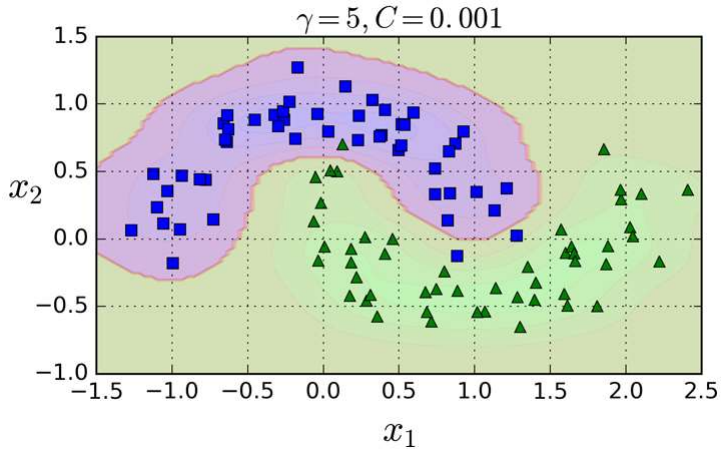


Gaussian RBF Kernel

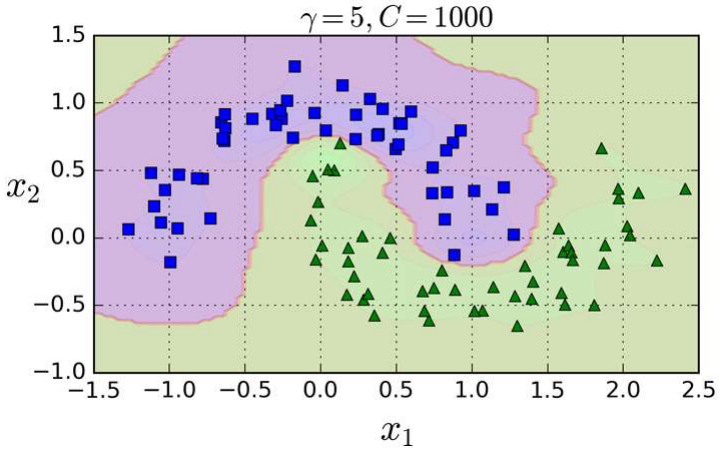


Increasing gamma makes the bell-shape curve narrower

Gaussian RBF Kernel



Gaussian RBF Kernel



- γ acts like a regularization hyperparameter
  - If your model is over-fitting, you should reduce it
  - If it is under-fitting, you should increase it

Computational Complexity

> The LinearSVC class is based on the *liblinear* library

– Uses an optimized algorithm for linear SVMs

"A Dual Coordinate Descent Method for Large-scale Linear SVM," Lin et al. (2008).

– Training time complexity is roughly $O(m \times n)$

– The algorithm takes longer if you require a very high precision

- Controlled by the tolerance hyperparameter ϵ (called `tol` in Scikit-Learn).

Computational Complexity

> The SVC class is based on the *libsvm* library
  - Uses an algorithm that supports the kernel trick
    - "Sequential Minimal Optimization (SMO)," J. Platt (1998)
  - The training time complexity is usually between $O(m^2 \times n)$ and $O(m^3 \times n)$
  - Dreadfully slow when the number of training instances gets large

Computational Complexity

Class	Time Complexity	Out-of-core support	Scaling Required	Kernel Trick
LinearSVC	$O(m \times n)$	No	Yes	No
SGDClassifier	$O(m \times n)$	Yes	Yes	No
SVC	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

SVM Regression

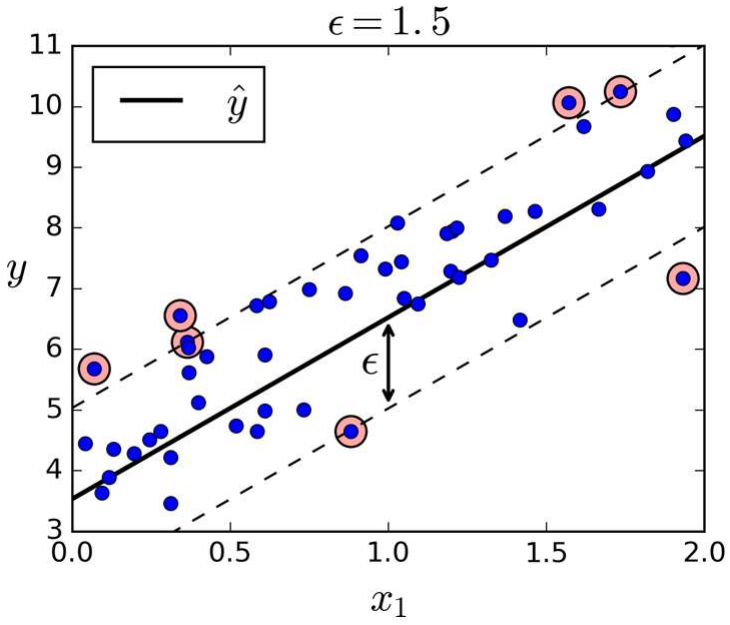
- The SVM algorithm is quite versatile
  - Supports linear and nonlinear classification
  - Supports linear and nonlinear regression
- The trick is to reverse the objective
  - Instead of trying to fit the largest possible street between two classes while limiting margin violations
- SVM Regression tries to fit as many instances as possible *on* the street while limiting margin violations

