Machine Learning: Algorithms and Applications

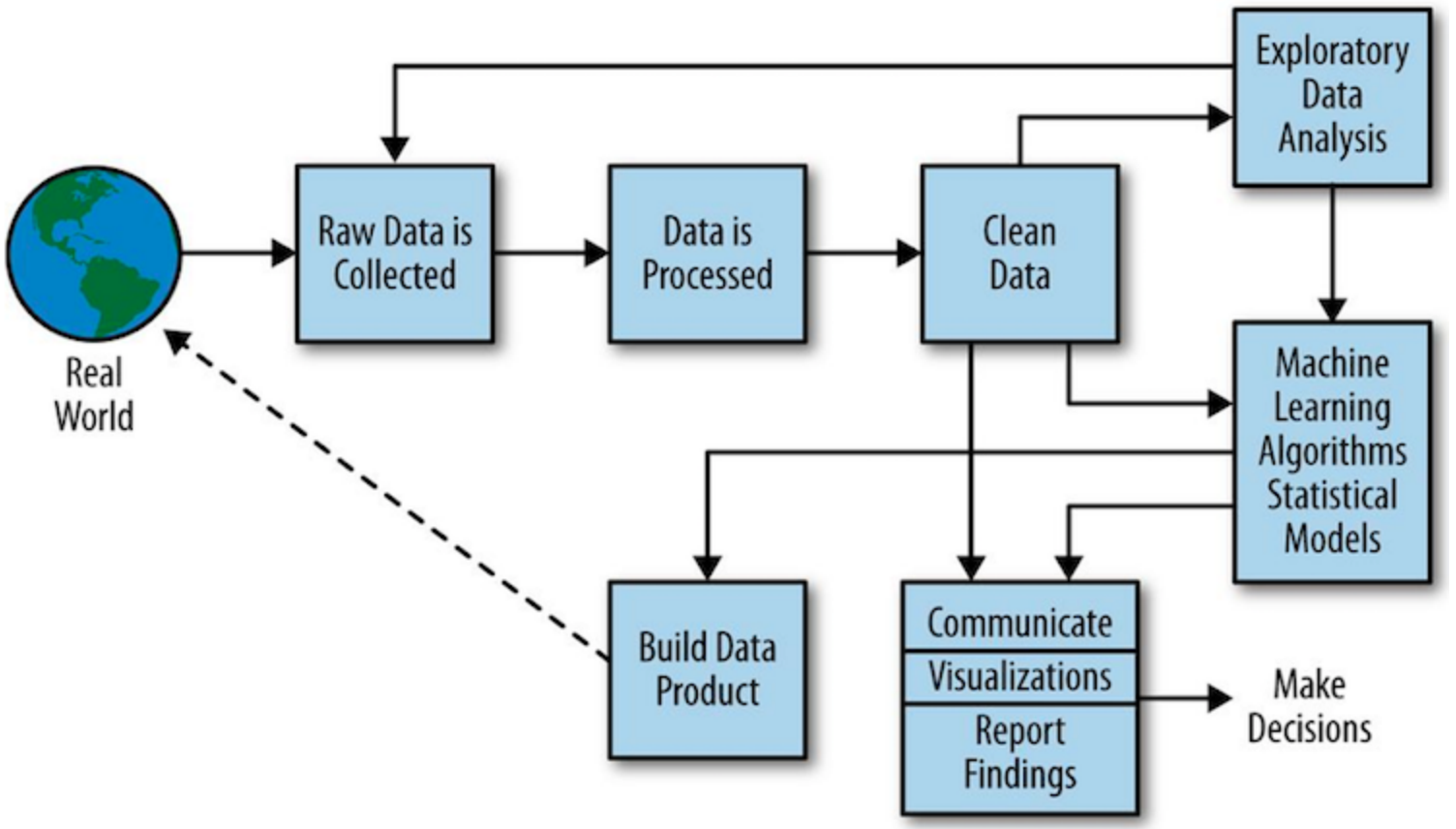
Day 1

Agenda - Day 1

- Introduction
- Python Numerics
- Pandas
- Data-Driven
- SciKit-Learn

Introduction

Data Science Pipeline



"Doing Data Science"

Big Concepts

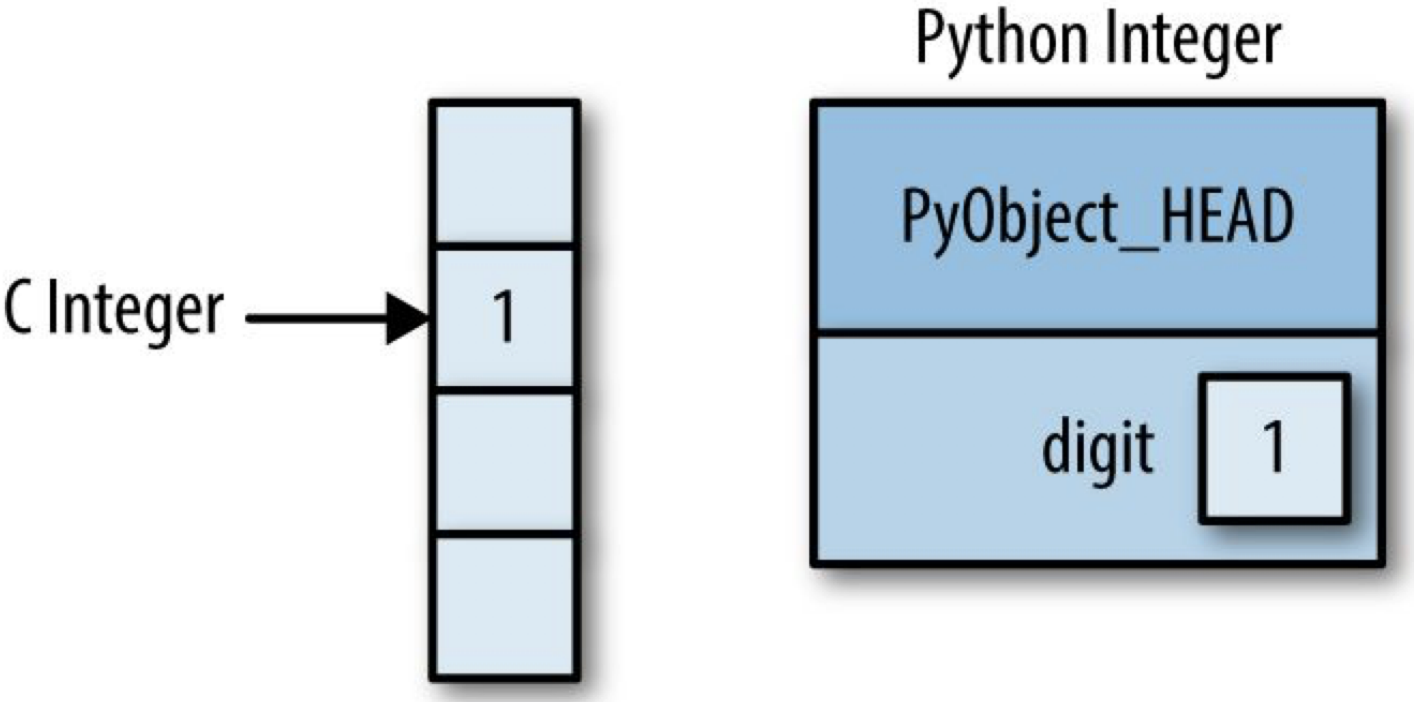
- Data as an Asset
- Role of Context
- Bad Data + ML = No Good
- Data Familiarity
- Curse of Dimensionality

