Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)
Systems Software &
Architecture Lab.
Seoul National University

Jan. 6 – 17, 2020

Python for Data Analytics

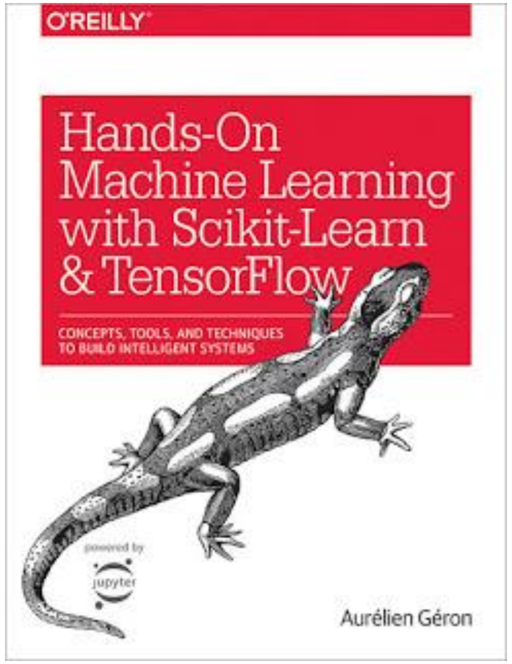
SK-Learn

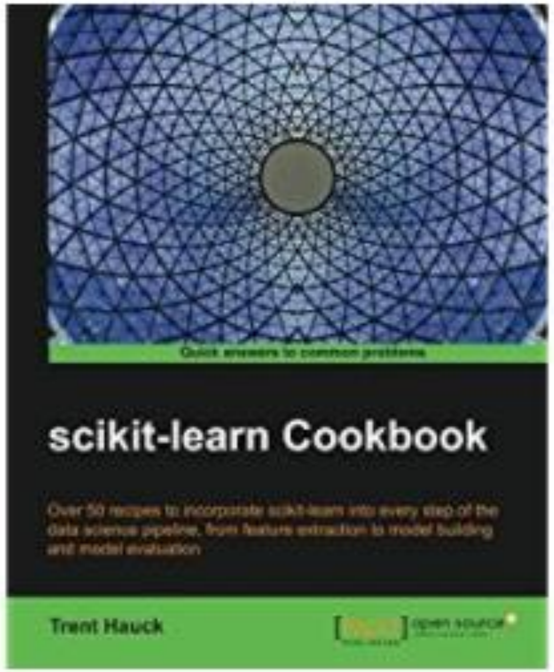
Outline

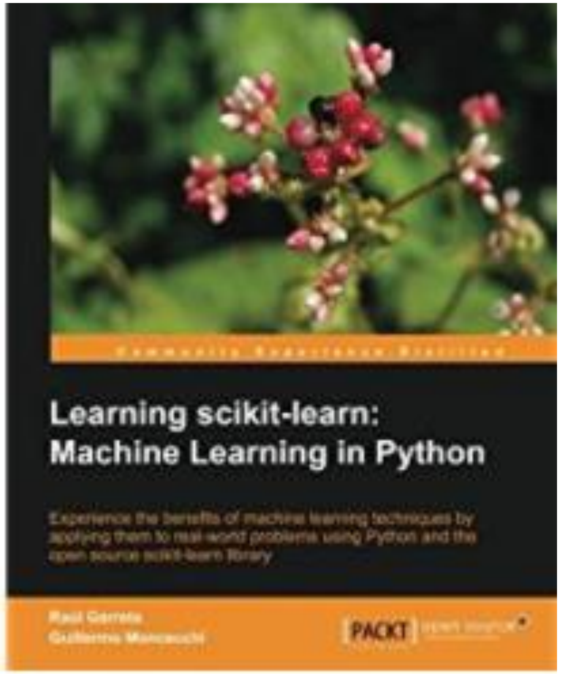
- What is SK-Learn?
- Linear Regression Classifier
- K-Nearest Neighbor (KNN) Classifier
- Decision Tree Classifier
- K-Means Clustering

What is SK-Learn?

Many SK-Learn Books







What is "Sklearn" Module?

- SciKit (SciPy Toolkit)-learn, or SK-Learn
- Open source machine learning library for Python
- Built on top of SciPy
  - Designed to interoperate with Python numerical and scientific library
- Dependency
  - NumPy, SciPy, Matplotlib
- Open source (https://scikit-learn.org)
  - Initially developed by David Cournapeau as a "Google Summer of Code" project in 2007
  - Still under active development (v0.22.1 as of January 2020)

Sklearn Modules

- Classification: identify to which category an object belongs to
  - Regression: predict continuous-valued attribute (linear, logistic, etc.)
  - SVM, Decision tree, Neural nets, Nearest neighbors, …
- Clustering: grouping of similar objects
  - K-means, Hierarchical clustering, etc.
- Model selection: validate and choosing parameters and model
  - Cross validation, metrics, etc.
- Preprocessing: feature extraction & normalization
- Dimensionality reduction: reducing number of variables
  - PCA, Feature selection, etc.
- Datasets

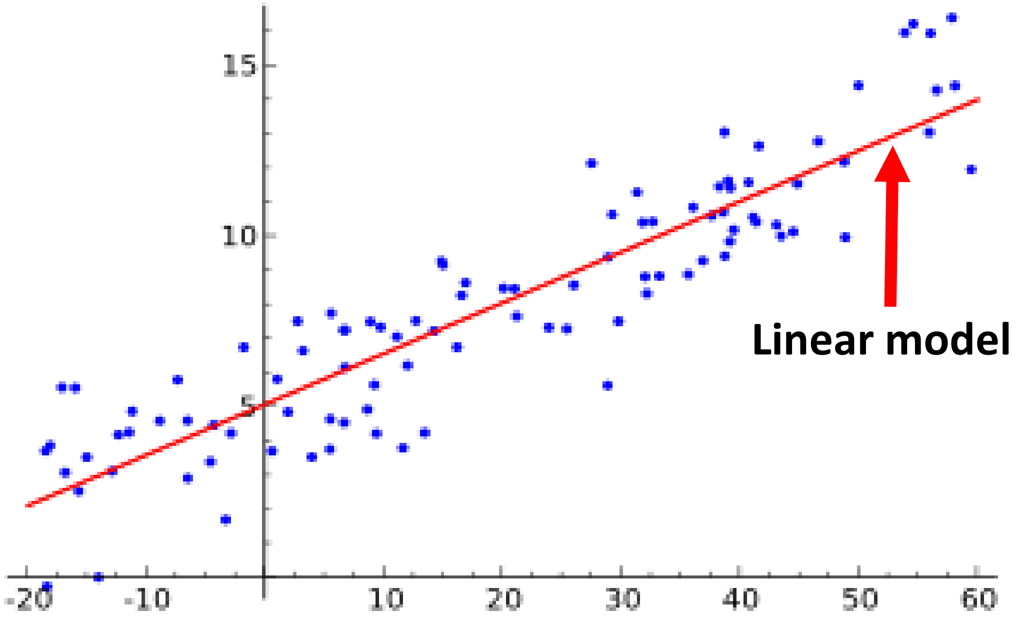
Linear Regression Classifier

Regression

- Finding an equation which explains the data
  - Explain ←→ Predict

- Started from 1800s
  - Legendre 1805, Gauss 1809

- Various regression models
  - Linear regression
  - Non-linear regression
  - Logistic regression



Linear Regression Concept

- Modeling relationship between continuous dependent variable **y** and one or more independent variables **X** using linear predictor function

$$\mathbf{Y = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n}$$

$\mathbf{x}_n$: arbitrary input, independent variable

Y: output based on $\mathbf{x}_n$, dependent variable

β: coefficients for accurate predictor function

$\boldsymbol{\beta}_0$: intercept

- Linear Regression Classifier

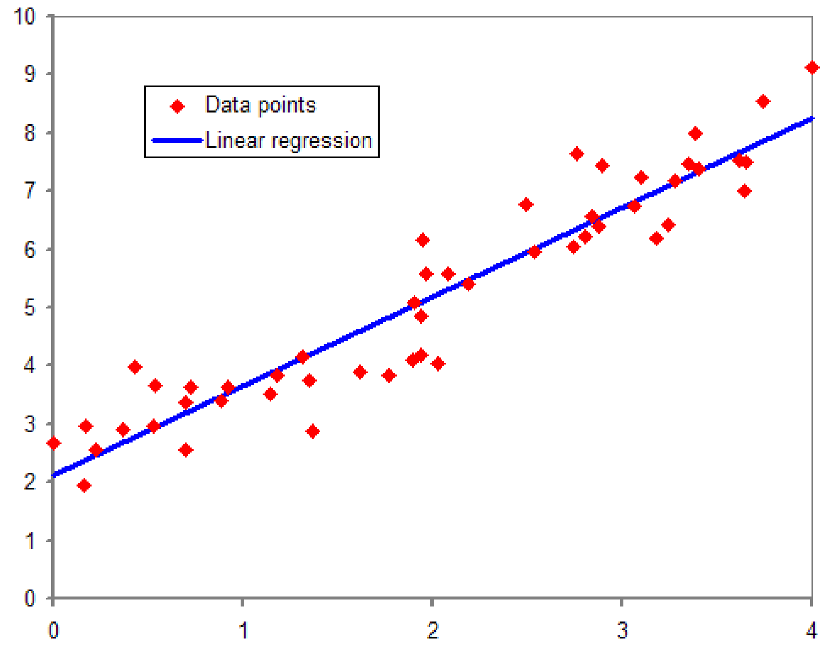
Linear Regression Concept (cont'd)

- Find the linear line minimizing distance from all points
- For new data with x values, predict Y with the linear predictor function

X	Y
0.5	2
0.7	2.5
1.2	3.4
…	….
3.6	7.5
3.8	8.2
4	9

$$Y = \beta_0 + \beta_1 X$$

New X	Predict Y
3.1	?



LinearRegression

- *linear_model*.LinearRegression([*fit_intercept*], [*normalize*], [*copy_X*], [*n_jobs*])
  - Ordinary least squares Linear Regression
  - *fit_intercept*: if False, no intercept will be used in calculations (e.g., data is expected to be already centered) (default: True)
  - *normalize*: if True, *X* will be normalized before regression (default: False)
  - *copy_X*: if True, *X* will be copied (default: True)
  - *n_jobs*: the number of jobs (CPUs) to use for the computation (default: 1)
- Attributes:
  - `coef_`, `intercept_`
- Methods:
  - `fit()`, `predict()`, `score()`, `get_params()`, `set_params()`

LinearRegression.fit()

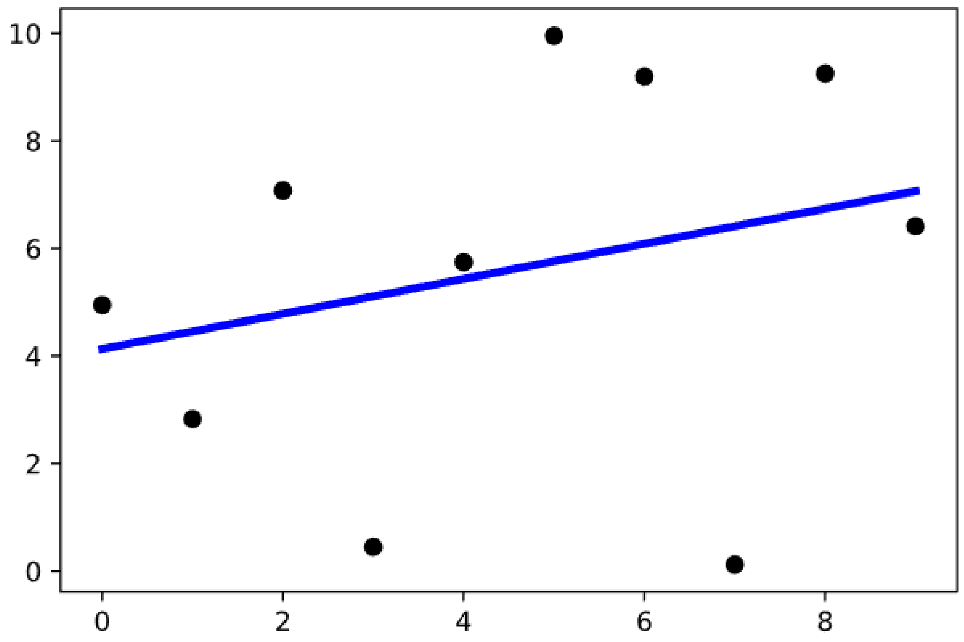
- *linear_model*.LinearRegression.`fit`(*X*, *y*, [*sample_weight*])
  - Fit linear model
  - *X*: training data, 2D array of shape [n_samples, n_features]
  - *y*: target values, 2D array of shape [n_samples, n_targets] (can be a 1D array)
  - *sample_weight*: individual weights for each sample

```
import numpy as np
from sklearn import linear_model
import matplotlib.pyplot as plt

x = [[ i ] for i in range(10) ]
y = [[ np.random.random()*10 ] for _ in range(10) ]
regr = linear_model.LinearRegression()
regr.fit(x, y)
```