SVM Regression

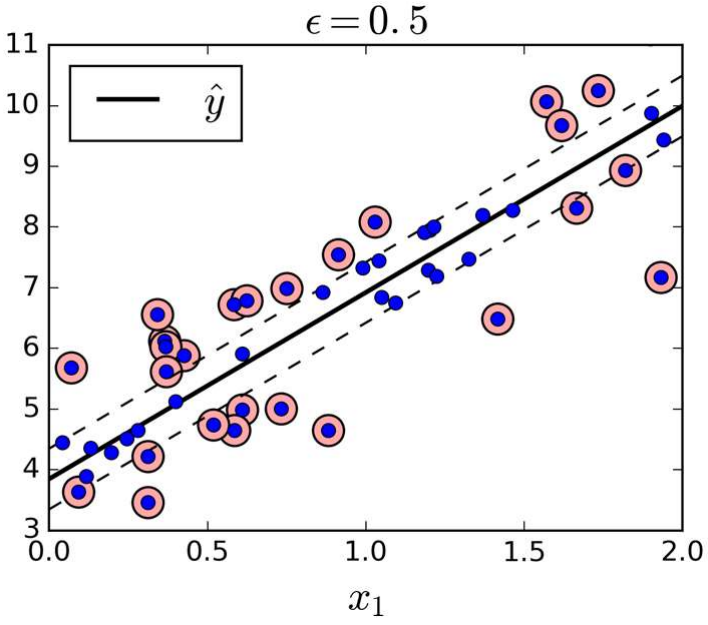
- You can use Scikit-Learn's LinearSVR class to perform linear SVM Regression

```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

SVM Regression



SVM Regression

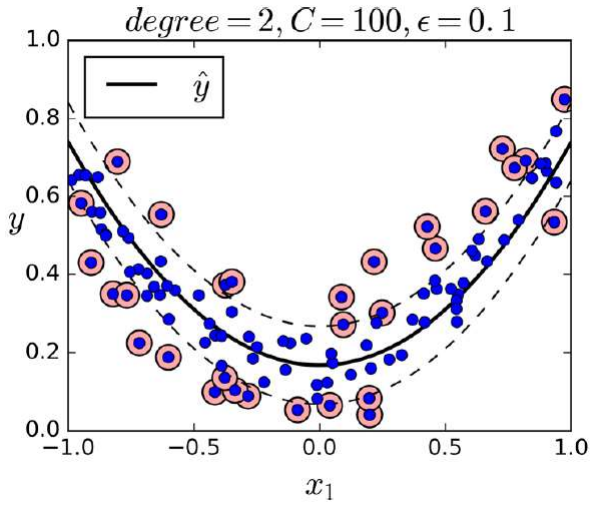


SVM Nonlinear Regression

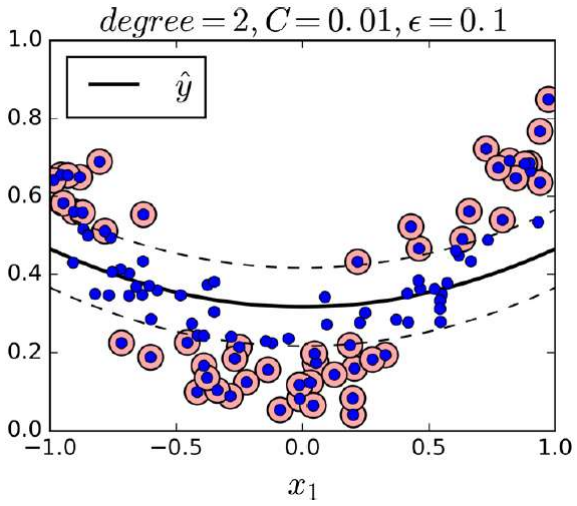
> To tackle nonlinear regression tasks, you can use a kernelized SVM model

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

SVM Nonlinear Regression



SVM Nonlinear Regression



Content

- *Decision Trees* are versatile ML algorithms that can perform both classification and regression tasks
- They are powerful algorithms, **capable of fitting complex datasets**
- Decision Trees are also the fundamental components of Random Forests
  - Random Forests are among the most powerful Machine Learning algorithms available today
- In this module, we will start by discussing how to train, visualize, and make predictions with Decision Trees.
- Then we will go through the CART (Classification and Regression Trees) training algorithm used by Scikit-Learn, and we will discuss how to regularize trees and use them for regression tasks

Decision Tree

- A decision tree is a tree
  - each node represents a feature (attribute)
  - each link (branch) represents a decision (rule)
  - each leaf represents an outcome (categorical or continues value)
- The whole idea is to create a tree like this for the entire data and process a single outcome at every leaf (or minimize the error in every leaf).

How to build Decision Tree

> There are couple of algorithms there to build a decision tree
  - CART (Classification and Regression Trees)
    - uses Gini Index (Classification) as metric
  - ID3 (Iterative Dichotomiser 3)
    - uses ***Entropy function*** and ***Information gain*** as metrics

Classification by Decision Tree

> Decision tree
  - A flow-chart-like tree structure
  - Internal node denotes a test on an attribute
  - Branch represents an outcome of the test
  - Leaf nodes represent class labels or class distribution
> Decision tree generation consists of two phases
  - Tree construction
    - At start, all the training examples are at the root
    - Partition examples recursively based on selected attributes
  - Tree pruning
    - Identify and remove branches that reflect noise or outliers
> Use of decision tree: Classifying an unknown sample
  - Test the attribute values of the sample against the decision tree

Classification by Decision Tree

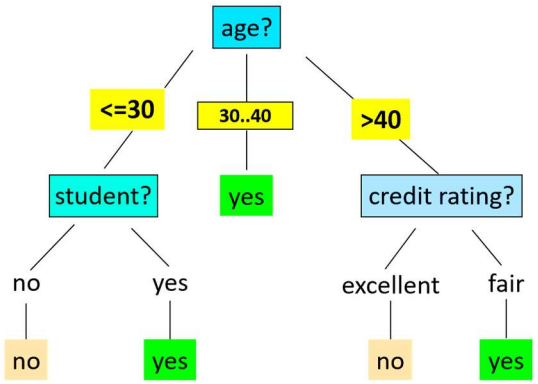
> Sample Training Dataset for Decision Tree

age	income	student	credit_rating	buys_computer
<=30	high	no	fair	no
<=30	high	no	excellent	no
30…40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31…40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31…40	medium	no	excellent	yes
31…40	high	yes	fair	yes
>40	medium	no	excellent	no

Quinlan's ID3 example

Classification by Decision Tree

> Output: A Decision Tree for «buys_computer»



Quinlan's ID3 example

Classification by Decision Tree

> Select the attribute with the highest information gain

> Assume there are **two classes, ...*P*** and ***....N***

– Let the set of examples *S* contain

- ***...p* elements of class *P***
- ***...n* elements of class *N***

– The amount of information, needed to decide if an arbitrary example in *S* belongs to *P* or *N* is defined as

$$I(p,n) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$$

age	income	student	credit_rating	buys_computer
<=30	high	no	fair	no
<=30	high	no	excellent	no
30…40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31…40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31…40	medium	no	excellent	yes
31…40	high	yes	fair	yes
>40	medium	no	excellent	no

Attribute Selection by Information Gain Computation

■I(p, n) = I(9, 5) =0.940

$$Gain(age) = I(p,n) - E(age)$$

$$E(A) = \sum_{i=1}^{v}\frac{p_i + n_i}{p+n} I(p_i, n_i)$$

Gain(age) = 0.940 –0.692 = 0.248

Similarly

$$E(age) = \frac{5}{14} I(2,3) + \frac{4}{14} I(4,0) + \frac{5}{14} I(3,2) = 0.692$$

age	p_i	n_i	$I(p_i, n_i)$
<=30	2	3	0.971
30…40	4	0	0
>40	3	2	0.971

$$Gain(income) = 0.029$$
$$Gain(student) = 0.151$$
$$Gain(credit\_rating) = 0.048$$

Classification by Decision Tree

> Assume that using attribute A

a set *S* will be partitioned into sets $\{S_1, S_2, \ldots, S_v\}$

– If S_i contains p_i examples of *P* and n_i examples of *N* ,the entropy, or the expected information needed to classify objects in all subtrees S_i is

$$E(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p + n} I(p_i, n_i)$$

> The encoding information that would be gained by branching on *A*

Information Gain in Decision Tree Induction

> Assume that using attribute A

a set *S* will be partitioned into sets $\{S_1, S_2, \ldots, S_v\}$

– If S_i contains p_i examples of *P* and n_i examples of *N* ,the entropy, or the expected information needed to classify objects in all subtrees S_i is

$$E(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p + n} I(p_i, n_i)$$

> The encoding information that would be gained by branching on *A*

$$Gain(A) = I(p, n) - E(A)$$

Classification by Decision Tree

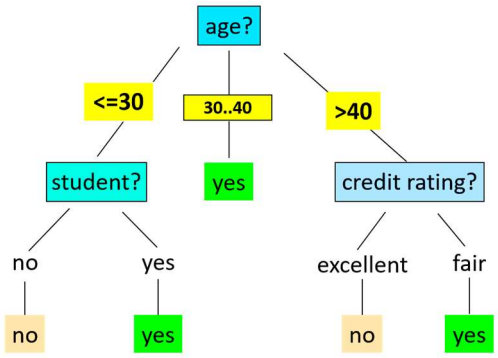
IF *age* = "<=30" AND *student* = "*no*" THEN *buys_computer* = "*no*"

IF *age* = "<=30" AND *student* = "*yes*" THEN *buys_computer* = "*yes*"

IF *age* = "31…40" THEN *buys_computer* = "*yes*"

IF *age* = ">40" AND *credit_rating* = "*excellent*" THEN *buys_computer* = "*yes*"

IF *age* = "<=30" AND *credit_rating* = "*fair*" THEN *buys_computer* = "*no*"



Classification by Decision Tree

> Sample Training Dataset for Decision Tree

age	income	student	credit_rating	buys_computer
<=30	high	no	fair	no
<=30	high	no	excellent	no
30…40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31…40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31…40	medium	no	excellent	yes
31…40	high	yes	fair	yes
>40	medium	no	excellent	no

Quinlan's ID3 example

age	p_i	n_i	$I(p_i, n_i)$
<=30	2	3	0.971
30…40	4	0	0
>40	3	2	0.971

Training and Visualizing a Decision Tree

> To understand Decision Trees, let's just build one and take a look at how it makes predictions

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

Training and Visualizing a Decision Tree

> You can visualize the trained Decision Tree by first using the export_graphviz() method to output a graph definition file called *iris_tree.dot*:

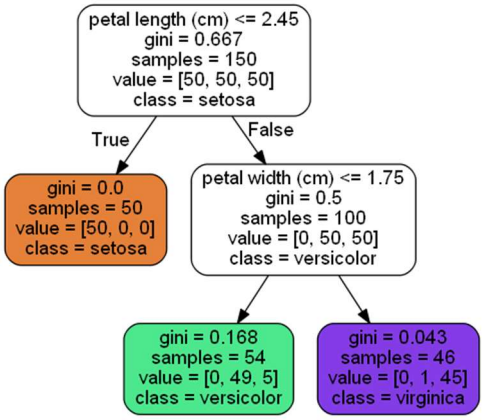
```
from sklearn.tree import export_graphviz

export_graphviz(
        tree_clf,
        out_file=image_path("iris_tree.dot"),
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```

Training and Visualizing a Decision Tree

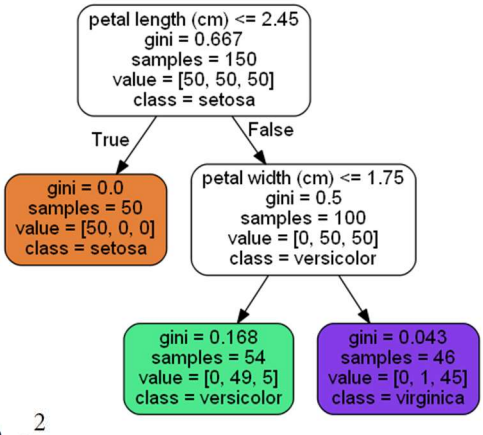
> You can convert this *.dot* file to a variety of formats such as PDF or PNG using the dot command-line tool from the *graphviz* package

```
> dot -Tpng iris_tree.dot -o iris_tree.png
```

Training and Visualizing a Decision Tree



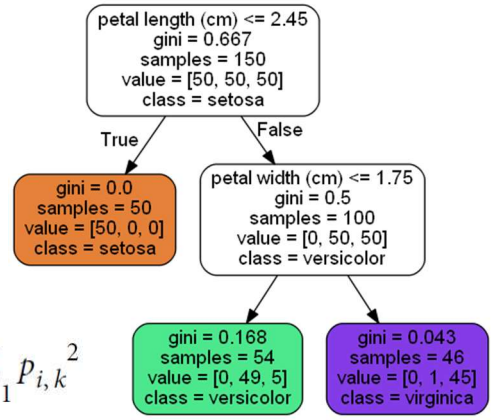
Training and Visualizing a Decision Tree