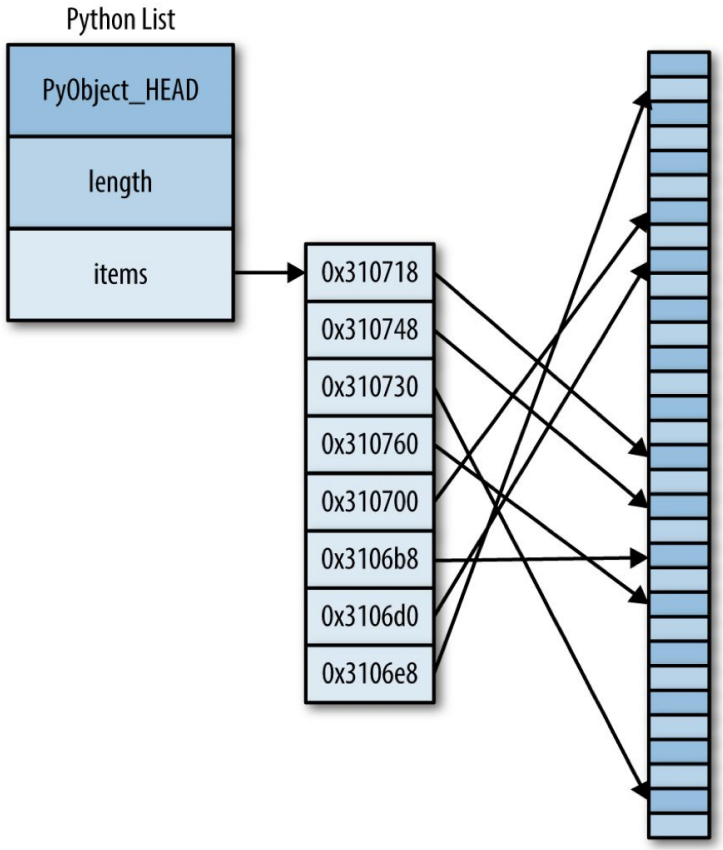
Exercise 1: Homework Review

Demo: Python Numerics

Python Numerics

- **`numpy`** is the de facto numerical toolkit for Python (http://www.numpy.org)
- there are screenshots in this presentation to maintain continuity, but let's open the notebook named **`Demo - Python Numerics.ipynb`** and go through it together
- when done, click here to skip screesnhots


Python Integer
PyObject_HEAD
digit
1
C Integer
1



"Python Data Science Handbook"

```
import numpy as np
np.random.seed(0)

def compute_reciprocals(values):
  output = np.empty(len(values))
  for i in range(len(values)):
    output[i] = 1.0 / values[i]

  return output

values = np.random.randint(1, 10, size = 5)
compute_reciprocals(values)
```

```
Out[1]: array([ 0.16666667, 1. , 0.25 , 0.25 , 0.125 ])
```

```
In[2]: big_array = np.random.randint(1, 100, size = 1000000)
       %timeit compute_reciprocals(big_array)

1 loop, best of 3: 2.91 s per loop
```

http://www.numpy.org

```
import numpy as np
```

```
>>> x = 2.345
```

```
>>> x - 2.345
0.0
```

```
>>> x - 2.345 == 0.0
True
```

np.trunc()

- nearest integer i which is closer to zero than x is

```
>>> np.trunc(x)
2.0
```

np.floor()

- the largest integer i, such that i <= x

```
>>> np.floor(x)
2.0
```

```
>>> np.floor(2.01)
2.0
```

```
>>> np.floor(2.00)
2.0
```

np.ceil()

- the smallest integer i, such that i >= x

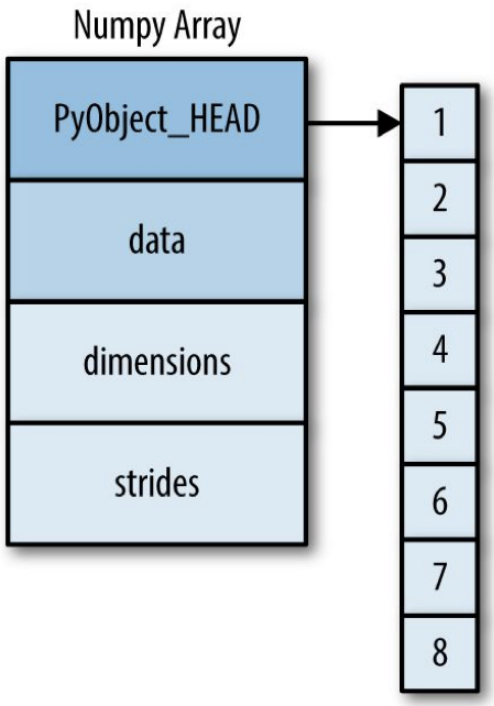
```
>>> np.ceil(x)
3.0
```

```
>>> np.ceil(2.01)
3.0
```

```
>>> np.ceil(2.0)
2.0
```

```
>>> np.ceil(x) - 1
2.0
```

```
>>> np.ceil(2.01) - 1
2.0
```



"Python Data Science Handbook"

```
>>> import numpy as np
>>> np.array([1, 4, 2, 5, 3])
array([1, 4, 2, 5, 3])
```

```
>>> np.array([3.14, 4, 2, 3])
array([ 3.14,  4.  ,  2.  ,  3.  ])
```

```
>>> np.array([1, 2, 3, 4], dtype='float32')
array([ 1.,  2.,  3.,  4.], dtype=float32)
```

```
>>> np.array([range(i, i + 3) for i in [2, 4, 6]])
array([[2, 3, 4],
       [4, 5, 6],
       [6, 7, 8]])
```

```
>>> np.zeros(10, dtype=int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
>>> np.ones((3, 5), dtype=float)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

```
>>> np.full((3, 5), 3.14)
array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14]])
```

```
>>> np.arange(0, 20, 2)
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
>>> np.linspace(0, 1, 5)
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

```
>>> np.random.random((3, 3))
array([[ 0.33687321,  0.57661167,  0.77955889],
       [ 0.7441428 ,  0.53766224,  0.62966105],
       [ 0.27348138,  0.27234265,  0.71558399]])
```

```
>>> np.random.normal(0, 1, (3, 3))
array([[-2.02151788, -0.04433702, -0.16415692],
       [ 0.28498092,  0.4638266 ,  0.64989753],
       [-0.93943869, -1.90040282, -0.94711293]])
```

```
>>> np.random.randint(0, 10, (3, 3))
array([[7, 3, 0],
       [3, 3, 4],
       [9, 2, 6]])
```

```
>>> np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Converting array types