LinearRegression.predict()

- *linear_model.*LinearRegression.predict(*X*)
  - Predict using the linear model
  - *X*: samples, 2D array of shape [n samples, n_features]
  - Return predicted values, 1D array of shape [n_samples,]

```
plt.scatter(x, y, c='black')
plt.plot(x, regr.predict(x), 'b-')
```



LinearRegression.score()

- *linear_model*.LinearRegression.score(*X*, *y*, [*sample_weight*])
  - Return the coefficient of determination R^2 of the prediction (variance score, 결정계수)

Score (0 ~ 1): 1 – u/v

u = ((y_true – y_pred) ** 2).sum()

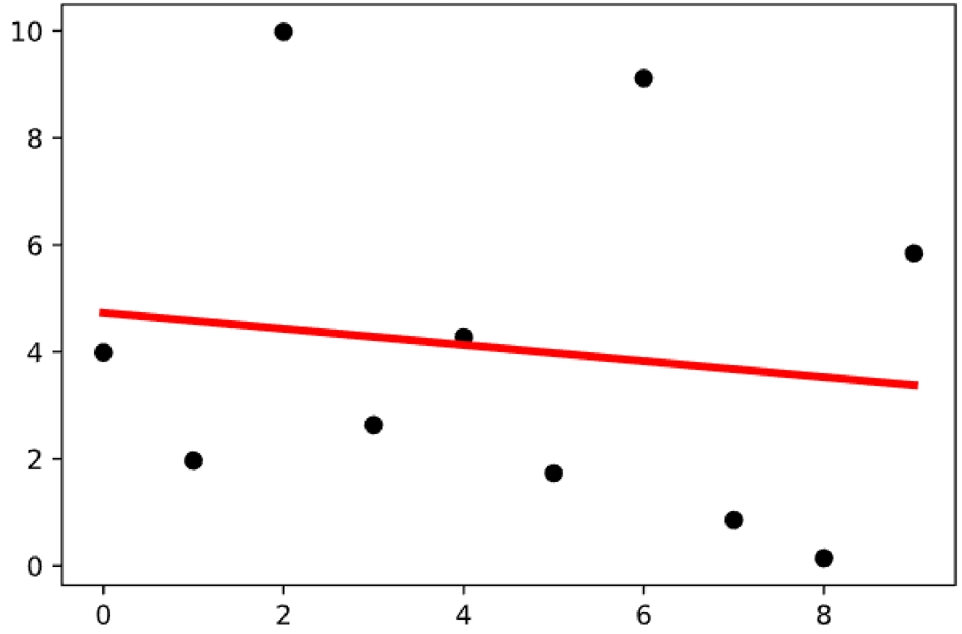
v = ((y_true – y_true.mean()) ** 2).sum()

$$R^2 = 1 - \frac{\sum_{i=0}^{n}(y_i - y'_i)^2}{\sum_{i=0}^{n}(y_i - \bar{y}_i)^2} \quad \begin{array}{l} \text{오차}^2 \\ \text{편차}^2 \end{array}$$

  - The best possible score is 1.0 and it can be negative
  - A constant model that always predicts the mean value would get $R^2 = 0$

Linear Regression using Numpy Array

```
N = 10
x = np.arange(N).reshape(N, 1)
y = (np.random.random(10)*10).reshape(N, 1)
regr = linear_model.LinearRegression()
regr.fit(x, y)
plt.scatter(x, y, c='black')
plt.plot(x, regr.predict(x), 'r-', linewidth=3)
plt.show()
```




```
x:
[[0]
 [1]
 [2]
 [3]
 [4]
 [5]
 [6]
 [7]
 [8]
 [9]]
```

```
y:
[[3.98684452]
 [1.9663881 ]
 [9.98518625]
 [2.63122155]
 [4.27284905]
 [1.73196867]
 [9.11531985]
 [0.85700765]
 [0.14199418]
 [5.84060473]]
```

Linear Regression using Pandas

```
N = 10
xy = [[i, round(np.random.random()*10,4)] for i in range(N)]
df = pd.DataFrame(data = xy, columns=('X', 'Y'))
x, y = df.X, df.Y
regr = linear_model.LinearRegression()
regr.fit(x, y)
plt.scatter(x, y, c='black')
plt.plot(x, regr.predict(x), 'r-')
plt.show()
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last) in
 4 x, y = df.X, df.Y
      5 regr = linear_model.LinearRegression()
----> 6 regr.fit(x, y)
      7 plt.scatter(x, y, c='black')
      8 plt.plot(x, regr.predict(x), 'r-')
ValueError: Expected 2D array, got 1D array instead:
array=[0 1 2 3 4 5 6 7 8 9].
```

Why?

	X	Y
0	0	7.7658
1	1	3.5812
2	2	1.1513
3	3	1.1718
4	4	4.8871
5	5	7.2234
6	6	4.1744
7	7	2.0397
8	8	1.8870
9	9	6.3205

Diabetes Example

- Using sklearn diabetes dataset (442 instances)
  - 10 attributes: Age, Sex, Body mass index (BMI), Average blood pressure (ABP), Six blood serum (S1-S6)
  - Target: quantitative measure of disease progression one year after baseline
  - All the attributes are numeric, mean centered and scaled by standard deviation
  - Can be loaded by `datasets.load_diabetes()`

442 instances

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6	target
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019908	-0.017646	151.0
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068330	-0.092204	75.0
2	0.085299	0.050680	0.044451	-0.005671	-0.045599	-0.034194	-0.032356	-0.002592	0.002864	-0.025930	141.0
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022692	-0.009362	206.0
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031991	-0.046641	135.0

...

Diabetes: Loading Dataset

```
%matplotlib inline
import numpy as np
import matplotlib as plt
from sklearn import datasets, linear_model

# Load the diabetes dataset
diabetes = datasets.load_diabetes()
```

`load_diabetes()` returns dictionary-like object.

Each field can be accessed as follows:

`diabetes.data`,
`diabetes.target`,
`diabetes.feature_names`, …

```
{'data': array([[ 0.03807591,  0.05068012,  0.06169621, ..., -0.00259226,
          0.01990842, -0.01764613],
        [-0.00188202, -0.04464164, -0.05147406, ..., -0.03949338,
         -0.06832974, -0.09220405],
        ...]]),
 'target': array([151.,  75., 141., 206., 135.,  97., 138.,  63., 110., 310., 101.,
         69., 179., 185., 118., 171., 166., 144.,  97., 168.,  68.,  49.,
         ...]),
  'DESCR': '.. _diabetes_dataset:\n\ ...
  'feature_names': ['age','sex','bmi','bp','s1','s2','s3','s4','s5','s6'],
```

Diabetes: Data Preprocessing

- Use only 3rd column values (i.e., BMI)

```python
# Use only one feature
diabetes_X = datasets.data[:, np.newaxis, 2]
```

Extract the 3rd column and add a new axis to make Nx1 2D numpy array

```
array([[ 0.06169621],
       [-0.05147406],
       [ 0.04445121],
       [-0.01159501],
       [-0.03638469],
       ...])
```

diabetes_X

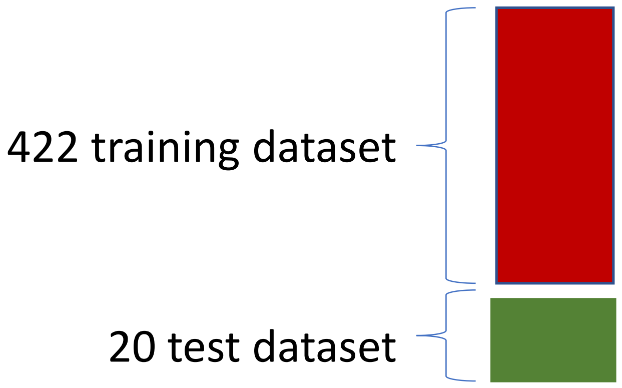
	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019908	-0.017646
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068330	-0.092204
2	0.085299	0.050680	0.044451	-0.005671	-0.045599	-0.034194	-0.032356	-0.002592	0.002864	-0.025930
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022692	-0.009362
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031991	-0.046641

Diabetes: Split Dataset for Training

- Split dataset into train dataset and test dataset

```
# Split the data into training/testing sets
diabetes_X_train = diabetes_X[:-20]
diabetes_X_test  = diabetes_X[-20:]

# Split the targets into training/testing sets
diabetes_y_train = diabetes.target[:-20]
diabetes_y_test  = diabetes.target[-20:]
```

  - Use the last 20 data for test dataset



`diabetes_X:` Numpy 2D array (442 x 1)

`diabetes.target:` Numpy 1D array (442,)

Diabetes: Learning from Data

- Create & train linear regression model with training data set

```
# Create linear regression object
regr = linear_model.LinearRegression()

# Train the model using the training sets
regr.fit(diabetes_X_train, diabetes_y_train)
```

- After training the data, we can do the followings:
  - `print(regr.coef_)` $\mathbf{Y} = \boldsymbol{\beta}_0 + \boldsymbol{\beta}_1 \mathbf{x}_1$
  - `print(regr.intercept_)`
  - `regr.predict(xi)` # xi in test_X
  - `regr.score(test_X, test_Y)` → variance score
  - `np.mean((regr.predict(test_X) - test_Y)**2)` → mean squared error

Diabetes: Validation

- Print the results of trained regression

```
print('Coefficients: \n', regr.coef_)
print('Intercept: \n', regr.intercept_)
# The mean squared error
print('Mean squared error: %.2f' %
      np.mean((regr.predict(diabetes_X_test) - diabetes_y_test) ** 2))
# Variance score: 1 is perfect prediction
print('Variance score: %.2f' %
      regr.score(diabetes_X_test, diabetes_y_test))
```

```
Coefficients:
 [938.23786125]
Intercept:
 152.91886182616167
Mean squared error: 2548.07
Variance score: 0.47
```

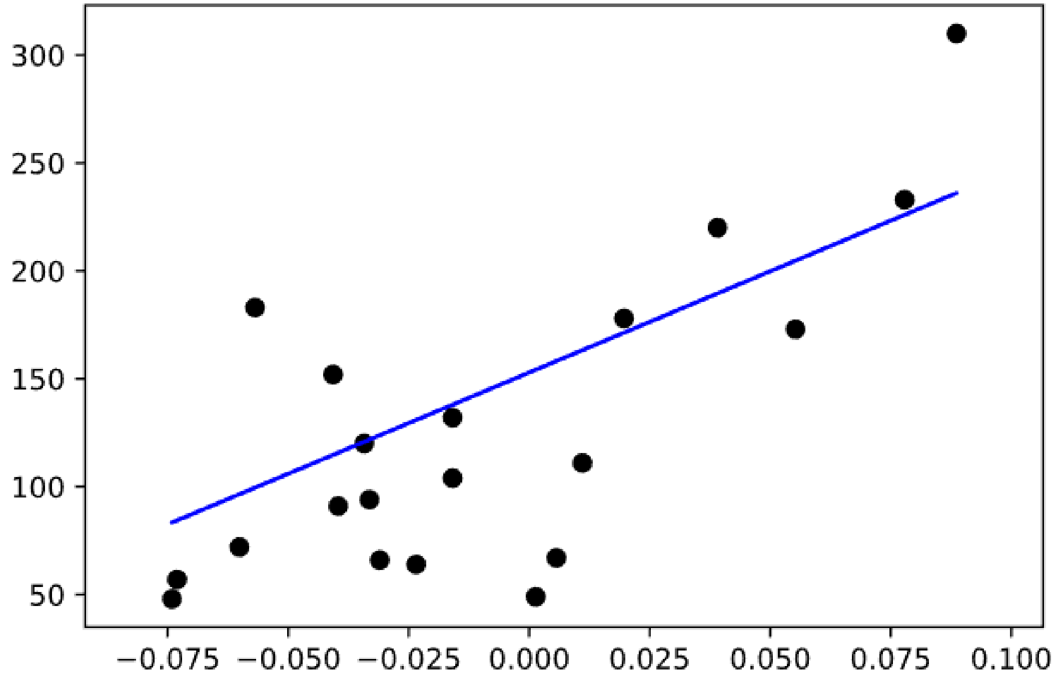
Regression Equation

$$y = \underbrace{\boxed{152.92}}_{\text{intercept}} + \underbrace{\boxed{938.24}}_{\text{coefficient}} * \text{BMI}$$

Diabetes: Result Plotting

- Plot the regression model with test data

```
# Plot outputs
plt.scatter(diabetes_X_test, diabetes_y_test, c='black')
plt.plot(diabetes_X_test, regr.predict(diabetes_X_test), 'b-')
```