$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^{\,2}$$

$p_{i,k}$ is the ratio of **class** k instances among the training instances in the i^{th} node

Training and Visualizing a Decision Tree



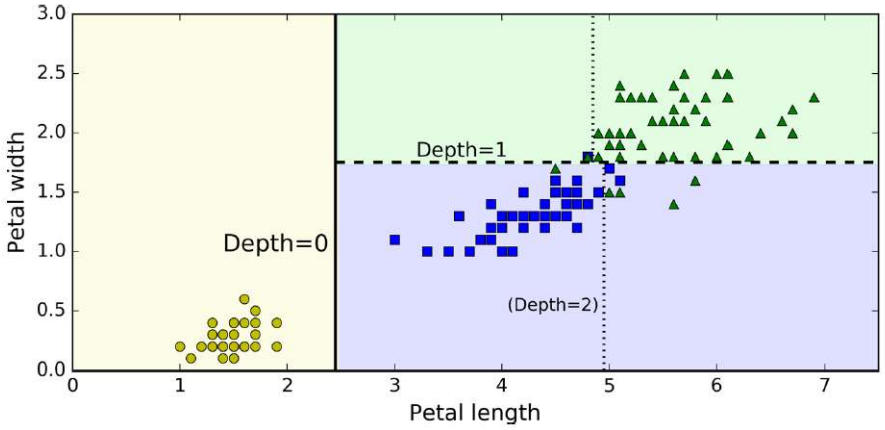
$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^{\,2}$$

$1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$

Decision Tree

> Scikit-Learn uses the CART algorithm
  – produces only *binary trees*: nonleaf nodes always have two children
  – questions only have yes/no answers
> ID3 can produce Decision Trees with nodes that have more than two children.

Decision Tree



Making Predictions

- Suppose you find an iris flower and you want to classify it.
- You start at the *root node* (depth 0, at the top)
  - this node asks whether the flower's petal length is smaller than 2.45 cm.
  - If it is, then you move down to the root's left child node (depth 1, left)
  - In this case, it is a *leaf node:* it does not ask any questions
    - you can simply look at the predicted class for that node and the Decision Tree predicts that your flower is an Iris-Setosa (class=setosa).

Making Predictions

- Now suppose you find another flower
- This time the petal length is greater than 2.45 cm
  - You must move down to the root's right child node (depth 1, right), which is not a leaf node
  - It asks another question: is the petal width smaller than 1.75 cm?
    - If it is, then your flower is most likely an Iris-Versicolor (depth 2, left).
    - If not, it is likely an Iris-Virginica (depth 2, right). It's really that simple.

Decision Tree

- One of the many qualities of Decision Trees
  - They require very little data preparation
  - They don't require feature scaling or centering at all

Model Interpretation: White Box *vs* Black Box

- Decision Trees are fairly intuitive and their decisions are easy to interpret.
  - Such models are often called *white box models*.
- In contrast, Random Forests or neural networks are generally considered *black box models*.
  - They make great predictions, and you can easily check the calculations that they performed to make these predictions
  - Nevertheless, it is usually hard to explain in simple terms why the predictions were made

Estimating Class Probabilities

> A Decision Tree can also estimate the probability that an instance belongs to a particular class *k*
  – first it traverses the tree to find the leaf node for this instance
  – then it returns the ratio of training instances of class *k* in this node.

Estimating Class Probabilities

> For example, suppose you have found a flower whose petals are 5 cm long and 1.5 cm wide.
  – The corresponding leaf node is the depth-2 left node,
  – Decision Tree should output the following probabilities: 0% for Iris-Setosa (0/54), 90.7% for Iris-Versicolor (49/54), and 9.3% for Iris-Virginica (5/54)
  – if you ask it to predict the class, it should output Iris-Versicolor (class 1) since it has the highest probability

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[ 0. ,  0.90740741,  0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

The CART Training Algorithm

> Scikit-Learn uses the *Classification And Regression Tree* (CART) algorithm to train Decision Trees
  – Also called "growing" trees
> The idea is really quite simple
  – The algorithm first splits the training set in two subsets using a single feature k and a threshold t_k (e.g., "petal length ≤ 2.45 cm")
> How does it choose k and t_k?

The CART Training Algorithm

> How does it choose k and t_k?
> It searches for the pair (k, t_k) that produces the purest subsets (weighted by their size).
> The cost function that the algorithm tries to minimize is

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

where $\begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset,} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset.} \end{cases}$

The CART Training Algorithm

- Once it has successfully split the training set in two, it splits the subsets using the same logic, then the sub-subsets and so on, recursively.
- It stops recursing once it reaches the maximum depth (defined by the `max_depth` hyperparameter)
- A few other hyperparameters to control additional stopping conditions
  - `min_samples_split`
  - `min_samples_leaf`
  - `min_weight_fraction_leaf`
  - `max_leaf_nodes`

Computational Complexity

- the CART algorithm is a *greedy algorithm*
  - it greedily searches for an optimum split at the top level, then repeats the process at each level.
  - It does not check whether or not the split will lead to the lowest possible impurity several levels down.
- A greedy algorithm often produces a reasonably good solution, but it is not guaranteed to be the optimal solution.
- Unfortunately, finding the optimal tree is known to be an *NP-Complete* problem
  - It requires $O(\exp(m))$ time
  - The problem intractable problem

Computational Complexity

- Making predictions requires traversing the Decision Tree from the root to a leaf.
- Decision Trees are generally approximately balanced, so traversing the Decision Tree requires going through roughly $O(log_2(m))$ nodes
- Since each node only requires checking the value of one feature, the overall prediction complexity is just $O(log_2(m))$
  - independent of the number of features
- So predictions are very fast, even when dealing with large training sets

Computational Complexity

- However, the training algorithm compares all features (or less if max_features is set) on all samples at each node.
- This results in a training complexity of $O(n \times m\ log(m))$.
  - For small training sets (less than a few thousand instances), Scikit-Learn can speed up training by presorting the data (set presort=True)
  - But this slows down training considerably for larger training sets.

Gini Impurity or Entropy?

> By defualt, Gini impurity measure is used

> But you can select the *entropy* impurity measure instead by setting the **`criterion`** hyperparameter to "entropy".

> The concept of entropy originated in thermodynamics as a measure of molecular disorder:

– entropy approaches zero when molecules are still and well ordered.

– It later spread to a wide variety of domains, including Shannon's *information theory*, where it measures the average information content of a message

– entropy is zero when all messages are identical.

Gini Impurity or Entropy?

> Gini

$$G_i = 1 - \sum_{k=1}^{n} {p_{i,k}}^2$$

> Entropy

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log\left(p_{i,k}\right)$$

Gini Impurity or Entropy?

> Should you use Gini impurity or entropy?

Gini Impurity or Entropy?

> Should you use Gini impurity or entropy?

> The truth is, most of the time it does not make a big difference: they lead to similar trees!

Gini Impurity or Entropy?

> Should you use Gini impurity or entropy?
> The truth is, most of the time it does not make a big difference: they lead to similar trees!
> Gini impurity is slightly faster to compute, so it is a good default

Gini Impurity or Entropy?

> Should you use Gini impurity or entropy?
> The truth is, most of the time it does not make a big difference: they lead to similar trees!
> Gini impurity is slightly faster to compute, so it is a good default
> Gini impurity tends to isolate the most frequent class in its own branch of the tree

Gini Impurity or Entropy?

> Should you use Gini impurity or entropy?

> The truth is, most of the time it does not make a big difference: they lead to similar trees!

> Gini impurity is slightly faster to compute, so it is a good default

> Gini impurity tends to isolate the most frequent class in its own branch of the tree

> Entropy tends to produce slightly more balanced trees.

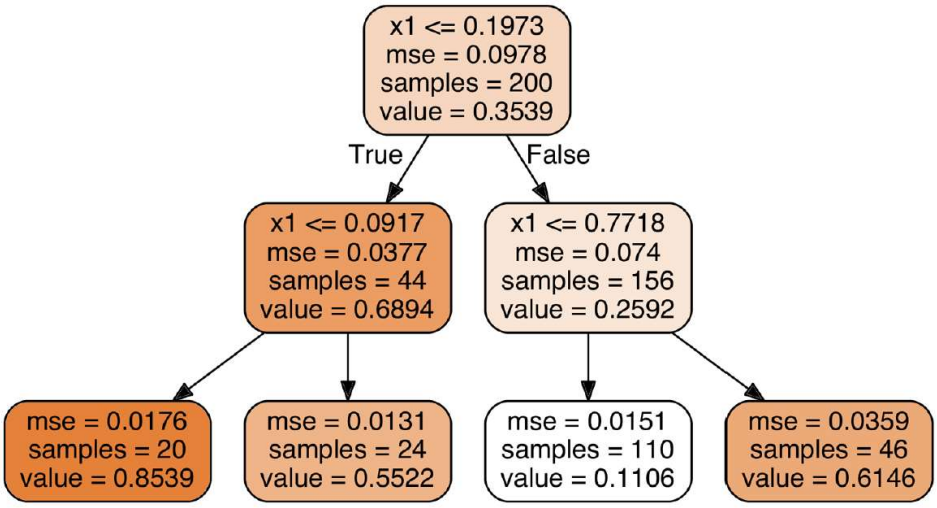
Regression

> Decision Trees are also capable of performing regression tasks.