```
>>> x = np.linspace(0,10,50)
>>> x
array([  0.        ,   0.20408163,   0.40816327,   0.6122449 ,
         0.81632653,   1.02040816,   1.2244898 ,   1.42857143,
         1.63265306,   1.83673469,   2.04081633,   2.24489796,
         2.44897959,   2.65306122,   2.85714286,   3.06122449,
         3.26530612,   3.46938776,   3.67346939,   3.87755102,
         4.08163265,   4.28571429,   4.48979592,   4.69387755,
         4.89795918,   5.10204082,   5.30612245,   5.51020408,
         5.71428571,   5.91836735,   6.12244898,   6.32653061,
         6.53061224,   6.73469388,   6.93877551,   7.14285714,
         7.34693878,   7.55102041,   7.75510204,   7.95918367,
         8.16326531,   8.36734694,   8.57142857,   8.7755102 ,
         8.97959184,   9.18367347,   9.3877551 ,   9.59183673,
         9.79591837,  10.        ])
```

```
>>> x.astype(int)
array([ 0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  3,  3,
        3,  3,  3,  4,  4,  4,  4,  4,  5,  5,  5,  5,  5,  6,  6,  6,  6,
        6,  7,  7,  7,  7,  7,  8,  8,  8,  8,  8,  9,  9,  9,  9, 10])
```

Multi-dimensional Arrays

```
>>> x2 = np.random.randint(10, size=(3, 4))
array([[3, 3, 0, 9],
       [0, 9, 7, 0],
       [5, 6, 2, 5]])
```

```
>>> x2[0, 0]
3
>>> x2[2, 0]
5
>>> x2[2, -1]
5
```

```
>>> x2[0, 0] = 12
>>> x2
array([[12,  3,  0,  9],
       [ 0,  9,  7,  0],
       [ 5,  6,  2,  5]])
```

```
>>> np.arange(0,9).reshape(3,3)
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Numpy Array Slicing

```
>>> x = np.arange(10)
>>> x[:5]
array([0, 1, 2, 3, 4])
>>> x[5:]
array([5, 6, 7, 8, 9])
```

```
>>> x[4:7]
array([4, 5, 6])
```

```
>>> x[::2]
array([0, 2, 4, 6, 8])
```

```
>>> x[1::2]
array([1, 3, 5, 7, 9])
```

```
>>> x[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

```
>>> x[5::-2]
array([5, 3, 1])
```

Filtering 1 dimensional data

```
>>> x = np.array([ 1, 1, 2, 1, 0, 3, 0, 0 ])
```

```
>>> np.nonzero(x)
(array([0, 1, 2, 3, 5]),)
```

```
>>> x[np.nonzero(x)]
array([1, 1, 2, 1, 3])
```

```
>>> x[np.nonzero(x < 3)]
array([1, 1, 2, 1, 0, 0, 0])
```

Filtering 2-dimensional data

```
>>> x = np.array([[1,0,0], [0,2,0], [1,1,0]])
>>> x
array([[1, 0, 0],
       [0, 2, 0],
       [1, 1, 0]])
```

```
>>> np.nonzero(x)
(array([0, 1, 2, 2]), array([0, 1, 0, 1]))
```

```
>>> x[ np.nonzero(x) ]
array([1, 2, 1, 1])
```

```
>>> np.transpose(np.nonzero(x))
array([[0, 0],
       [1, 1],
       [2, 0],
       [2, 1]])
```

Multi-dimensional Subarrays

```
>>> x2
array([[12,  3,  0,  9],
       [ 0,  9,  7,  0],
       [ 5,  6,  2,  5]])
```

```
>>> x2[:2, :3]
array([[12,  3,  0],
       [ 0,  9,  7]])
```

```
>>> x2[:3, ::2]
array([[12,  0],
       [ 0,  7],
       [ 5,  2]])
```

```
>>> x2[::-1, ::-1]
array([[ 5,  2,  6,  5],
       [ 0,  7,  9,  0],
       [ 9,  0,  3, 12]])
```

Subarray Views

```
>>> x2
array([[12,  3,  0,  9],
       [ 0,  9,  7,  0],
       [ 5,  6,  2,  5]])
```

```
>>> x2_sub = x2[:2, :2]
>>> x2_sub
array([[12,  3],
       [ 0,  9]])
```

```
>>> x2_sub[0, 0] = 99
>>> x2_sub
array([[99,  3],
       [ 0,  9]])
```

```
>>> x2
array([[99,  3,  0,  9],
       [ 0,  9,  7,  0],
       [ 5,  6,  2,  5]])
```

Vectorized Operations

```
big_array = np.random.randint(1, 100, size=1000000)
```

```
In [2]: big_array = np.random.randint(1, 100, size=1000000)
   ...: %timeit (1.0 / big_array)
   ...:
100 loops, best of 3: 3.51 ms per loop
```

```
In [4]: big_array = np.random.rand(1000000)
   ...: %timeit sum(big_array)
   ...: %timeit np.sum(big_array)
10 loops, best of 3: 85.2 ms per loop
1000 loops, best of 3: 400 µs per loop
```

```
>>> x = np.arange(9).reshape((3, 3))
>>> 2 ** x
array([[  1,   2,   4],
       [  8,  16,  32],
       [ 64, 128, 256]])
```

```
>>> x = np.arange(4)
>>> -(0.5*x + 1) ** 2
array([-1.  , -2.25, -4.  , -6.25])
```

Operator	ufunc	Description
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

Trigonometric ufuncs

```
>>> theta = np.linspace(0, np.pi, 3)
>>> theta
array([ 0.        ,  1.57079633,  3.14159265])
>>> np.sin(theta)
array([  0.00000000e+00,   1.00000000e+00,   1.22464680e-16])
>>> np.cos(theta)
array([  1.00000000e+00,   6.12323400e-17,  -1.00000000e+00])
>>> np.tan(theta)
array([  0.00000000e+00,   1.63312394e+16,  -1.22464680e-16])
>>> x         =  [-1, 0, 1]
>>> np.arcsin(x)
array([-1.57079633,  0.        ,  1.57079633])
>>> np.arccos(x)
array([ 3.14159265,  1.57079633,  0.        ])
>>> np.arctan(x)
array([-0.78539816,  0.        ,  0.78539816])
```

Exponent and Logarithm ufuncs

```
>>> x = [1, 2, 3]
>>> np.exp(x)
array([  2.71828183,   7.3890561 ,  20.08553692])
>>> np.exp2(x)
array([ 2.,  4.,  8.])
>>> np.power(3, x)
array([ 3,  9, 27])
>>> x = [1, 2, 4, 10]
>>> np.log(x)
array([ 0.        ,  0.69314718,  1.38629436,  2.30258509])
>>> np.log2(x)
array([ 0.        ,  1.        ,  2.        ,  3.32192809])
>>> np.log10(x)
array([ 0.        ,  0.30103   ,  0.60205999,  1.        ])
```

Aggregate ufuncs