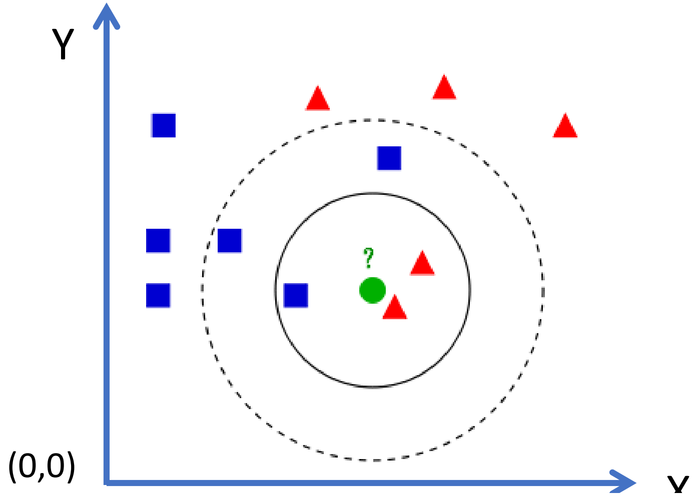


K-Nearest Neighbor (KNN) Classifier

KNN Classifier

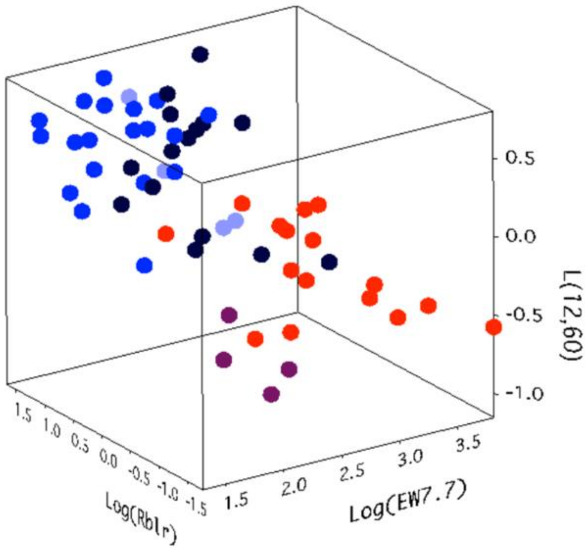
X	Y	Shape
1	3	Rectangle
1	4	Rectangle
1	7	Rectangle
…	…	
3	7.1	Triangle
5	7.2	Triangle
7	7.0	Triangle

(4, 3) → Shape?



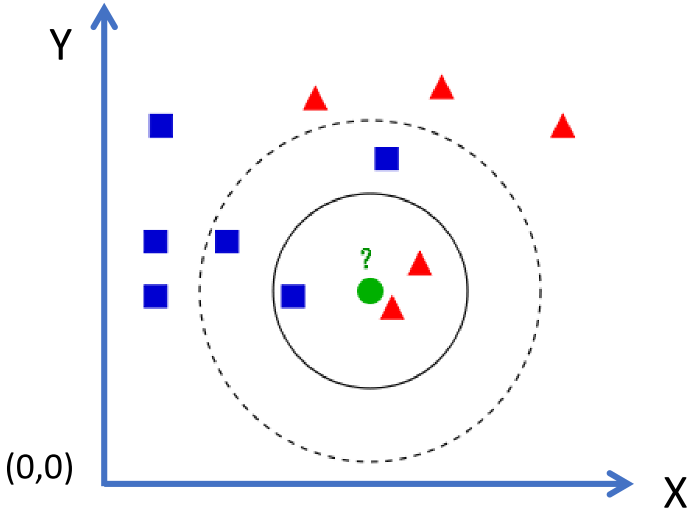
X	Y	Z	Color Ball
0.3	0.4	2.44	Blue
1.2	1.67	2.22	Red
…	….	….	
1.1	1.9	3.8	Black

(0.5, 1.2, 2.6) → Color?



k-Nearest Neighbors (KNN)

- Assume that similar data will be located closely
- Determine the class of new data based on *k* closest data
- No model (no formula) is used, only data is used for KNN
- Various distance metrics
  - Euclidean distance, Manhattan distance, Mahalanobis distance, etc.



Want to classify new data

Distance vs. Count?

For k = 3, 2 triangles & 1 rectangle
Result → classify new data as red triangle

For k = 5, 2 triangles & 3 rectangles
Result → classify new data as blue rectangle

KNeighborsClassifier

- *neighbors*.KNeighborsClassifier([*n_neighbors*], [*weights*], [*algorithm*], [*leaf_size*], [*p*], [*metric*], [*n_jobs*], ...)
  - Classifier implementing the k-nearest neighbors vote
  - *n_neighbors*: number of neighbors to use (default: 5)
  - *weights*: weight function: 'uniform', 'distance', or user-defined (default: 'uniform')
  - *algorithm*: 'ball_tree', 'kd_tree', 'brute', or 'auto' (default: 'auto')
  - *leaf_size*: leaf size for BallTree or KDTree
  - *p*: power parameter for minkowski metric (1: Manhattan, 2: Euclidean)
  - *metric*: the distance metric to use for the tree (default: minkowski)
  - *n_jobs*: number of jobs for computation (default: 1)

fit() and predict()

- *neighbors*.KNeighborsClassifier.`fit`(*X*, *y*)
  - Fit the model using *X* as training data and *y* as target values
  - *X*: training data, 2D array of shape [n_samples, n_features]
  - *y*: target values, array of shape [n_samples] or [n_samples, n_outputs]

- *neighbors*.KNeighborsClassifier.`predict`(*X*)
  - Predict the class labels for the provided data
  - *X*: test samples, 2D array of shape [n_queries, n_features]
  - *y*: class labels for each data sample, array of shape [n_samples] or [n_samples, n_outputs]

kneighbors()

- *neighbors*.KNeighborsClassifier.`kneighbors`(*X*, [*n_neighbors*], [*return_distance*])
  - Finds the K-neighbors of a point
  - *X*: query point(s), array of shape [n_queries, n_features]
  - *n_neighbors*: number of neighbors to get
  - *return_distance*: if False, distances will not be returned
  - Return distance and indices of the nearest points

Iris Example

- Using sklearn iris dataset (150 instances)
  - 3 classes, 50 instances for each class
  - 4 column data: sepal(꽃받침) length, sepal width, petal(꽃잎) length, petal width
  - Can be loaded by `datasets.load_iris()`

150 instances

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

...

target
0
0
0
0
0

Iris: Loading Dataset

```
%matplotlib inline
import numpy as np
import matplotlib as plt
from sklearn import datasets, neighbors

# Load the iris dataset
iris = datasets.load_iris()
```

`load_iris()` returns dictionary-like object.

Each field can be accessed as follows:

`iris.data`,

`iris.target`,

`iris.feature_names`, ...

```
{'data': array([[5.1, 3.5, 1.4, 0.2],
        [4.9, 3. , 1.4, 0.2],
        [4.7, 3.2, 1.3, 0.2],
        [4.6, 3.1, 1.5, 0.2],
        ...]]),
 'target': array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        ...]),
 'target_names': array(['setosa', 'versicolor', 'virginica'], dtype='<U10'),
 'DESCR': '.. _iris_dataset:\n\ ...
 'feature_names': ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)'],
```

Iris: Data Preprocessing

- Use only 1st and 2nd column values (i.e., sepal length and sepal width)

```
# We only take the first two features
iris_X = iris.data[:, :2]

iris_y = iris.target
```

```
array([[5.1, 3.5],
       [4.9, 3. ],
       [4.7, 3.2],
       [4.6, 3.1],
       [5. , 3.6],
       ...])
```

iris_X

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

Iris: Train and Predict KNN Classifier

- Create & train KNN model with training set

```python
n_neighbors = 15

# Create KNeighborsClassifier object
neigh = neighbors.KNeighborsClassifier(n_neighbors, weights='uniform')

# Train the model using the training sets
neigh.fit(iris_X, iris_y)
```

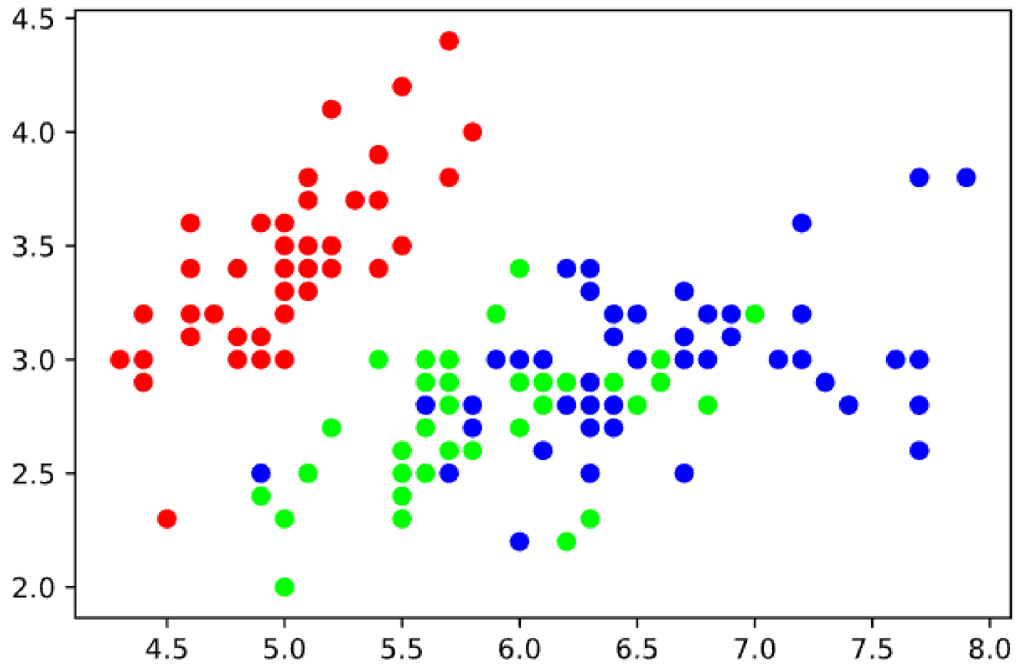
- Predict the class of a new sample

```python
new_sample = [[3.7, 4.5]]
iris_class = neigh.predict(new_sample)
print('The iris class for new sample:', iris.target_names[iris_class[0]])
```

```
The iris class for new sample: setosa
```

Iris: Plotting the Dataset

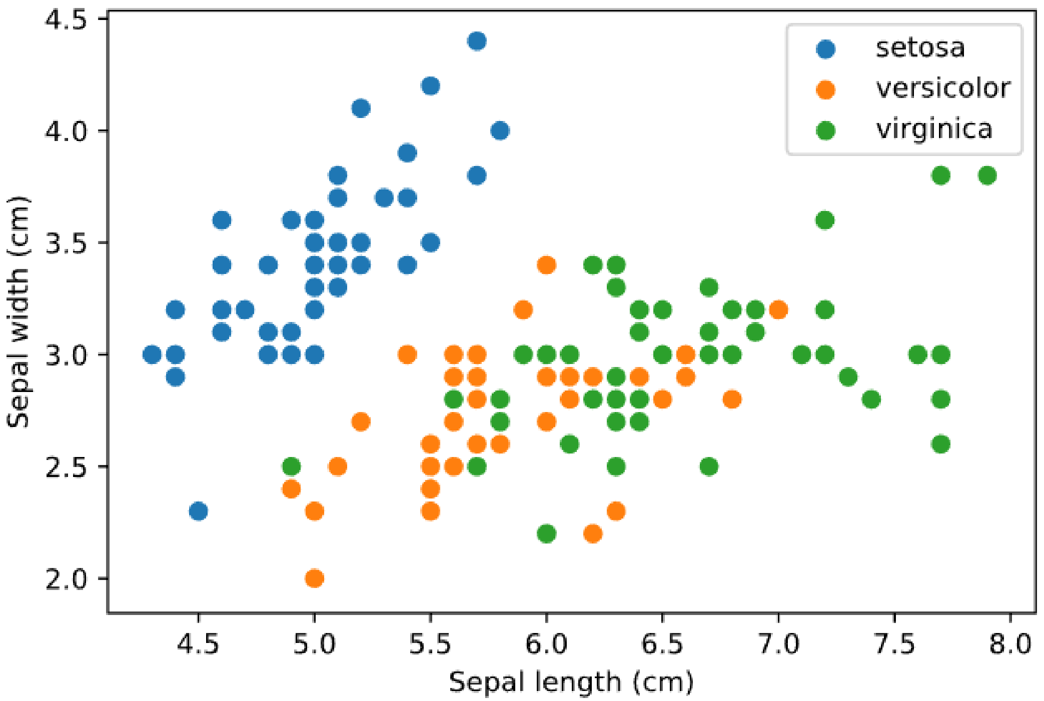
```
import matplotlib.colors as matcol

cmap_iris = matcol.ListedColormap(
            ['#ff0000', '#00ff00', '#0000ff'])

plt.scatter(iris_X[:, 0], iris_X[:, 1], c=iris_y,
            cmap=cmap_iris)
```