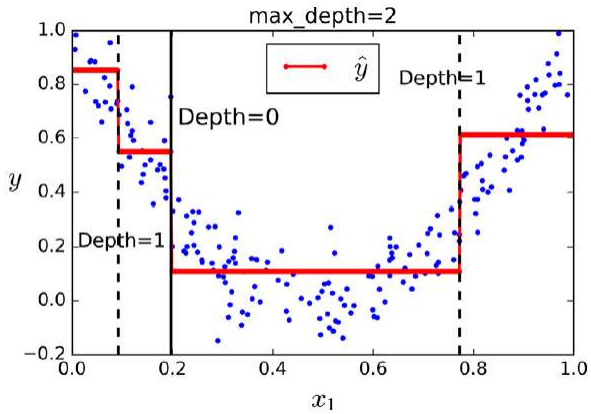
> Let's build a regression tree using Scikit-Learn's DecisionTreeRegressor class, training it on a noisy quadratic dataset with max_depth=2:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

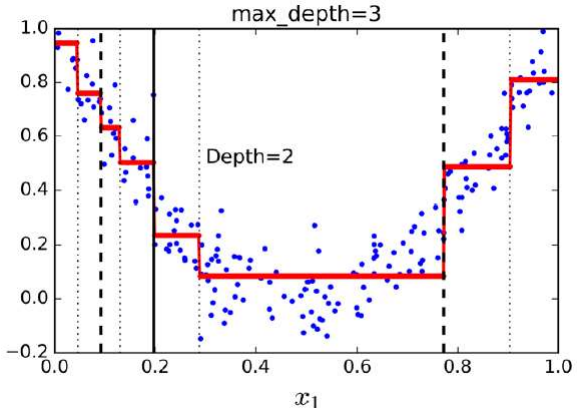
Regression



Regression



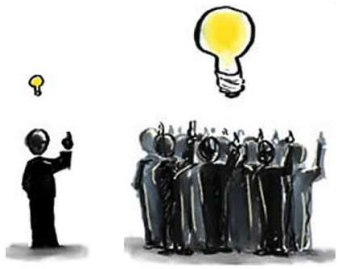
Regression



ENSEMBLE LEARNING AND RANDOM FORESTS

Wisdom of the Crowd

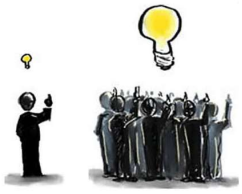
- Suppose you ask a complex question to thousands of random people, then **aggregate their answers**
- In many cases you will find that this aggregated answer is better than an expert's answer. This is called the ***wisdom of the crowd***.



Ensemble

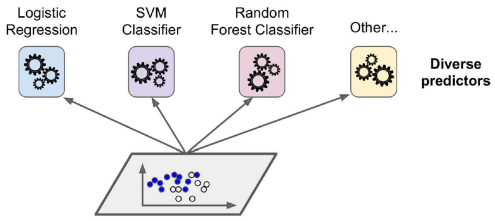
Definition of ENSEMBLE

- a group producing a single effect.
- a group of people or things that make up a complete unit.
- a group of things or people acting or taken together as a whole, especially a group of musicians who regularly play together:
  - Example: The Mozart Ensemble is/are playing at Carnegie Hall tonight.



Ensemble

- Similarly, if you **aggregate the predictions of a group of predictors** (such as classifiers or regressors), you will often get **better predictions** than with the best individual predictor.
- A group of predictors is called an ***ensemble***; thus, this technique is called ***Ensemble Learning***, and an Ensemble Learning algorithm is called an *Ensemble method*.



Ensemble Learning

- You can train a group of Decision Tree classifiers
  - **each on a different random subset of the training set**
- To make predictions, you just obtain the predictions of all individual trees, then **predict the class that gets the most votes**
- Such an ensemble of Decision Trees is called a ***Random Forest***, and despite its simplicity,
  - Random Forest is one of the most powerful Machine Learning algorithms available today

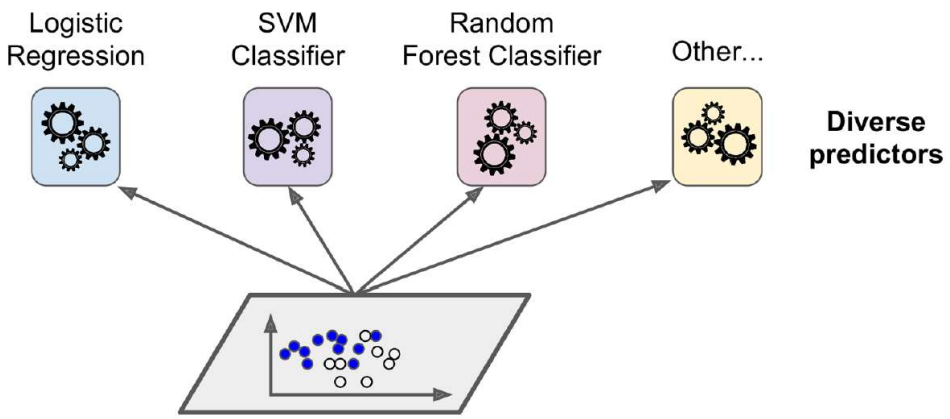
Content

> In this module we will discuss the most popular Ensemble methods
  - ***bagging***
  - ***boosting***
  - ***stacking***
> We will also explore Random Forests

Voting Classifiers

> Suppose you have trained a few classifiers, each one achieving about 80% accuracy.
  - An SVM classifier
  - A Random Forest classifier
  - A Logistic Regression classifier
  - A K-Nearest Neighbors classifier, …

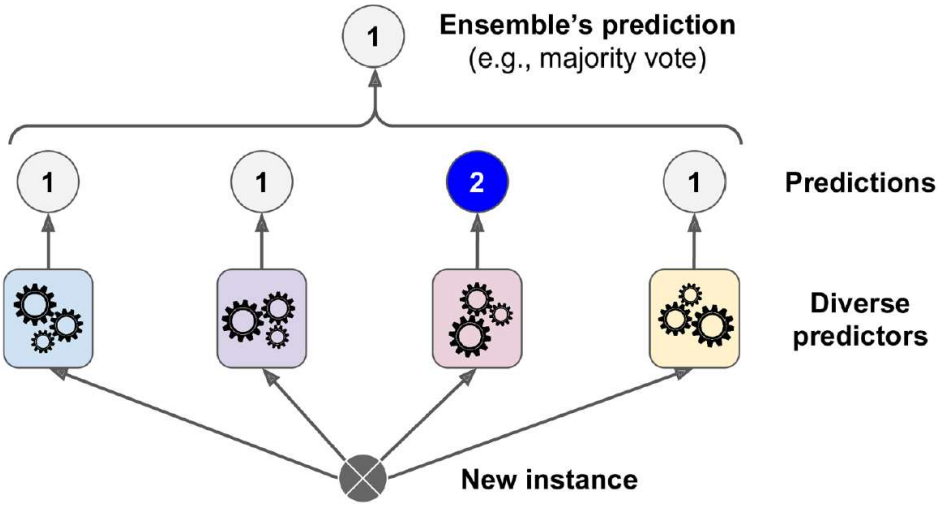
Voting Classifiers



Voting Classifiers

- A very simple way to create an even better classifier is to aggregate the predictions of each classifier and **predict the class that gets the most votes**.
- This ***majority-vote classifier*** is called a ***hard voting*** classifier

Voting Classifiers



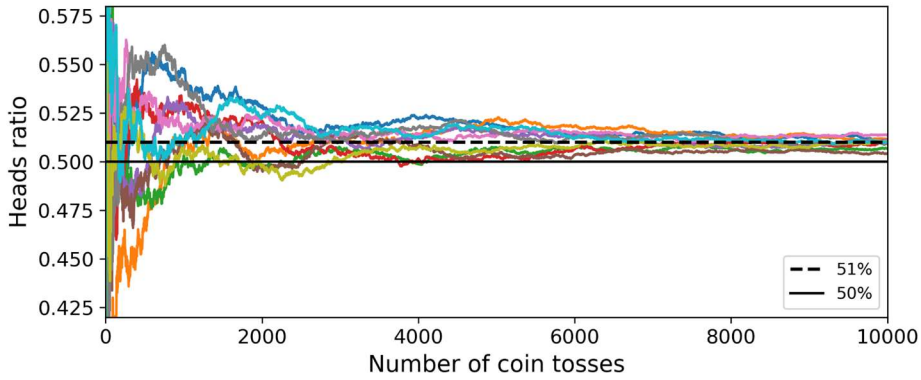
Voting Classifiers

> Somewhat surprisingly, **this voting classifier often achieves a higher accuracy than the best classifier in the ensemble**.

> In fact, even if each classifier is **a *weak learner***, the **ensemble can still be a *strong learner***, provided there are a sufficient number of weak learners and they are sufficiently diverse.

> How is this possible?