```
>>> x = np.arange(1, 6)
>>> np.add.reduce(x)
15
>>> np.multiply.reduce(x)
120
>>> np.add.accumulate(x)
array([ 1,  3,  6, 10, 15])
>>> np.multiply.accumulate(x)
array([  1,   2,   6,  24, 120])
```

Exercise: Python Numerics

(open the notebook named **`Exercise - Python Numerics.ipynb`**)

Demo: Pandas

Pandas

- **`pandas`** is the de facto data analysis toolkit for Python (http://pandas.pydata.org)
- there are screenshots in this presentation to maintain continuity, but let's open the notebook named **`Demo - Pandas.ipynb`** and go through it together
- when done, click here to skip screesnhots

Pandas Series

```
>>> import pandas as pd
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0])
>>> data
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

```
>>> data.values
array([ 0.25,  0.5 ,  0.75,  1.  ])
>>> data.index
RangeIndex(start=0, stop=4, step=1)
```

```
>>> data[1]
0.5
```

```
>>> data[1:3]
1    0.50
2    0.75
dtype: float64
```

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
```

```
>>> data
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

```
>>> data['c']
0.75
```

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])
```

```
>>> data
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

```
>>> data[5]
0.5
```

Implicit and Explicit Indexing

```
>>> data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
>>> data
1    a
3    b
5    c
dtype: object
```

```
>>> data[1]
'a'
```

```
>>> data[1:3]
3    b
5    c
dtype: object
```

loc and iloc

```
>>> data
1    a
3    b
5    c
dtype: object
```

```
>>> data.loc[1]
'a'
>>> data.loc[1:3]
1    a
3    b
dtype: object
```

```
>>> data.iloc[1]
'b'
>>> data.iloc[1:3]
3    b
5    c
dtype: object
```

Python dicts as Series

```
>>> population_dict = {'California': 38332521,
                       'Texas': 26448193,
                       'New York': 19651127,
                       'Florida': 19552860,
                       'Illinois': 12882135}
>>> population = pd.Series(population_dict)
```

```
>>> population
California    38332521
Florida       19552860
Illinois      12882135
New York      19651127
Texas         26448193
dtype: int64
```

```
>>> population['California']
38332521
```

```
>>> population['California':'Illinois']
California    38332521
Florida       19552860
Illinois      12882135
dtype: int64
```

Pandas DataFrame

```
>>> area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
                 'Florida': 170312, 'Illinois': 149995}
>>> area = pd.Series(area_dict)
>>> area
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
dtype: int64
```

```
>>> states = pd.DataFrame({'population': population,
                           'area': area})
>>> states
              area  population
California  423967    38332521
Florida     170312    19552860
Illinois    149995    12882135
New York    141297    19651127
Texas       695662    26448193
```

```
>>> states.index
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'],
  dtype='object')
```

```
>>> states.columns
Index(['area', 'population'], dtype='object')
```

```
>>> states['area']
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

```
>>> states.values
array([[  423967, 38332521],
       [  170312, 19552860],
       [  149995, 12882135],
       [  141297, 19651127],
       [  695662, 26448193]])
```

Sales Data

```
>>> dat = pd.read_csv("data/WA_Fn-UseC_-Sales-Win-Loss.csv")
>>> dat.columns
Index(['Opportunity Number', 'Supplies Subgroup', 'Supplies Group', 'Region',
       'Route To Market', 'Elapsed Days In Sales Stage', 'Opportunity Result',
       'Sales Stage Change Count', 'Total Days Identified Through Closing',
       'Total Days Identified Through Qualified', 'Opportunity Amount USD',
       'Client Size By Revenue', 'Client Size By Employee Count',
       'Revenue From Client Past Two Years', 'Competitor Type',
       'Ratio Days Identified To Total Days',
       'Ratio Days Validated To Total Days',
       'Ratio Days Qualified To Total Days', 'Deal Size Category'],
      dtype='object')
```

```
>>> dat['Opportunity Result']
0         Won
1        Loss
2         Won
3        Loss
4        Loss
5        Loss
6         Won
7        Loss
...
```

Counting Values

```
>>> dat['Opportunity Result'].value_counts()
Loss    60398
Won     17627
Name: Opportunity Result, dtype: int64
```

```
>>> dat['Supplies Group'].value_counts()
Car Accessories           49810
Performance & Non-auto    27325
Tires & Wheels              609
Car Electronics             281
Name: Supplies Group, dtype: int64
```

```
>>> dat['Elapsed Days In Sales Stage'].value_counts()
16     5010
44     2388
62     1738
7      1629
23     1455
37     1412
45     1238
24     1233
35     1226
...
```

Top Five values

```
>>> dat['Supplies Subgroup'].value_counts()[:5]
Motorcycle Parts           15174
Exterior Accessories       13876
Garage & Car Care           9733
Shelters & RV               9606
Batteries & Accessories     9192
Name: Supplies Subgroup, dtype: int64
```

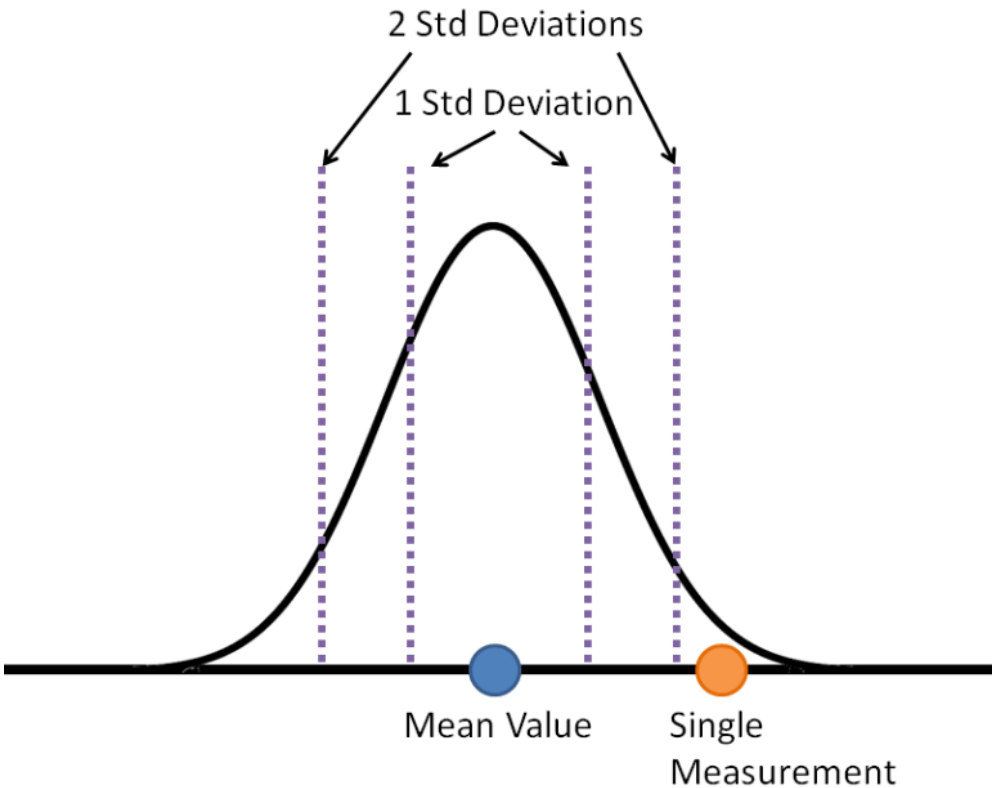
Extracting Columns