Iris: Plotting the Dataset (with Pandas)

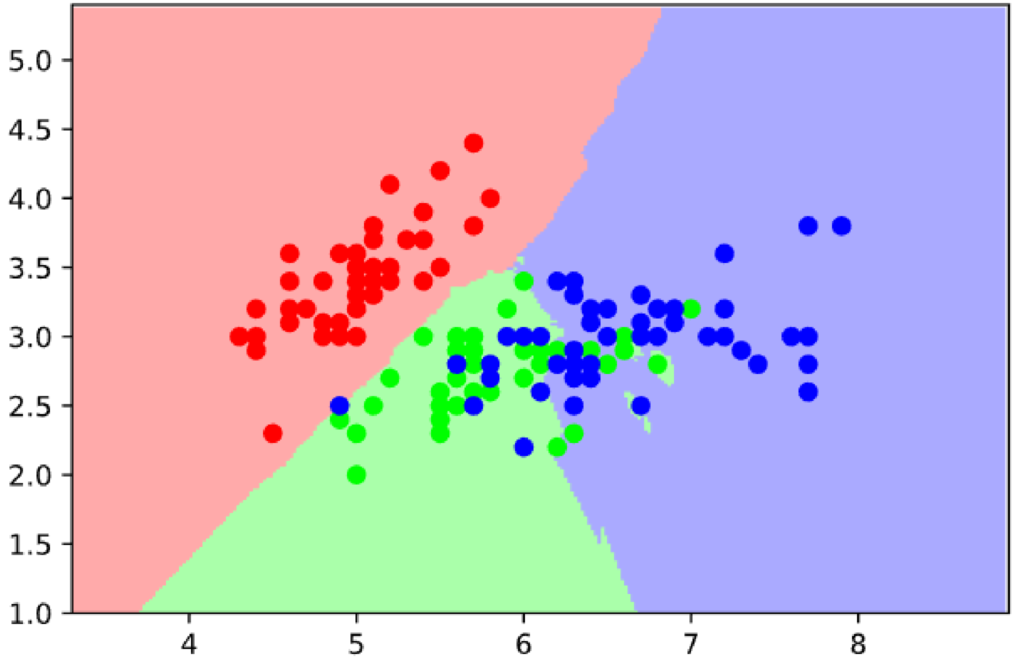
```
cols = ['SepalLength', 'SepalWidth',
        'PetalLength', 'PetalWidth']
df = pd.DataFrame(iris.data, columns=cols)
df2 = pd.DataFrame(iris.target, columns=['Class'])
df = pd.concat([df, df2], axis=1)

groups = df.groupby('Class')
for cls, group in groups:
    plt.scatter(group.SepalLength, group.SepalWidth,
                label=iris.target_names[cls])
plt.xlabel('Sepal length (cm)')
plt.ylabel('Sepal width (cm)')
plt.legend()
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Class
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0



Iris: Plotting Prediction Results