The law of large numbers



Voting Classifiers

- Suppose you build an ensemble containing **1,000 classifiers** that are individually **correct only 51%** of the time (barely better than random guessing).
- If you predict the majority voted class, you can hope for up to 75% accuracy!
- However, this is only true
  - if all classifiers are **perfectly independent** making uncorrelated errors
  - Clearly not the case since they are trained on the same data.
  - They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.

Voting Classifiers

- **Ensemble methods work best** when the predictors are as independent from one another as possible.
- One way to get diverse classifiers is to train them using very different algorithms.
- Another approach is to use the same training algorithm for every predictor, but to train them on different random subsets of the training set
- This increases the chance that they will make very different types of errors, improving the ensemble's accuracy.

Voting classifier in Scikit-Learn

- The following code creates and trains a voting classifier in Scikit-Learn, composed of **three diverse classifiers**

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
        estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
        voting='hard'
    )
voting_clf.fit(X_train, y_train)
```

Voting classifier in Scikit-Learn

> Let's look at each classifier's accuracy on the test set

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
>>>     clf.fit(X_train, y_train)
>>>     y_pred = clf.predict(X_test)
>>>     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896
```

Voting classifier in Scikit-Learn

> Let's look at each classifier's accuracy on the test set

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
>>>     clf.fit(X_train, y_train)
>>>     y_pred = clf.predict(X_test)
>>>     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896
```

> **There you have it**!

> The voting classifier **slightly outperforms** all the individual classifiers.

Soft Voting

- If all classifiers are able to estimate **class probabilities**, then you can tell Scikit-Learn to predict the class with the **highest class probability**, **averaged over all the individual classifiers**.
- This is called ***soft voting***
- It often achieves higher performance than hard voting because it gives more weight to highly confident votes.
- All you need to do is replace voting="hard" with voting="soft" and ensure that all classifiers can estimate class probabilities

Soft Voting

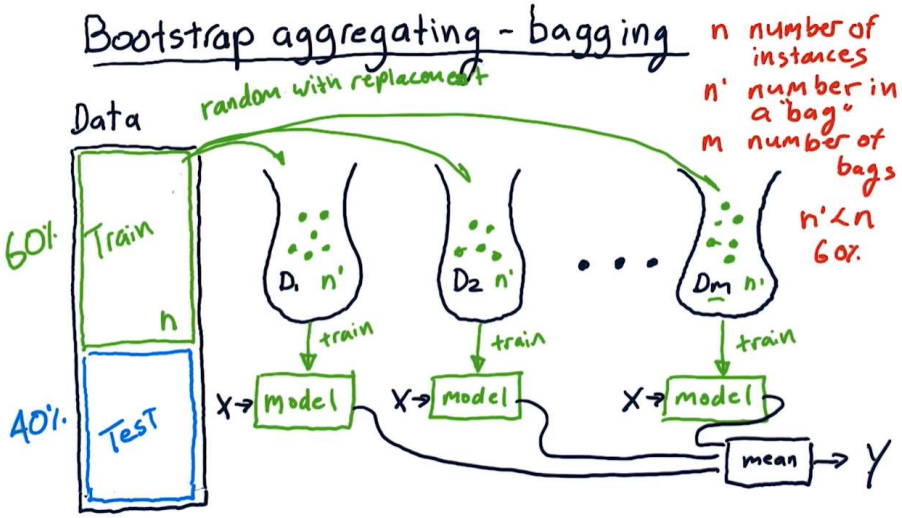
```
In [9]: log_clf = LogisticRegression(solver="liblinear", random_state=42)
        rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
        svm_clf = SVC(gamma="auto", probability=True, random_state=42)

        voting_clf = VotingClassifier(
            estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
            voting='soft')
        voting_clf.fit(X_train, y_train)

Out[9]: VotingClassifier(estimators=[('lr', LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                  intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                  penalty='l2', random_state=42, solver='liblinear', tol=0.0001,
                  verbose=0, warm_start=False)), ('rf', RandomFor...bf',
          max_iter=-1, probability=True, random_state=42, shrinking=True,
          tol=0.001, verbose=False))],
                flatten_transform=None, n_jobs=1, voting='soft', weights=None)
```

Soft Voting

```
In [9]: log_clf = LogisticRegression(solver="liblinear", random_state=42)
        rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
        svm_clf = SVC(gamma="auto", probability=True, random_state=42)

        voting_clf = VotingClassifier(
            estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
            voting='soft')
        voting_clf.fit(X_train, y_train)

Out[9]: VotingClassifier(estimators=[('lr', LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                  intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                  penalty='l2', random_state=42, solver='liblinear', tol=0.0001,
                  verbose=0, warm_start=False)), ('rf', RandomFor...bf',
          max_iter=-1, probability=True, random_state=42, shrinking=True,
          tol=0.001, verbose=False))],
                flatten_transform=None, n_jobs=1, voting='soft', weights=None)
```

```
In [10]: from sklearn.metrics import accuracy_score

         for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
             clf.fit(X_train, y_train)
             y_pred = clf.predict(X_test)
             print(clf.__class__.__name__, accuracy_score(y_test, y_pred))

LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.912
```

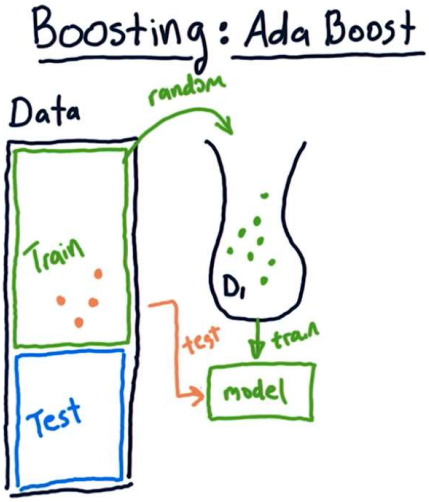
Ensemble - Bagging



BAGGING – divide data into bags with replacement. Same data/observation can be located in the same bags

* Figures form StatQuest

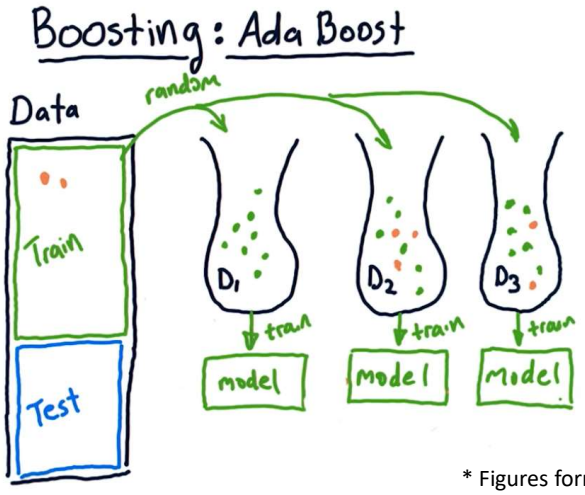
Ensemble - Boosting



* Figures form StatQuest

BOOSTING is a special case of BAGGING.
Select poorly modeled data for bagging. Test each bagged model with training data.

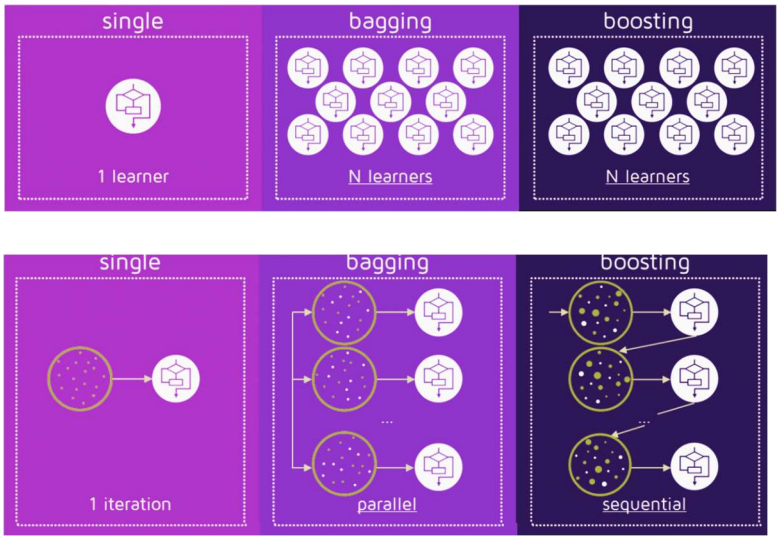
Voting Classifiers



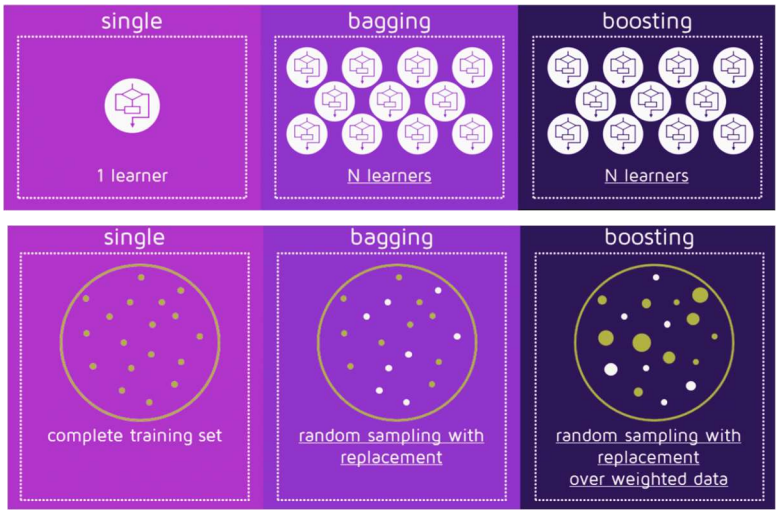
* Figures form StatQuest

Boosting: Select poorly modeled data for bagging.
Test each bagged model with training data.

Bagging / Boosting



Bagging / Boosting



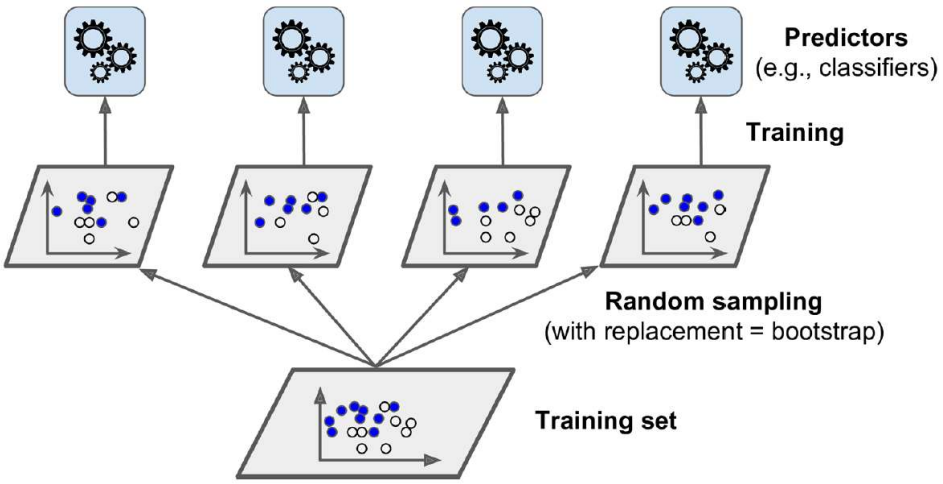
Bagging and Pasting

- One way to get a diverse set of classifiers is to use very different training algorithms
- Another approach is to use the same training algorithm for every predictor, but to train them on different random subsets of the training set
  - When sampling is performed *with* replacement, this method is called ***bagging***
    - short for ***bootstrap aggregating***
  - When sampling is performed *without* replacement, it is called ***pasting***

Bagging and Pasting

- Both bagging and pasting allow training instances to be sampled several times across multiple predictors
- Only bagging allows training instances to be sampled several times for the same predictor

Bagging and Pasting



Bagging and Pasting

- Predictors can all be trained in parallel, **via different CPU cores or even different servers**.
- Similarly, **predictions can be made in parallel**.
- This is one of the reasons why bagging and pasting are such popular methods: **they scale very well**.

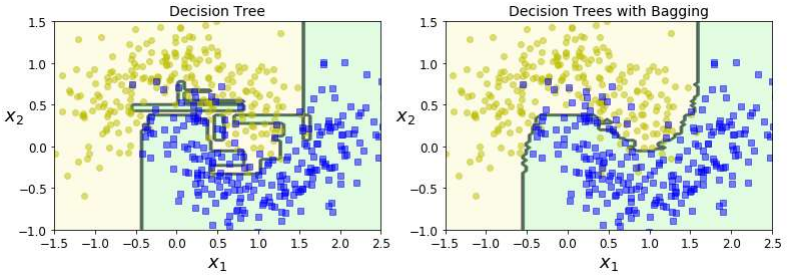
Bagging and Pasting in Scikit-Learn

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
        DecisionTreeClassifier(), n_estimators=500,
        max_samples=100, bootstrap=True, n_jobs=-1
    )
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

Bagging and Pasting in Scikit-Learn

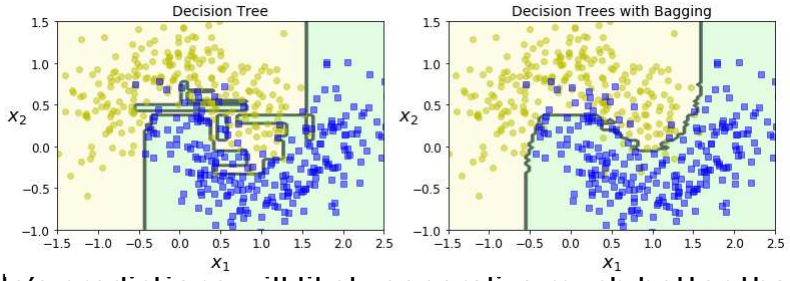
- Scikit-Learn offers a simple API for both bagging and pasting with the **BaggingClassifier** class (or **BaggingRegressor** for regression)
- The code trains an ensemble of 500 Decision Tree classifiers, 5 each trained on 100 training instances randomly sampled from the training set with replacement (this is an example of bagging, but if you want to use pasting instead, just set bootstrap=False).
- The **n_jobs** parameter tells Scikit-Learn **the number of CPU cores** to use for training and predictions **(–1 tells Scikit-Learn to use all available cores**):

Bagging and Pasting in Scikit-Learn



> The ensemble's predictions will likely generalize much better than the single Decision Tree's predictions
  – the decision boundary is less irregular for Ensemble.
> Overall, bagging often results in better models, which explains why it is generally preferred.

Bagging and Pasting in Scikit-Learn



> The ensemble's predictions will likely generalize much better than the single Decision Tree's predictions
  – the decision boundary is less irregular for Ensemble.

> Bagging ends up with a slightly higher bias than pasting, but this also means that predictors end up being less correlated, so the ensemble's variance is reduced.

> Overall, bagging often results in better models, which explains why it is generally preferred.

Out-of-Bag Evaluation

- With bagging,
  - some instances may be sampled several times for any given predictor, while others may not be sampled at all.
- By default a BaggingClassifier samples *m* training instances with replacement (bootstrap=True), where *m* is the size of the training set
- Only about 63% of the training instances are sampled on average for each predictor
- The remaining 37% of the training instances that are not sampled are called ***out-of-bag*** (oob) instances
- Since a predictor never sees the oob instances during training, it can be evaluated on these instances, without the need for a separate validation set or cross-validation

Out-of-Bag Evaluation

- In Scikit-Learn, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic oob evaluation after training

```
>>> bag_clf = BaggingClassifier(
>>>         DecisionTreeClassifier(), n_estimators=500,
>>>         bootstrap=True, n_jobs=-1, oob_score=True)
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.93066666666666664
```

- According to this oob evaluation, this `BaggingClassifier` is likely to achieve about ~93.1% accuracy on the test set

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.93600000000000005
```

Out-of-Bag Evaluation

> The oob decision function for each training instance is also available through the **`oob_decision_function_`** variable.

> In this case (since the base estimator has a pre **`dict_proba()`** method) the decision function returns the class probabilities for each training instance.

```
>>> bag_clf.oob_decision_function_
array([[ 0.        ,  1.        ],
       [ 0.60588235,  0.39411765],
       [ 1.        ,  0.        ],
       ...
       [ 1.        ,  0.        ],
       [ 0.        ,  1.        ],
       [ 0.48958333,  0.51041667]])
```

Random Patches and Random Subspaces

> The **`BaggingClassifier`** class supports sampling the features as well.

> This is controlled by two hyperparameters: **`max_features`** and **`bootstrap_features`**.

> They work the same way as **`max_samples`** and bootstrap, but for feature sampling instead of instance sampling.

> Thus, each predictor will be trained on a random subset of the input features

> This is particularly useful when you are dealing with high-dimensional inputs

> Sampling both training instances and features is called the ***Random Patches*** method

Random Patches and Random Subspaces

- *Random Subspaces* method
  - Keeping all training instances
    - `bootstrap=False` and `max_samples=1.0`
  - Sampling features
    - `bootstrap_features=True` and/or `max_features` smaller than 1.0

Random Forests

- Random Forest is an ensemble of Decision Trees
- Generally trained via the bagging method typically with `max_samples` set to the size of the training set
- Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can instead use the `RandomForestClassifier` class, which is more convenient and optimized for Decision Trees

Random Forests

> The following code trains a Random Forest classifier with 500 trees (each limited to maximum 16 nodes), using all available CPU cores:

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500,
                         max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

Random Forests

> With a few exceptions, a **`RandomForestClassifier`** has all the hyperparameters of a **`DecisionTreeClassifier`** (to control how trees are grown), plus all the hyperparameters of a **`BaggingClassifier`** to control the ensemble itself.

Random Forests

> The Random Forest algorithm introduces extra randomness when growing trees
> Instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features.
> This results in a greater tree diversity, which trades a higher bias for a lower variance, generally yielding an overall better model.