```
>>> region_results = dat[["Region", "Opportunity Result"]]
>>> region_results.shape
(78025, 2)
>>> region_results.head()
0         Northwest                Won
1           Pacific               Loss
2           Pacific                Won
3               NaN               Loss
4           Pacific               Loss
```

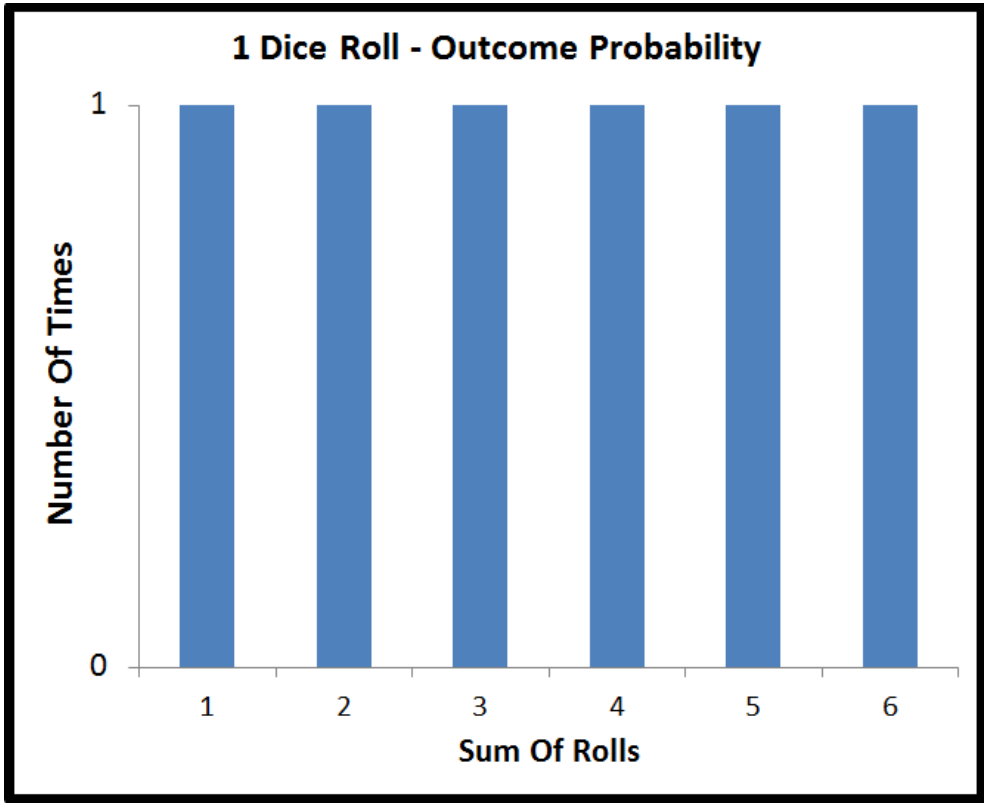
Exercise: Pandas

(open the notebook named **`Exercise - Pandas.ipynb`**)

Data-Driven



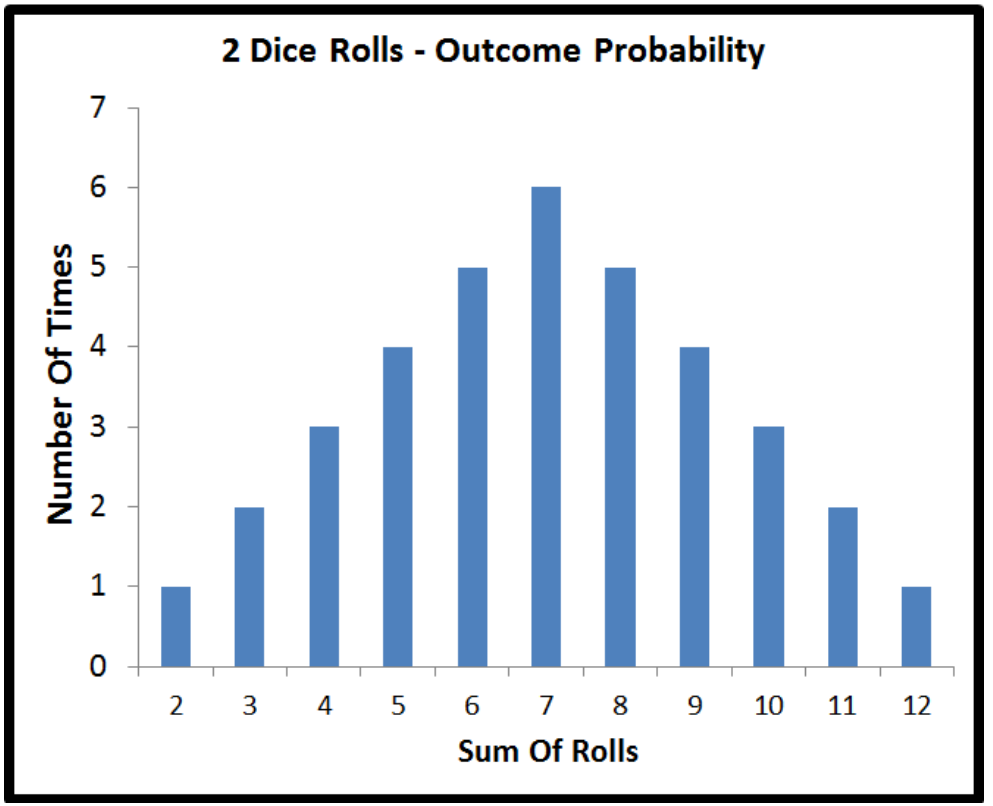
http://www.fairlynerdy.com/intuitive-guide-statistical-significance/



No way to predict the outcome of a single die roll
(i.e., no outcome is more likely than any other)

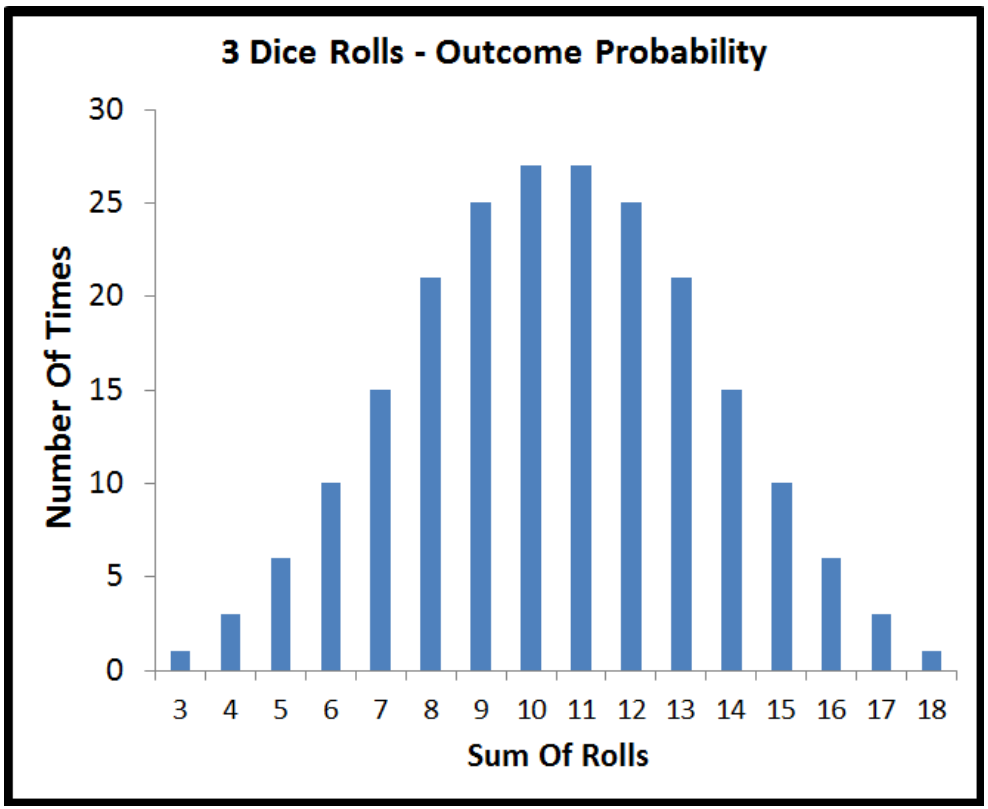
http://www.fairlynerdy.com/intuitive-guide-statistical-significance/

CC BY 2.0, https://www.flickr.com/photos/andereri/8503769663

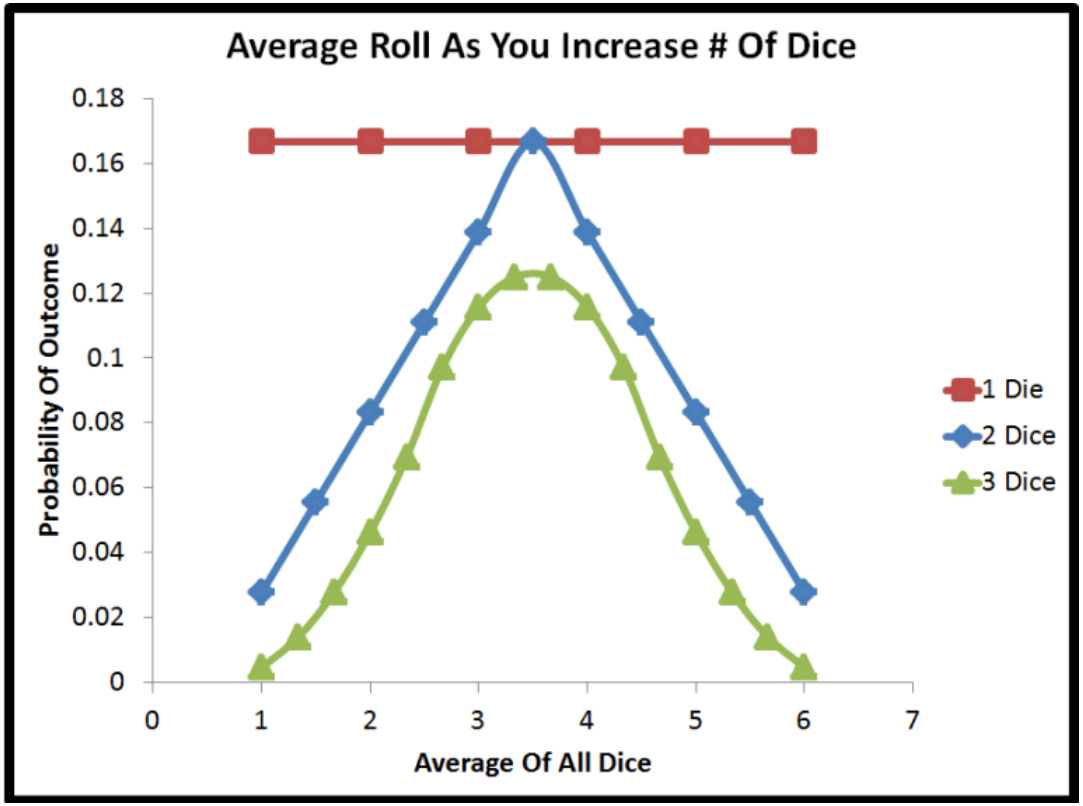


$6 \times 6 = 36$ combinations, but only 11 values