```
# Find the min, max of each axis
x_min = iris_X[:, 0].min() - 1
x_max = iris_X[:, 0].max() + 1
y_min = iris_X[:, 1].min() - 1
y_max = iris_X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))
# Get coordinates
xr, yr = xx.flatten(), yy.flatten()
xy = np.c_[xr, yr]
# Get prediction results
z = neigh.predict(xy)
zz = z.reshape(xx.shape)
# Plot using pcolormesh()
cmap_light = matcol.ListedColormap(['#FFAAAA',
             '#AAFFAA', '#AAAAFF'])
plt.pcolormesh(xx, yy, zz, cmap=cmap_light)
plt.scatter(iris_X[:, 0], iris_X[:, 1], c=iris_y,
            cmap=cmap_iris)
```



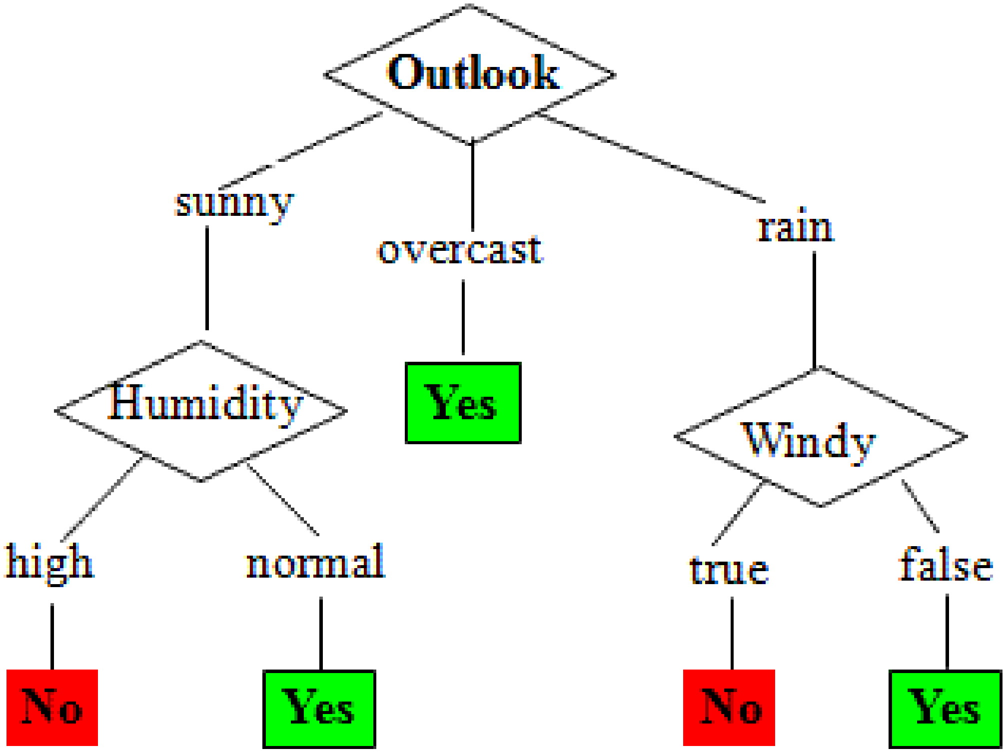
Decision Tree Classifier

Decision Tree

- Classifier using Tree

- Early Decision Trees
  - CHAID (1980), CHART (1984)

- Current Decision Trees
  - ID3 (1986) → C4.5 (1993) → C5.0
  - C5.0: commercial

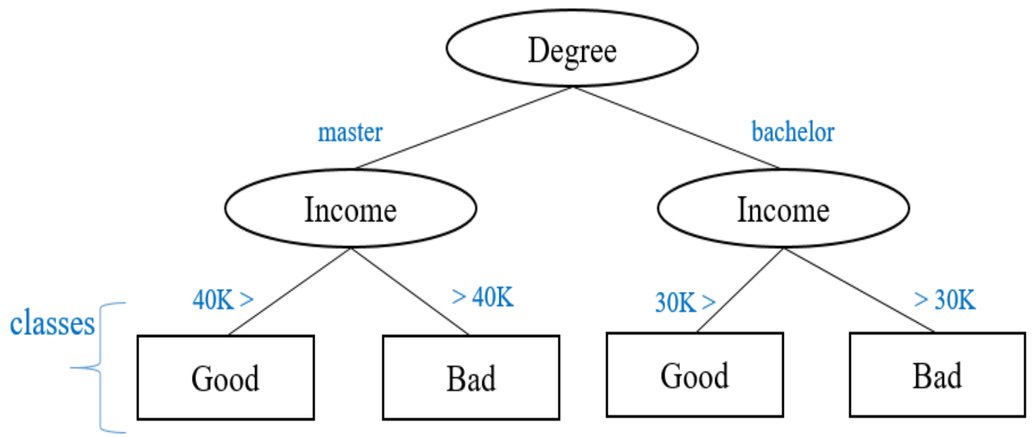


What is Decision Tree?

- Supervised learning model
- Flowchart-like structure to classify an outcome based on a set of predictors
  - Ellipse node: split condition
  - Rectangle node: classified class (= leaf node)

Name	Degree	Income	Credit Status
H.Kim	Master	$50000	Good
P. Lee	Bachelor	$35000	Good
J. Hong	Master	$18000	Bad
W. Sawn	Master	$39000	Good
J. Doe	Bachelor	$55000	Good
W. Son	Bachelor	$25000	Bad
Q. Li	Bachelor	$15000	Bad

Decision tree classifier



What is Decision Tree? (cont'd)

- Split dataset based on an attribute with highest purity
  - Purity (p_k): proportion of data in a split that belong to class *k*
  - Maximum purity: each split has data of same class
  - Impurity measure: Gini Index, Entropy

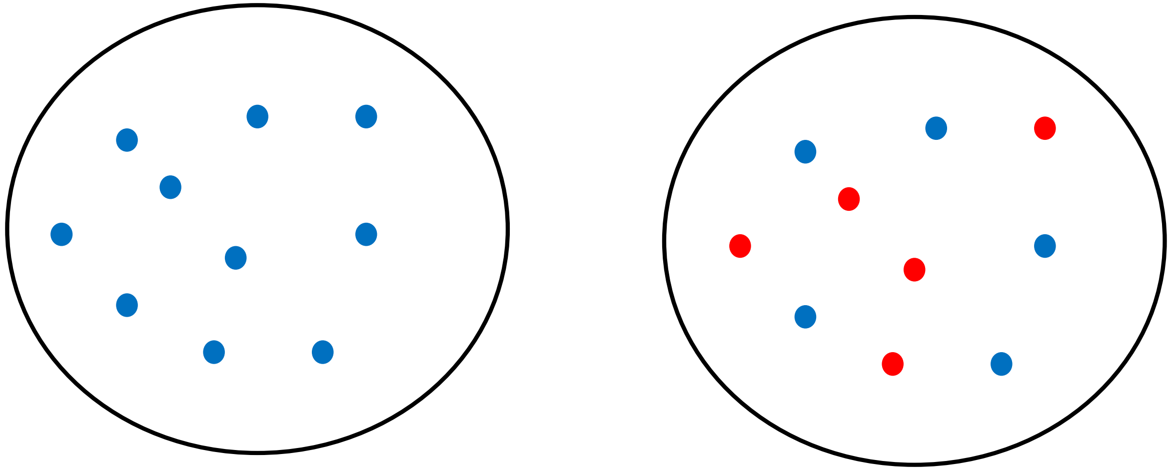
Gini Index: **0 (pure)** $\sim (m-1)/m$

$$I(A) = 1 - \sum_{k=1}^{m} p_k 2$$

Entropy: **0 (pure)** $\sim \log_2(m)$

$$entropy(A) = -\sum_{k=1}^{m} p_k log_2(p_k)$$

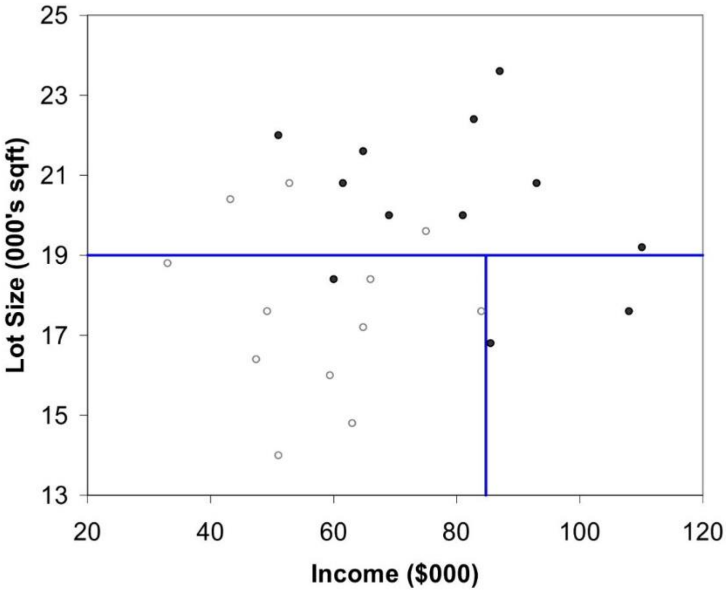
m: *total number of classes*

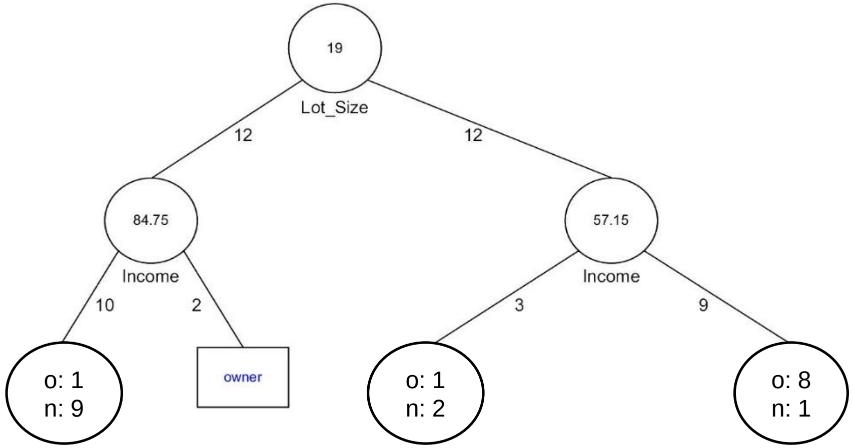


Max Purity **Min Purity**

Ownership of Riding Lawn Mower

Income ($1000)	Lot_Size (1000 ft^2)	Ownership
60.0	18.4	owner
85.5	16.8	owner
64.8	21.6	owner
61.5	20.8	owner
87.0	23.6	owner
110.1	19.2	owner
108.0	17.6	owner
82.8	22.4	owner
69.0	20.0	owner
93.0	20.8	owner
51.0	22.0	owner
81.0	20.0	owner
75.0	19.6	non-owner
52.8	20.8	non-owner
64.8	17.2	non-owner
43.2	20.4	non-owner
84.0	17.6	non-owner
49.2	17.6	non-owner
59.4	16.0	non-owner
66.0	18.4	non-owner
47.4	16.4	non-owner
33.0	18.8	non-owner
51.0	14.0	non-owner
63.0	14.8	non-owner





Recursive Partitioning in Decision Tree

- Pick one of the predictor variables, x_i
- Pick a value of x_i, say s_i, that divides the training data into two (not necessarily equal) portions
- Measure how "pure" or homogeneous each of the resulting portions are
  - "Pure" = containing records of mostly one class
- Algorithm tries different values of x_i, and s_i to maximize purity in initial split
- After you get a "maximum purity" split, repeat the process for a second split, and so on

Example:

- Goal: Classify 24 households as owning or not owning riding lawn mowers
  - Predictors = Income, Lot size

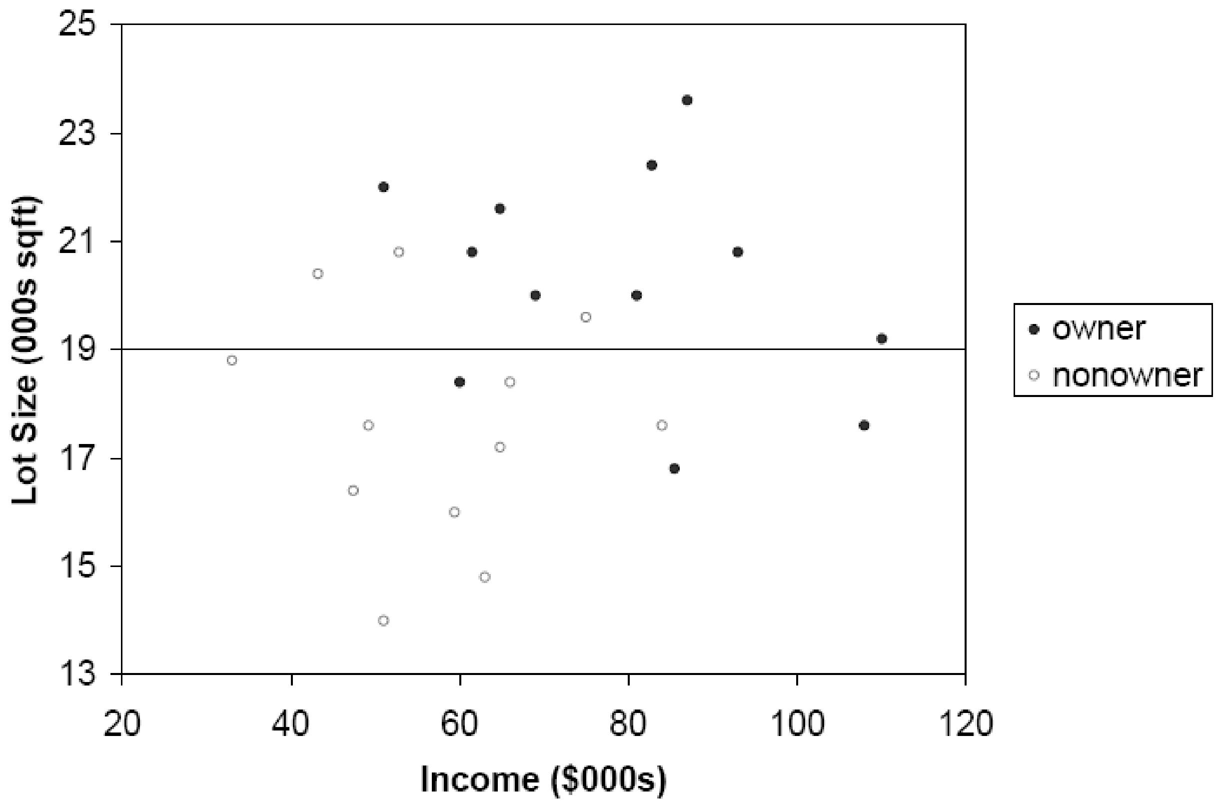
- How to split the values of continuous variable?
  - Sort records according to one variable (say, lotsize)
  - Find the split point of lotsize (halfway between 14.0 and 23.6 → 19) using Gini Index calculation
  - Divide records into those with lotsize > 19 and those with lotsize < 19