Random Forests

> The following **`BaggingClassifier`** is roughly equivalent to the previous **`RandomForestClassifier`**:

```
bag_clf = BaggingClassifier(
        DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
        n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1
    )
```

Extra-Trees

- When you are growing a tree in a Random Forest, at each node only a random subset of the features is considered for splitting
- It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds
- A forest of such extremely random trees is simply called an ***Extremely Randomized Trees*** ensemble
  - ***Extra-Trees*** for short

Extra-Trees

- You can create an Extra-Trees classifier using Scikit-Learn's `ExtraTreesClassifier` class.
- Its API is identical to the `RandomForestClassifier` class.
- Similarly, the `ExtraTreesRegressor` class has the same API as the `RandomForestRegressor` class.

Extra-Trees

- It is hard to tell in advance whether a `RandomForestClassifier` will perform better or worse than an `ExtraTreesClassifier`.
- Generally, the only way to know is to try both and compare them using cross-validation

Feature Importance

- if you look at a single Decision Tree, important features are likely to appear closer to the root of the tree, while unimportant features will often appear closer to the leaves
- It is therefore possible to get an estimate of a feature's importance by computing the average depth at which it appears across all trees in the forest

Feature Importance

> You can access the result using the **`feature_importances_`** variable

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
>>>     print(name, score)
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

Boosting

> *Boosting* refers to any Ensemble method that can combine several weak learners into a strong learner.

> The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor.

> There are many boosting methods available

> The most popular are

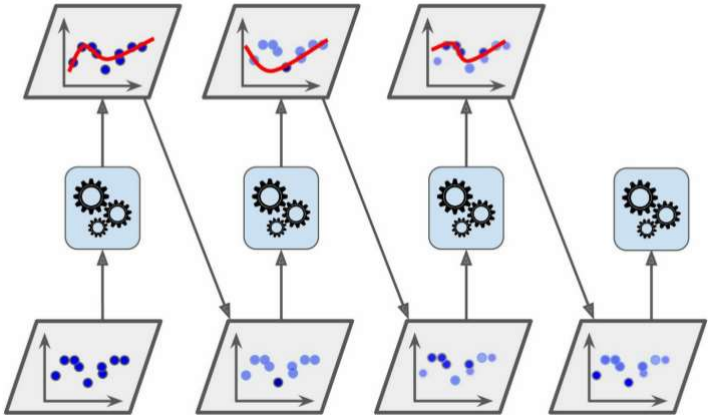
– ***AdaBoost*** (short for *Adaptive Boosting*)

– ***Gradient Boosting***

AdaBoost

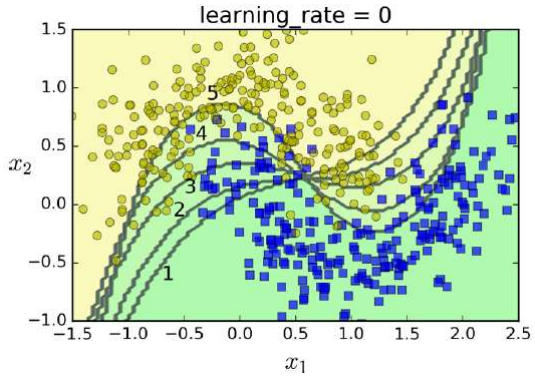
> One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted.
> This results in new predictors focusing more and more on the hard cases
> This is the technique used by AdaBoost.

AdaBoost



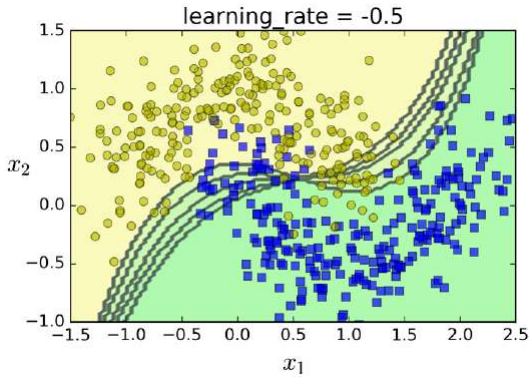
AdaBoost

> *Decision boundaries of consecutive predictors*



AdaBoost

> *Decision boundaries of consecutive predictors*



AdaBoost

> The following code trains an AdaBoost classifier based on 200 *Decision Stumps* using Scikit-Learn's AdaBoostClassifier class

> A Decision Stump is a Decision Tree with max_depth=1

– A tree composed of a single decision node plus two leaf nodes.

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
        DecisionTreeClassifier(max_depth=1), n_estimators=200,
        algorithm="SAMME.R", learning_rate=0.5
    )
ada_clf.fit(X_train, y_train)
```

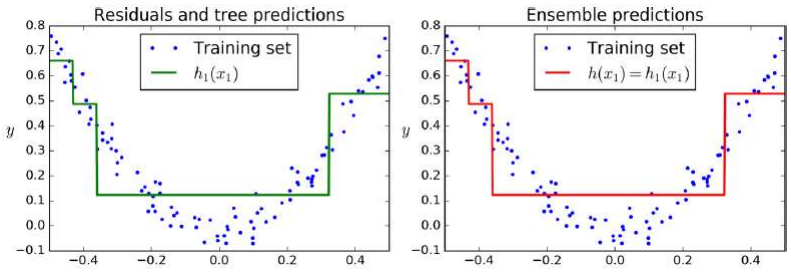
Gradient Boosting

> Another very popular Boosting algorithm is **Gradient Boosting**

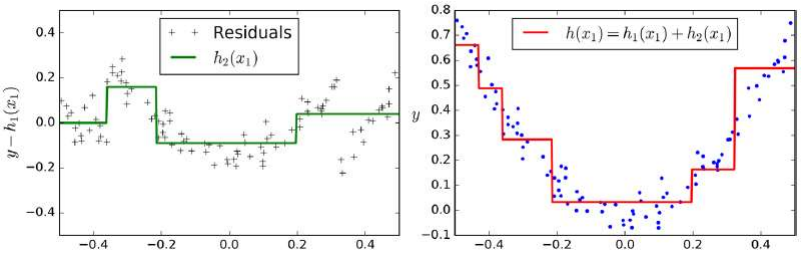
> Just like **AdaBoost**, **Gradient Boosting** works by sequentially adding predictors to an ensemble, each one correcting its predecessor.

> However, instead of tweaking the instance weights at every iteration like **AdaBoost** does, this method tries to fit the new predictor to the ***residual errors*** made by the previous predictor.

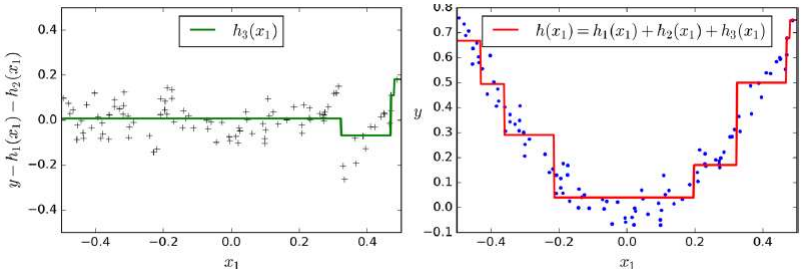
Gradient Boosting



Gradient Boosting

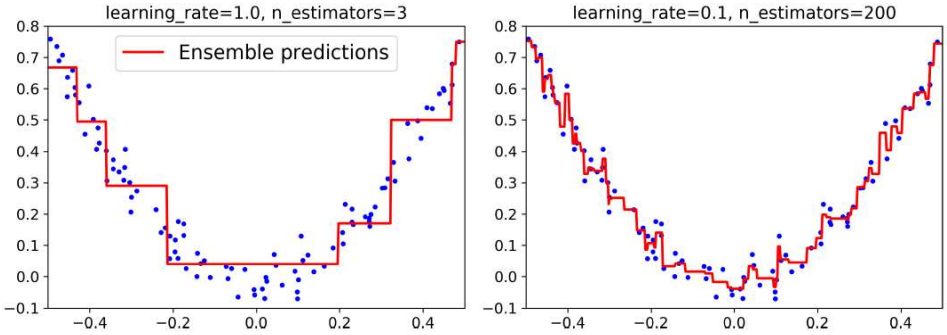


Gradient Boosting



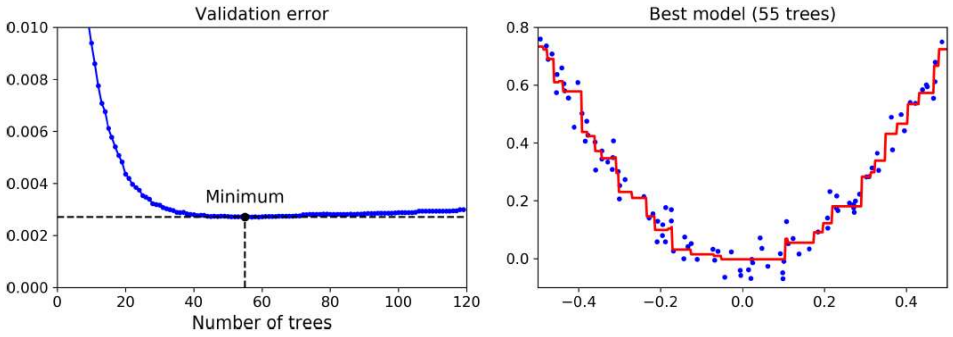
Gradient Boosting

> *GBRT ensembles with not enough predictors (left) and too many (right)*



Gradient Boosting

> *Tuning the number of trees using early stopping*



Gradient Boosting

> It is also possible to implement early stopping by actually stopping training early by setting warm_start=True
  - makes Scikit-Learn keep existing trees when the fit() method is called
  - allows incremental training

Gradient Boosting

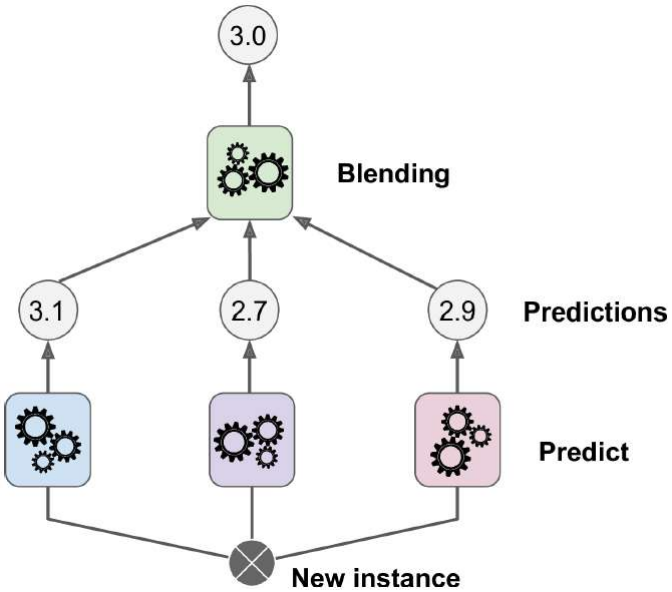
> The following code stops training when the validation error does not improve for five iterations in a row:

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)

min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break  # early stopping
```