i.e., all of a sudden we can start to make predictions about the sum of 2 dice

http://www.fairlynerdy.com/intuitive-guide-statistical-significance/

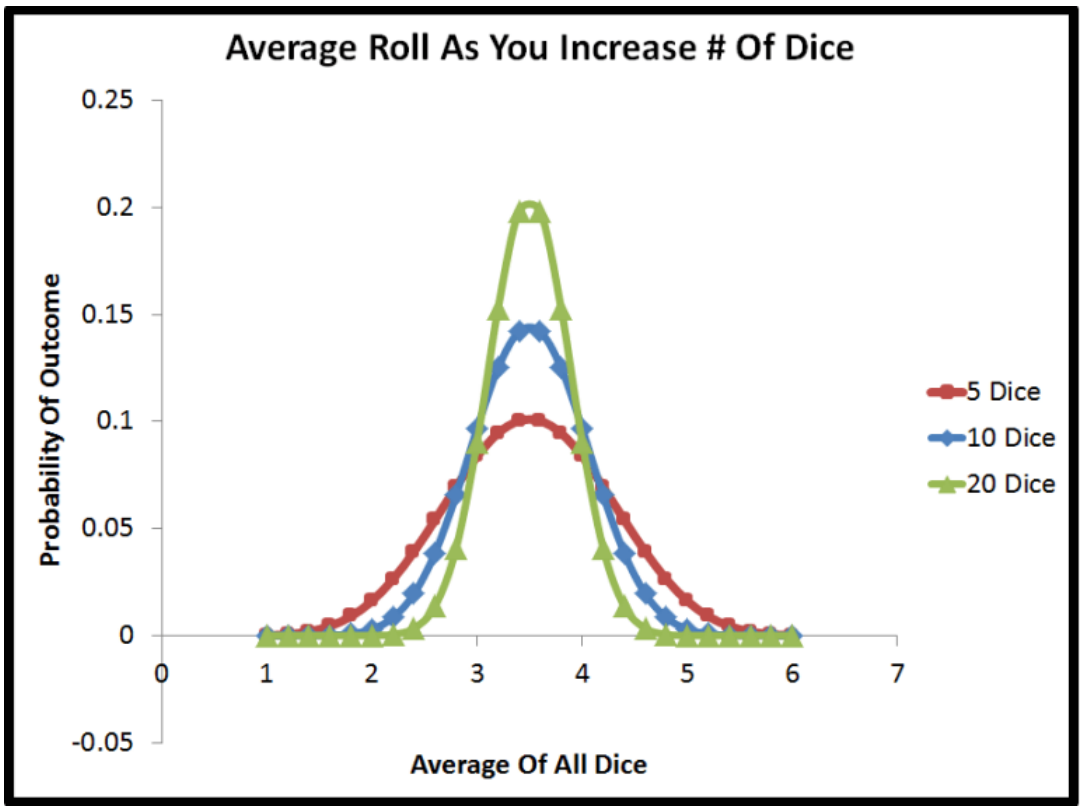


http://www.fairlynerdy.com/intuitive-guide-statistical-significance/

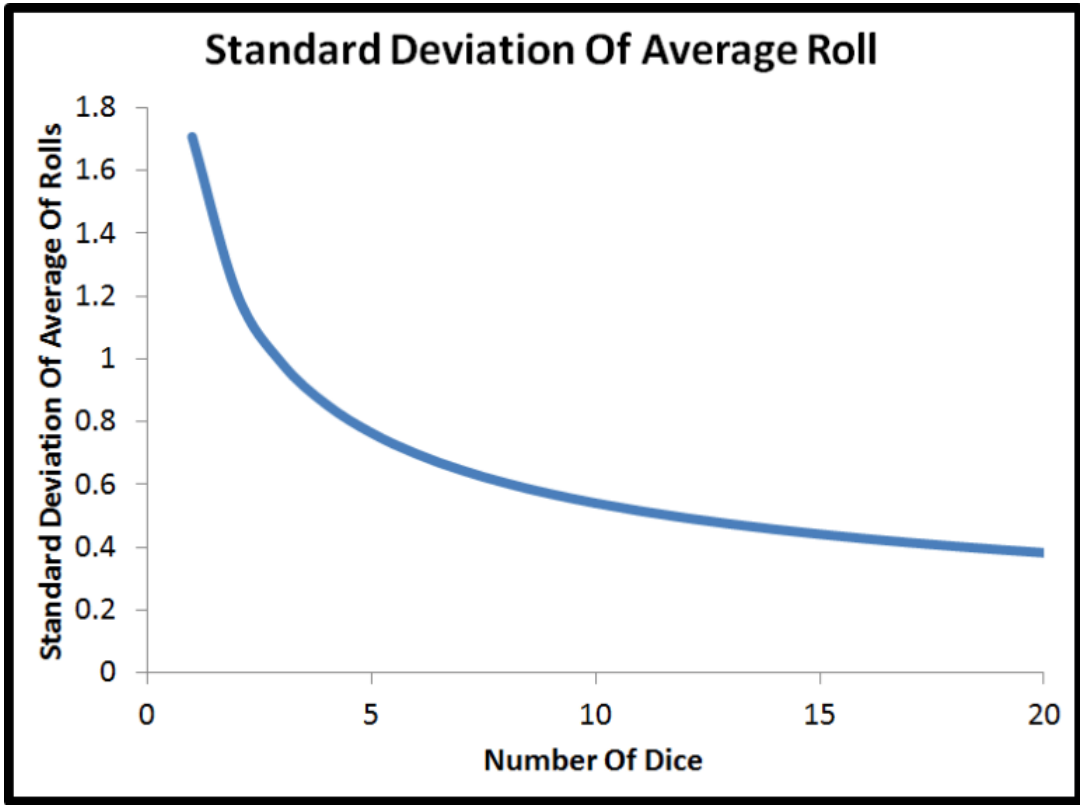


- by making multiple measurements, we can be more confident of their average value than we can in the result of any single measurement
- range of possible outcomes for the average is same as range of possible outcomes for a single roll, but distribution is more clustered in the center
- standard deviation of the mean of multiple measurements is lower than standard deviation of a single measurement

http://www.fairlynerdy.com/intuitive-guide-statistical-significance/

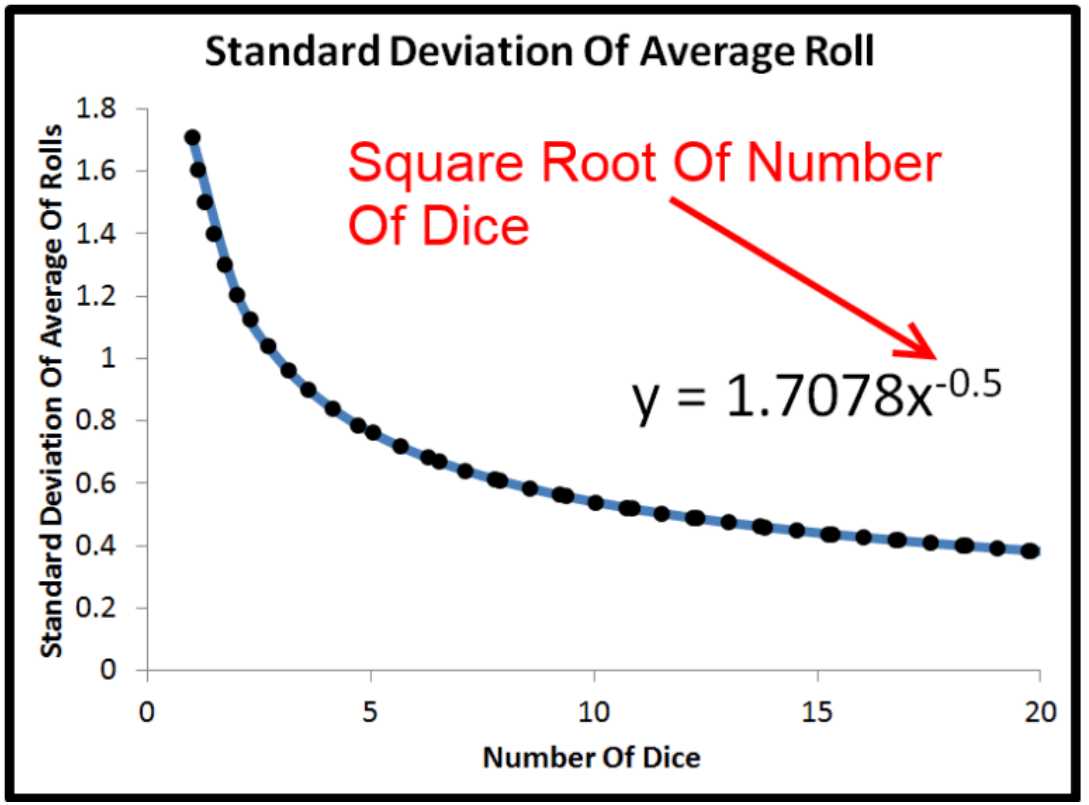


http://www.fairlynerdy.com/intuitive-guide-statistical-significance/