  - For each splitted area, try the next variable (say, income), which is $84,000 and $57,000

Income	Lot_Size	Ownership
60.0	18.4	owner
85.5	16.8	owner
64.8	21.6	owner
61.5	20.8	owner
87.0	23.6	owner
110.1	19.2	owner
108.0	17.6	owner
82.8	22.4	owner
69.0	20.0	owner
93.0	20.8	owner
51.0	22.0	owner
81.0	20.0	owner
75.0	19.6	non-owner
52.8	20.8	non-owner
64.8	17.2	non-owner
43.2	20.4	non-owner
84.0	17.6	non-owner
49.2	17.6	non-owner
59.4	16.0	non-owner
66.0	18.4	non-owner
47.4	16.4	non-owner
33.0	18.8	non-owner
51.0	14.0	non-owner
63.0	14.8	non-owner

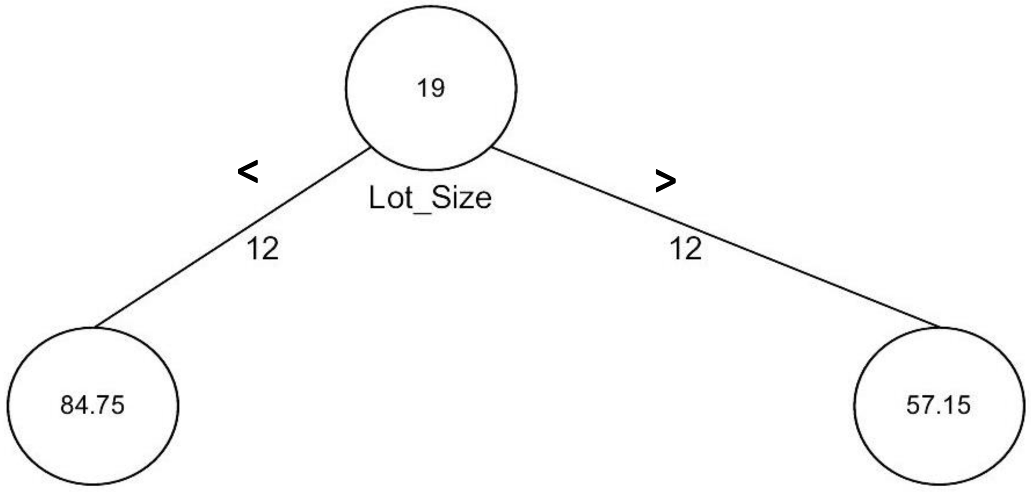
Example: The First Split

- Lot size → 19,000 sqft

1st Round



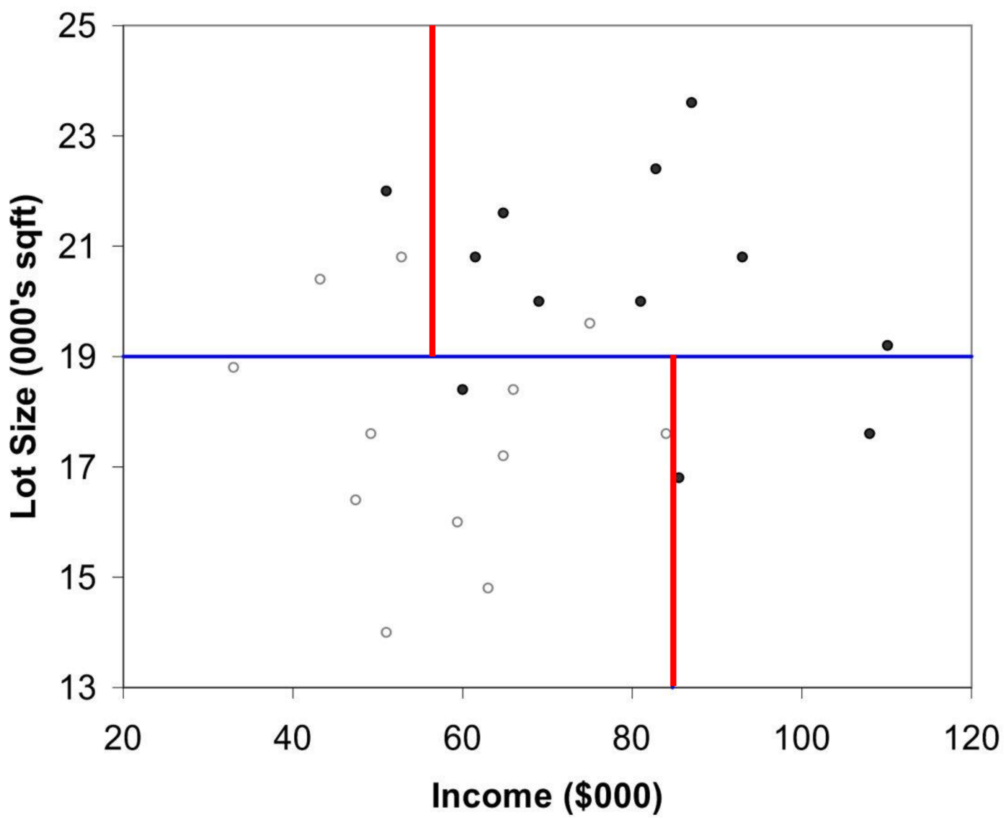
Tree structure



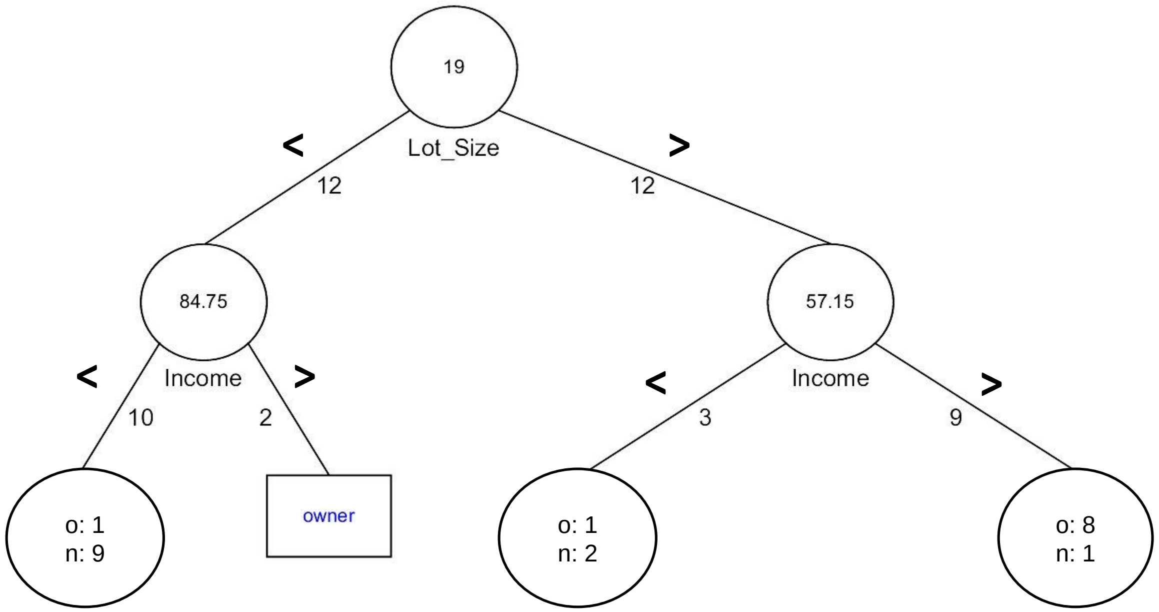
Example: The Second Split

- Income → \$84000 & \$57,000

2nd Round

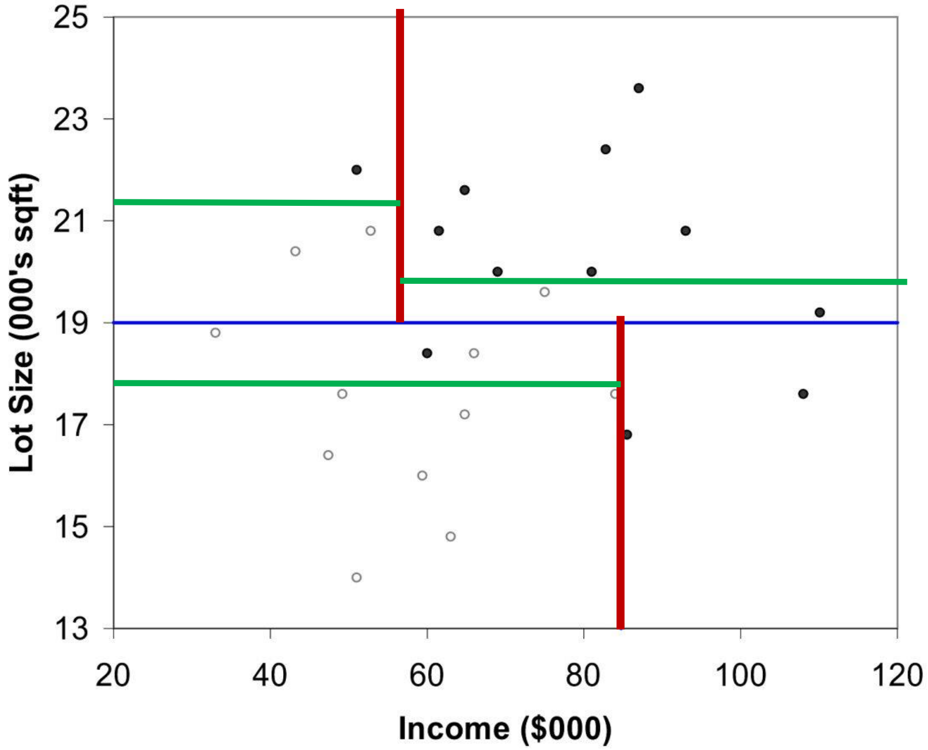


Tree structure

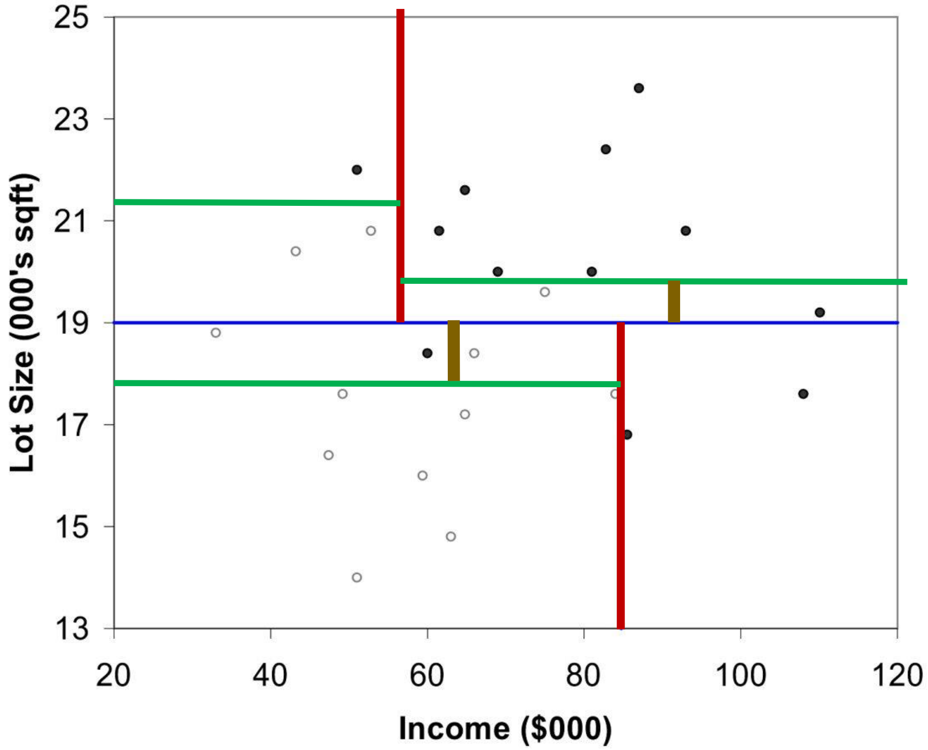


Example: The Third & Fourth Split

3rd Round

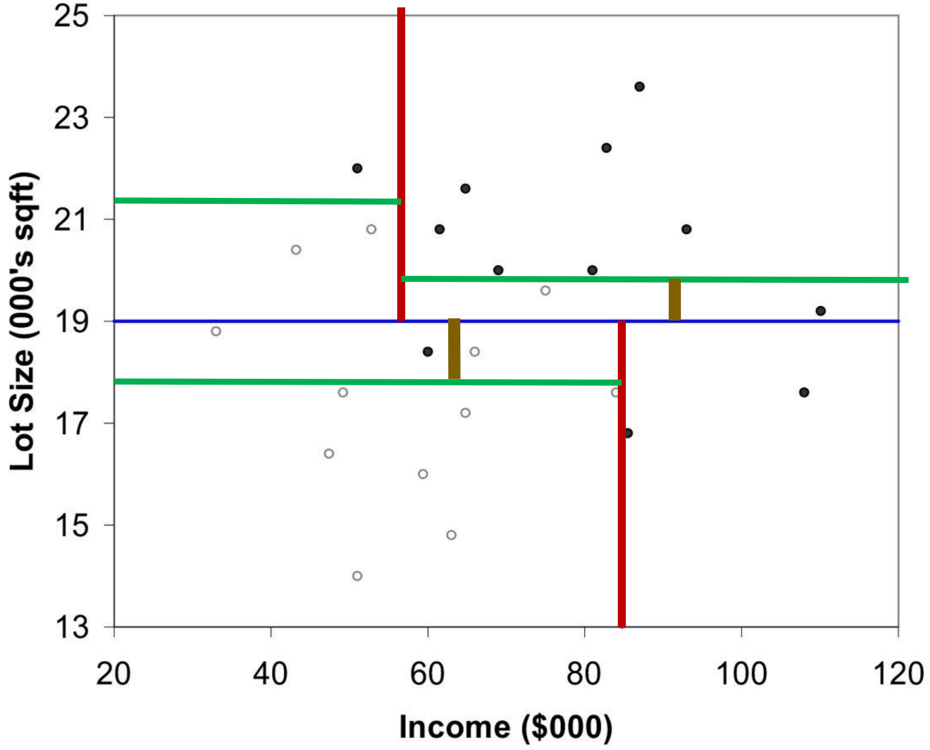


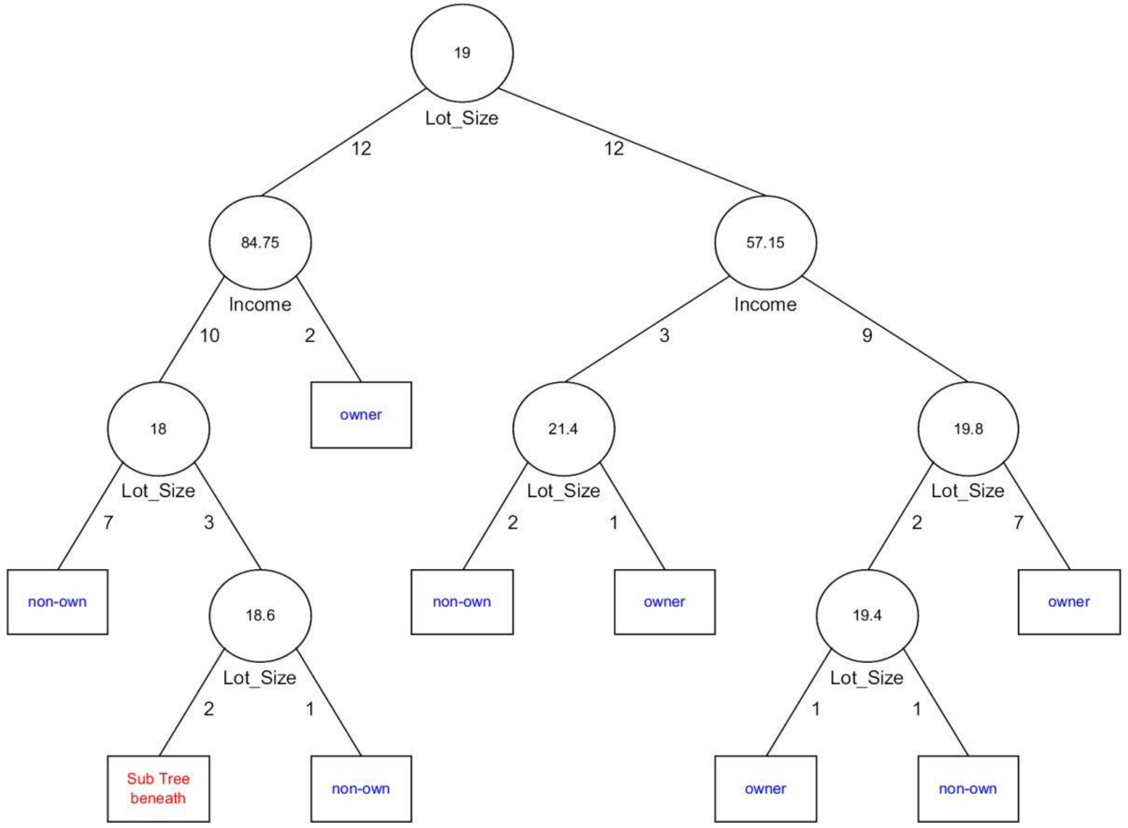
4th Round



Example: After All Splits

- Result of full-grown tree: each rectangle is completely pure
- Danger of overfitting problem





Stopping Tree Growth

- Natural end of process is 100% purity in each leaf
- This overfits the data, which end up fitting noise in the data
- Overfitting leads to low predictive accuracy of new data
- Past a certain point, the error rate for the validation data starts to increase

DecisionTreeClassifier

- *tree*.DecisionTreeClassifier([*criterion*], [*splitter*], [*max_depth*], [*min_samples_split*], [*min_samples_leaf*], [*max_features*], [*min_impurity_split*], ...)
  - A decision tree classifier
  - *criterion*: function to measure the quality of a split – 'gini' (default) or 'entropy'
  - *splitter*: strategy used to choose the split at each node – 'best' (default) or 'random'
  - *max_depth*: maximum depth of the tree
  - *min_samples_split*: min. # of samples required to split an internal node (default: 2)
  - *min_samples_leaf*: min. # of samples required to be at a leaf node (default: 1)
  - *max_features*: number of features to consider when looking for the split
  - *min_impurity_split*: A node will be split if this split induces a decrease of the impurity greater than or equal to this value (default: 0.)

fit() and predict()

- *tree*. DecisionTreeClassifier.`fit`(*X*, *y*, ...)
  - Build a decision tree classifier from the training set (*X*, *y*)
  - *X*: training input samples, array of shape [n_samples, n_features]
  - *y*: target values, array of shape [n_samples] or [n_samples, n_outputs]

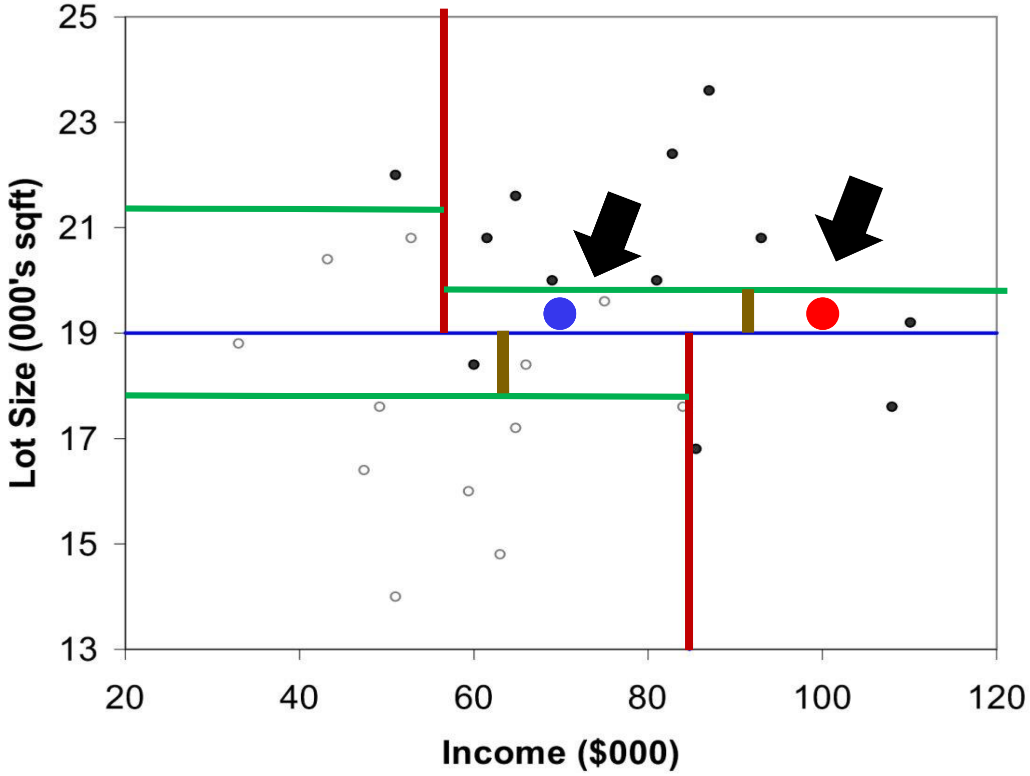
- *tree*. DecisionTreeClassifier.`predict`(*X*)
  - Predict the class or regression value for *X*
  - *X*: input samples, array of shape [n_samples, n_features]

Lawn Mower using Python Lists (1)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import datasets, tree

income =  [60.0, 85.5, 64.8, 61.5, 87.0, 110.1, 108.0, 82.8, 69.0, 93.0, 51.0, 81.0,
           75.0, 52.8, 64.8, 43.2, 84.0, 49.2, 59.4, 66.0, 47.4, 33.0, 51.0, 63.0]
lotsize = [18.4, 16.8, 21.6, 20.8, 23.6, 19.2, 17.6, 22.4, 20.0, 20.8, 22.0, 20.0,
           19.6, 20.8, 17.2, 20.4, 17.6, 17.6, 16.0, 18.4, 16.4, 18.8, 14.0, 14.8]
ownership = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
xy = [ [income[i], lotsize[i]] for i in range(len(income)) ]
z = [ [i] for i in ownership ]

dt = tree.DecisionTreeClassifier()
dt.fit(xy, z)
```