Stacking

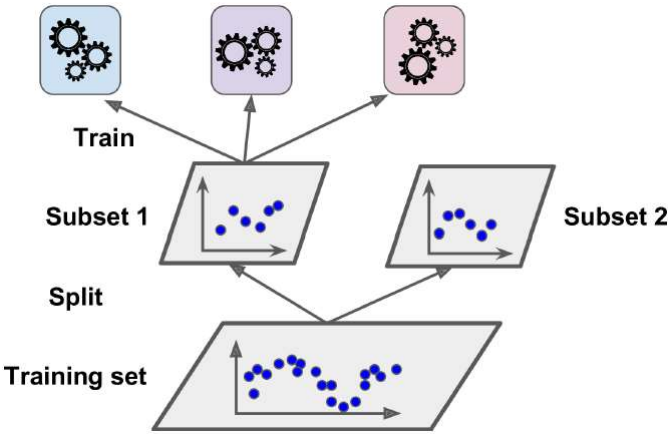
> Instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, ***why don't we train a model to perform this aggregation***?

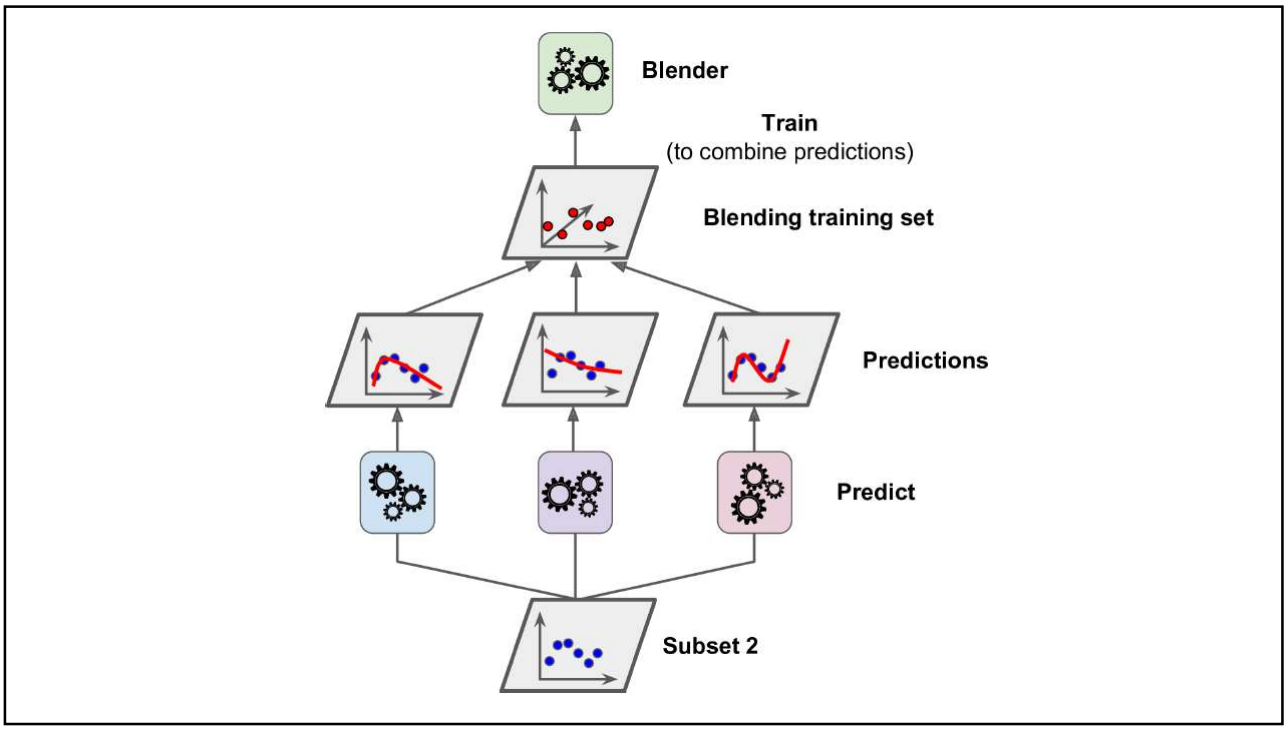
Stacking

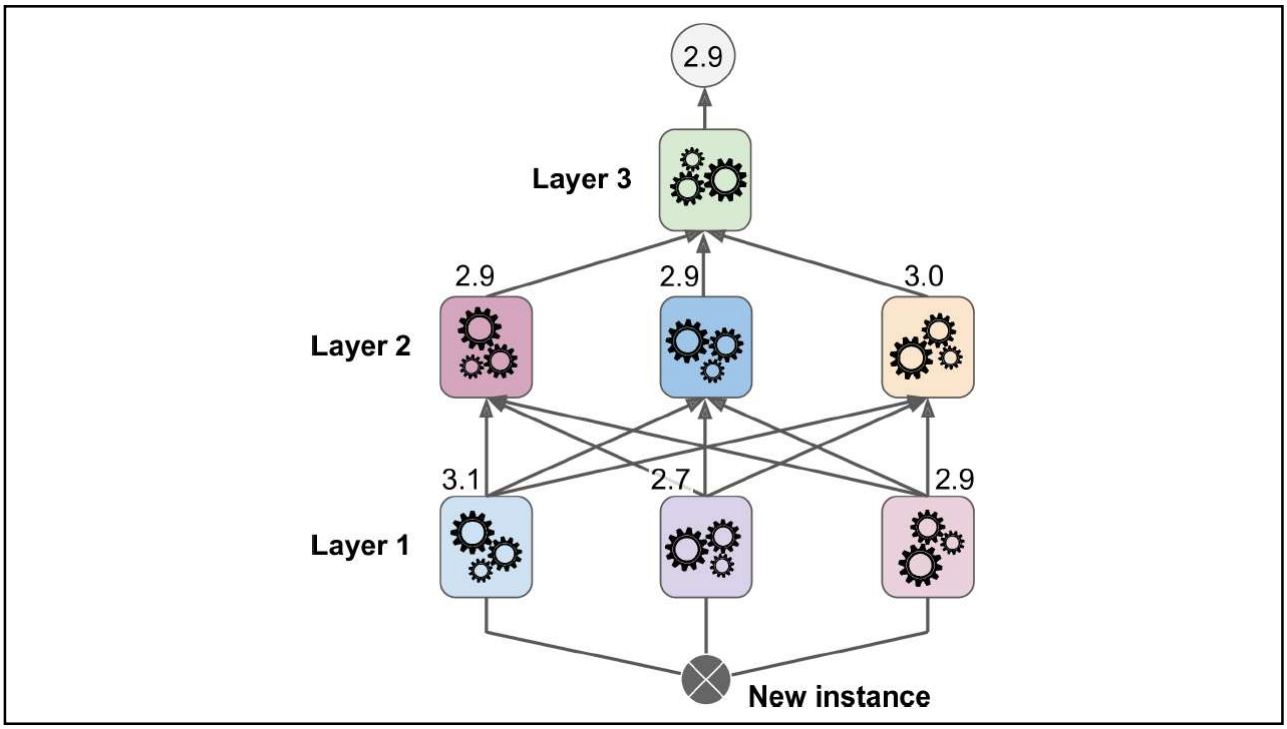


Stacking

> *Training the first layer*




Blender
Train
(to combine predictions)
Blending training set
Predictions
Predict
Subset 2


2.9
Layer 3
2.9
2.9
3.0
Layer 2
3.1
2.7
2.9
Layer 1
New instance