```
>>> import numpy as np
# compute standard deviation of one die roll
>>> np.std((1, 2, 3, 4, 5, 6))
1.707825127659933
```

http://www.fairlynerdy.com/intuitive-guide-statistical-significance/



http://www.fairlynerdy.com/intuitive-guide-statistical-significance/

$$y = 1.7078x^{-0.5}$$

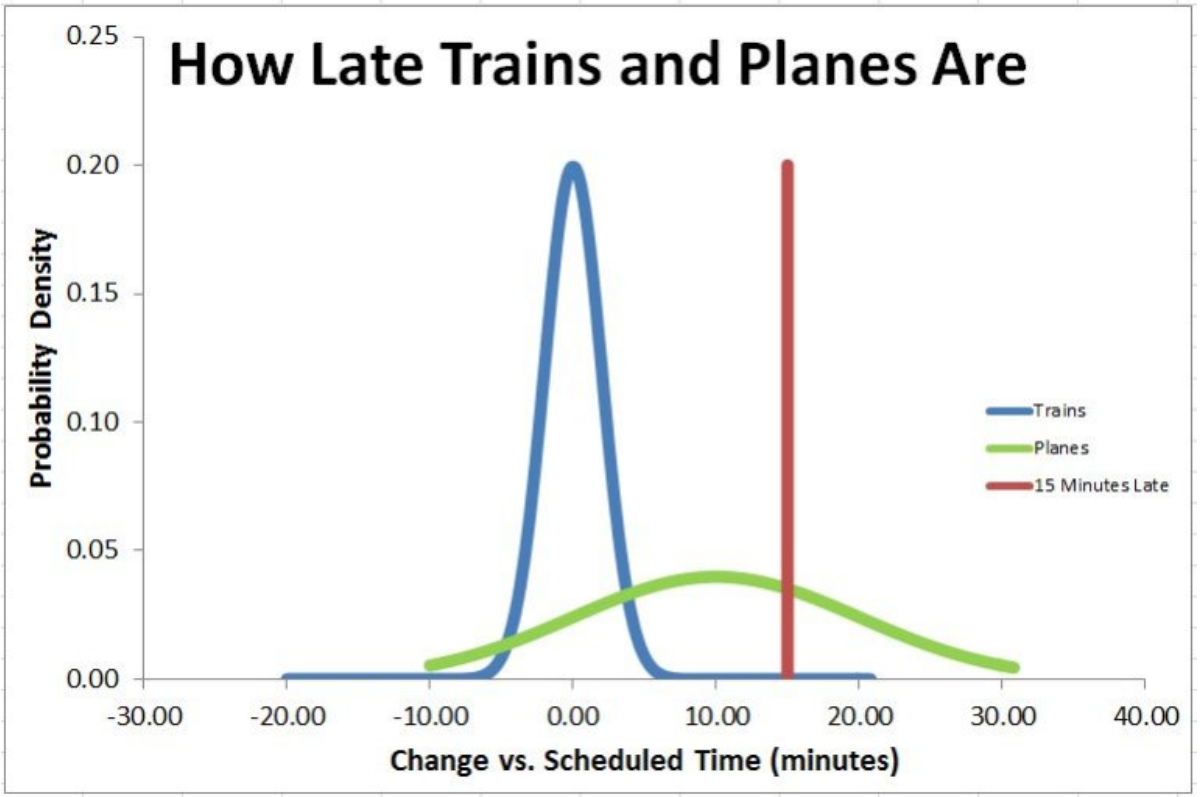
$$y = \frac{1.7078}{\sqrt{x}}$$

$$y = \frac{\sigma}{\sqrt{n}}$$

Z Test

$$Z = \frac{\bar{x} - \mu_0}{\frac{\sigma}{\sqrt{n}}}$$

Variable	Meaning
$\bar{x}$	sample mean
μ_0	population mean
σ	population standard deviation
n	test sample size



http://www.fairlynerdy.com/understanding-statistical-significance

Should we send our kid to an SAT prep course amid claims the course will increase your kid's score?

$$Z = \frac{\bar{x} - \mu_0}{\frac{\sigma}{\sqrt{n}}}$$