Lawn Mower using Python Lists (2)

```
>>> print(dt.predict([[100., 19.2]]))
[1]       # Owner
>>> print(dt.predict([[70.0, 19.2]]))
[0]       # Not-owner
```



Lawn Mower using Python Lists (3)

```
print('min_samples_leaf = 1')
for i in range(len(xy))
    print(dt.predict([xy[i]]), z[i])

dt = tree.DecisionTreeClassifier
      (min_samples_leaf=2)
dt.fit(xy, z)

print('min_samples_leaf = 2')
for i in range(len(xy))
    print(dt.predict([xy[i]]), z[i])
```

```
min_samples_leaf = 1
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
```

```
min_samples_leaf = 2
[0] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[1] [1]
[0] [1]
[1] [1]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
[0] [0]
```

Decision Tree using NumPy

```
N = 10
x = np.array(range(N)).reshape(N, 1)
y = (np.random.random(N)*10).reshape(N, 1)
xy = np.concatenate((x, y), axis=1)
z = np.array([1, 1, 1, 1, 1, 2, 2, 2, 2, 2])

dt = tree.DecisionTreeClassifier()
dt.fit(xy, z)

print(dt.predict([[3, 3.01]]))
print(dt.predict(np.array([[7, 8.01]])))
```

```
[1]
[2]
```

```
x: [[0]
    [1]
    [2]
    [3]
    [4]
    [5]
    [6]
    [7]
    [8]
    [9]]
```

```
y: [[9.52890254]
    [2.15185172]
    [0.81483473]
    [8.63402819]
    [7.1018699 ]
    [9.11495804]
    [6.92878962]
    [6.92305586]
    [7.32459969]
    [9.80431005]]
```

```
xy: [[0.         9.52890254]
     [1.         2.15185172]
     [2.         0.81483473]
     [3.         8.63402819]
     [4.         7.1018699 ]
     [5.         9.11495804]
     [6.         6.92878962]
     [7.         6.92305586]
     [8.         7.32459969]
     [9.         9.80431005]]
```

Decision Tree using Pandas

```
N = 10
x = np.array(range(N)).reshape(N, 1)
y = (np.random.random(N)*10).reshape(N, 1)
xy = np.concatenate((x, y), axis=1)
p_xy = pd.DataFrame(xy, columns=['x', 'y'])
p_z = pd.DataFrame([1, 1, 1, 1, 1, 2, 2, 2, 2, 2],
                   columns=['z'])

dt = tree.DecisionTreeClassifier()
dt.fit(p_xy, p_z)

print(dt.predict([[3, 3.01]]))
print(dt.predict(np.array([[7, 8.01]])))
```

```
[1]
[2]
```

	x	y
0	0.0	1.523718
1	1.0	8.331948
2	2.0	9.847882
3	3.0	4.707726
4	4.0	6.254159
5	5.0	5.520755
6	6.0	2.372673
7	7.0	9.910699
8	8.0	6.541729
9	9.0	8.952439

	z
0	1
1	1
2	1
3	1
4	1
5	2
6	2
7	2
8	2
9	2

Iris: Loading and Training

- Use all the data columns

```
%matplotlib inline
import numpy as np
import matplotlib as plt
from sklearn import datasets, tree

# Load the iris dataset
iris = datasets.load_iris()

# Train with DecisionTreeClassifier
dt = tree.DecisionTreeClassifier()
dt.fit(iris.data, iris.target)
a = dt.predict([[4.8, 3.1, 1.5, 0.2]])
print(a)
```

```
[0]
```

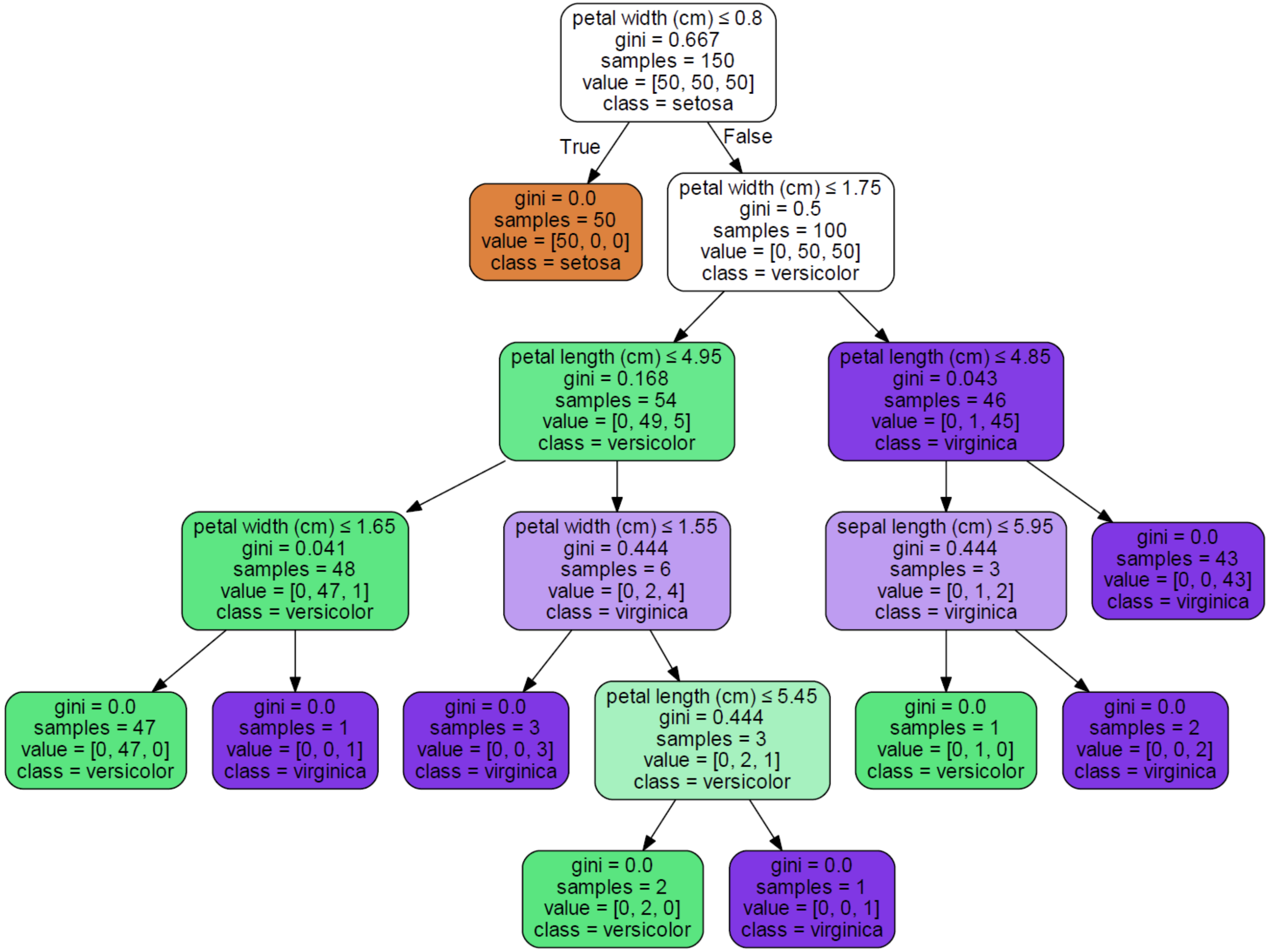
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

Iris: Drawing Decision Tree

- Requires the installation of pydotplus and GraphViz packages

```
import io
import pydotplus

# Convert the decision tree in dot language code
dot_data = io.StringIO()
tree.export_graphviz(dt, out_file=dot_data, feature_names=iris.feature_names,
                     class_names=iris.target_names, filled=True, rounded=True,
                     special_characters=True)

# Transform dot language code to graph by calling GraphViz
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_pdf('iris.pdf')
```

Iris: Decision Tree



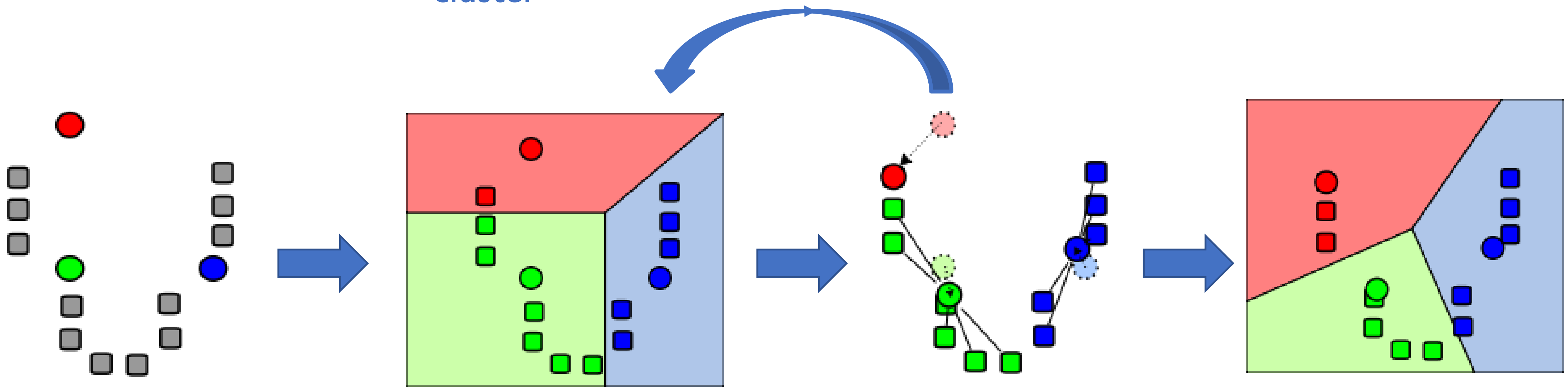
K-Means Clustering

K-Means Clustering

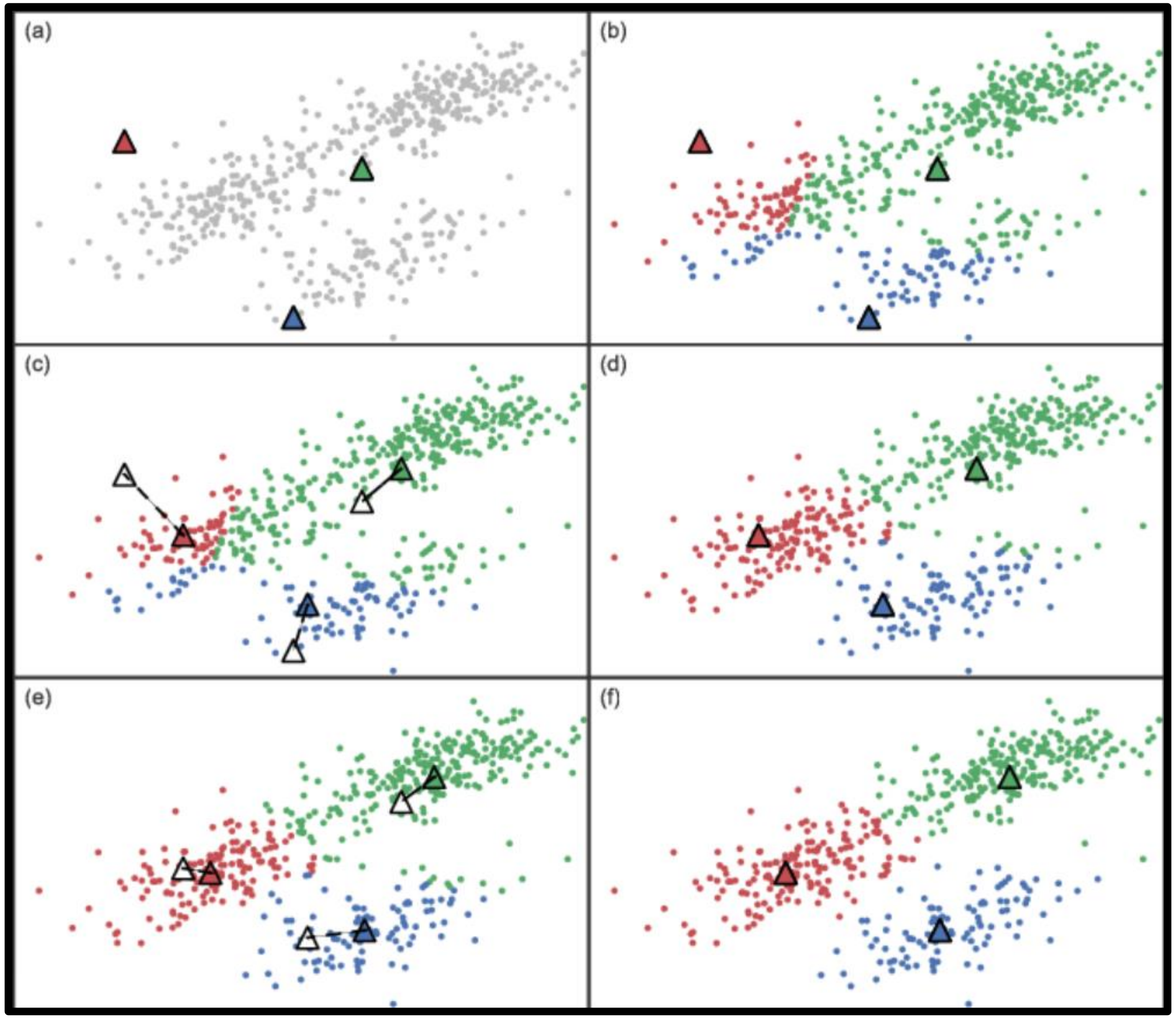
- Unsupervised learning model
- Similar to K Nearest-Neighbor algorithm, assume that similar data will be located closely
- Based on such assumption, k-means algorithm aims to partition *n* data into *k* clusters
- Each observation belongs to the cluster with nearest centroid (mean)

K-Means Clustering Procedure

- Step 1
  - Among given data, pick *k* centroids randomly
- Step 2
  - Calculate distance between all data and centroids; Assign each data point to the closest centroid's cluster
- Step 3
  - Relocate each clusters centroids to the mean points
- Step 4
  - Repeat Step 2 & 3 until convergence

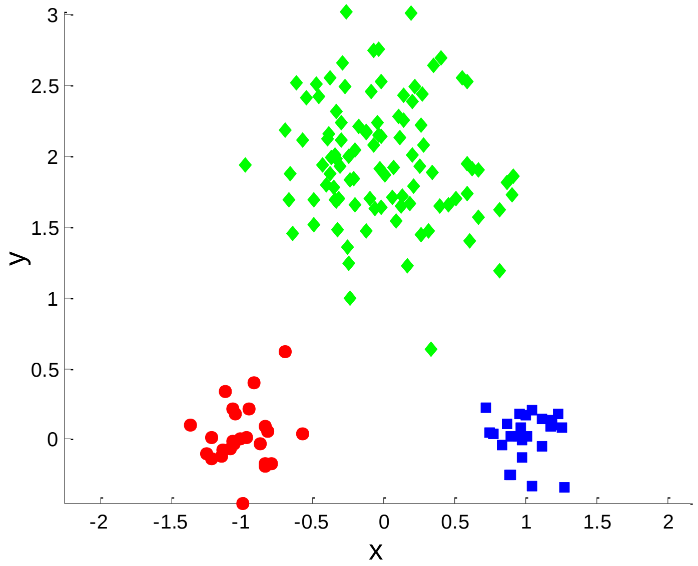


K-Means Clustering Example

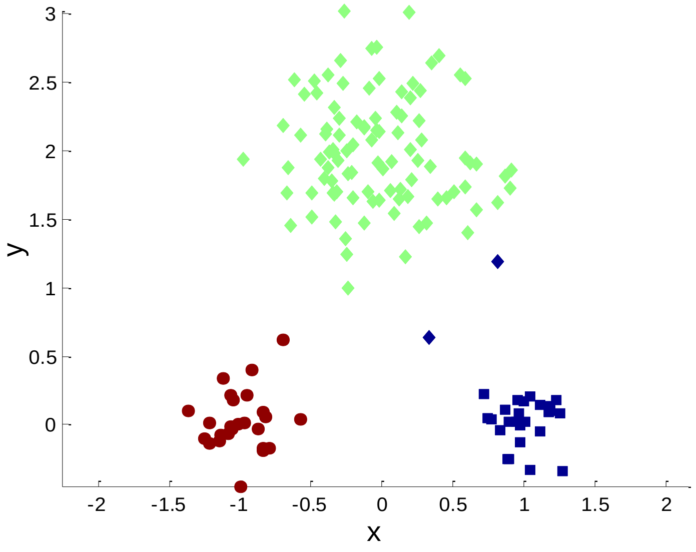


Two Different Results

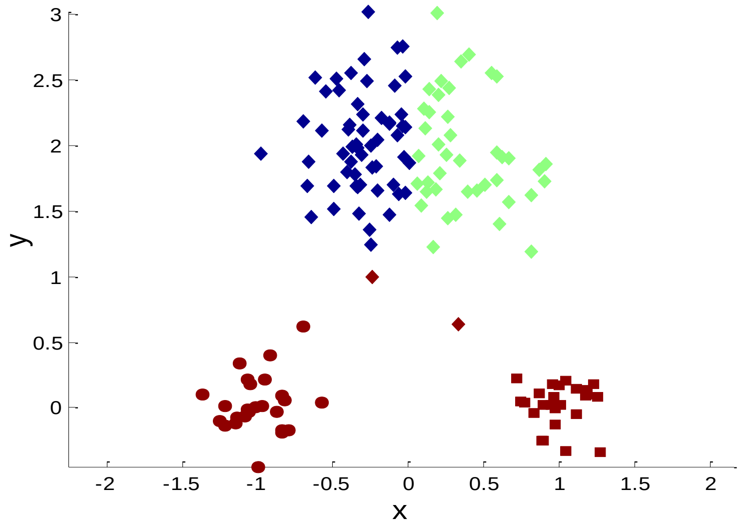
Original Points



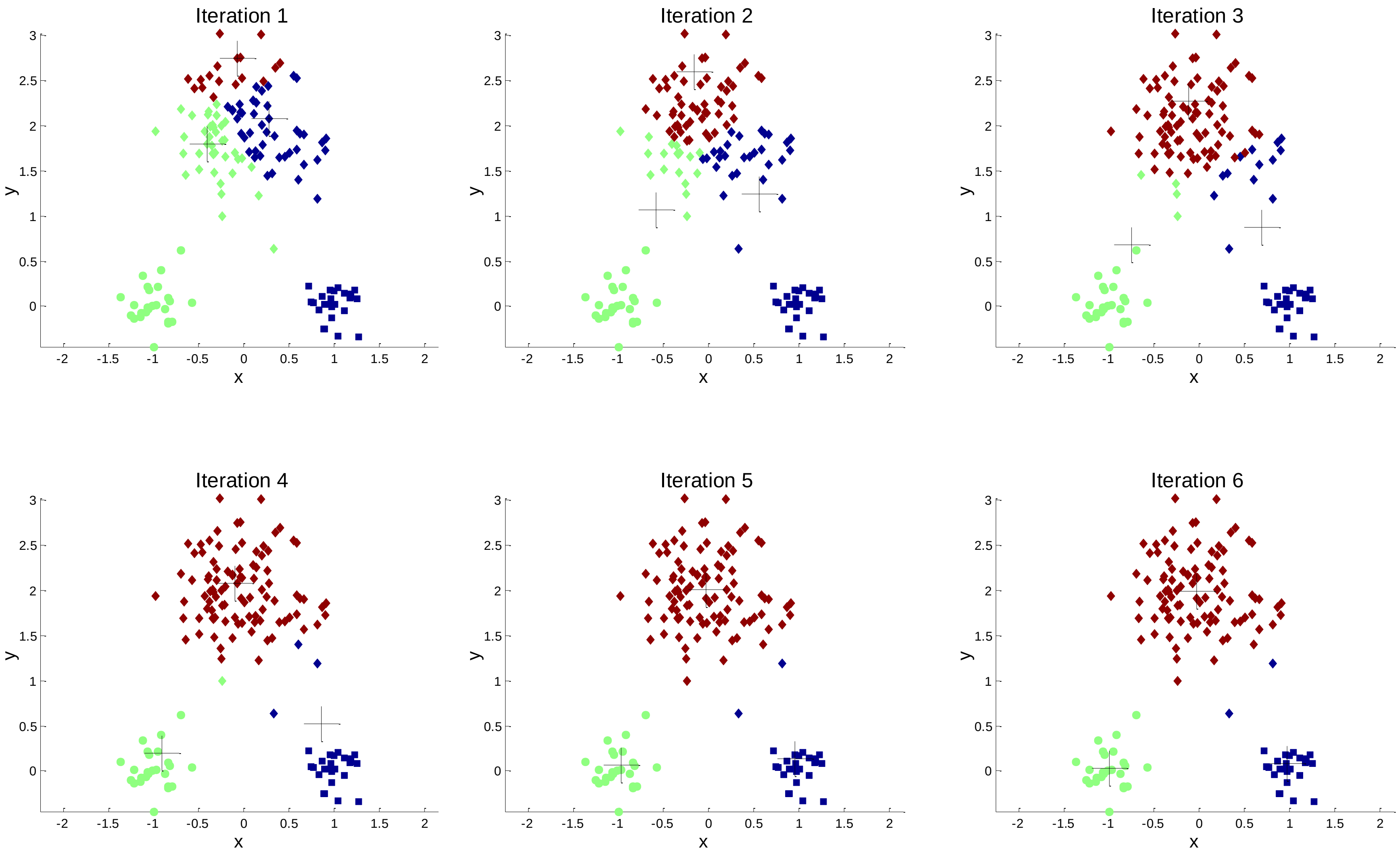
Optimal Clustering



Sub-optimal Clustering



Importance of Choosing Initial Centroids



KMeans

- *cluster*.KMeans([*n_clusters*], [*init*], [*n_init*], [*max_iter*], [*tol*], [*precompute_distances*], [*random_state*], [*algorithm*], ...)
  - K-Means clustering
  - *n_clusters*: number of clusters to form
  - *init*: 'k-means++', 'random', or user-provided. 'k-means++' for smart init. (default)
  - *n_init*: number of runs with different centroid seeds (default: 10)
  - *max_iter*: max number of iterations for a single run (default: 300)
  - *tol*: relative tolerance with regards to inertia to declare convergence (default: 1e-4)
  - *precompute_distances*: 'auto', True, or False (default: 'auto')
  - *random_state*: random number seed or generator
  - *algorithm*: 'auto', 'full', or 'elkan' (default: 'auto')

fit() and predict()

- *cluster*.`KMeans.fit`(*X*)
  - Compute Kmeans clustering
  - *X*: training instances to cluster

- *cluster*.`KMeans.predict`(*X*)
  - Predict the closest cluster each sample in *X* belongs to
  - *X*: new data to predict

Iris: Loading and Data Preprocessing

- Use only 3rd and 4th column values (i.e., petal length and petal width)

```python
%matplotlib inline
import numpy as np
import matplotlib as plt
from sklearn import datasets, cluster

# Load the iris dataset
iris = datasets.load_iris()

# We only take the 3rd & 4th features
iris_X = iris.data[:, 2:4]
```

```
array([[1.4, 0.2],
       [1.4, 0.2],
       [1.3, 0.2],
       [1.5, 0.2],
       [1.4, 0.2],
       ...])
```

iris_X

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

Iris: Clustering

- Learning k-means clustering model with Iris dataset

```python
# Create KMeans object
km = cluster.KMeans(n_clusters=3)

# Train the model using the training sets
km.fit(iris_X)

# Get the clustering result
labels = km.labels_
print(km.labels_)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 1 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 2 1 1 1 1
 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1
 1 1]
```

Iris: Plotting the Clustering Result

```
plt.scatter(iris_X[:, 0], iris_X[:, 1], c=labels)
```