```
>>> import numpy as np
>>> import math
>>> mu = 500 # population mean
>>> sigma = 100 # population standard deviation
```

```
>>> samples = [434, 694, 457, 534, 720, 400, 484, 478, 610, 641, 425, 636,
               454, 514, 563, 370, 499, 640, 501, 625, 612, 471, 598, 509, 531]
>>> len(samples)
25
>>> np.mean(samples)
536.0
>>> xbar = np.mean(samples)
```

```
>>> z_score = (xbar - mu) / (sigma / math.sqrt(len(samples)))
>>> z_score
1.8
```

Standard Normal Probabilities

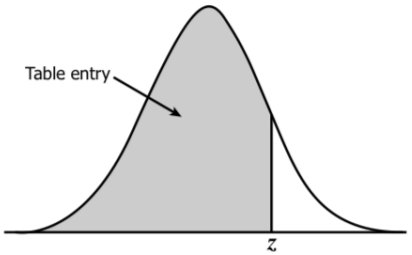


Table entry for z is the area under the standard normal curve to the left of z.

z	.00	.01	.02	.03	.04	.05	.06	.07	.08	.09
0.0	.5000	.5040	.5080	.5120	.5160	.5199	.5239	.5279	.5319	.5359
0.1	.5398	.5438	.5478	.5517	.5557	.5596	.5636	.5675	.5714	.5753
0.2	.5793	.5832	.5871	.5910	.5948	.5987	.6026	.6064	.6103	.6141
0.3	.6179	.6217	.6255	.6293	.6331	.6368	.6406	.6443	.6480	.6517
0.4	.6554	.6591	.6628	.6664	.6700	.6736	.6772	.6808	.6844	.6879
0.5	.6915	.6950	.6985	.7019	.7054	.7088	.7123	.7157	.7190	.7224
0.6	.7257	.7291	.7324	.7357	.7389	.7422	.7454	.7486	.7517	.7549
0.7	.7580	.7611	.7642	.7673	.7704	.7734	.7764	.7794	.7823	.7852
0.8	.7881	.7910	.7939	.7967	.7995	.8023	.8051	.8078	.8106	.8133
0.9	.8159	.8186	.8212	.8238	.8264	.8289	.8315	.8340	.8365	.8389
1.0	.8413	.8438	.8461	.8485	.8508	.8531	.8554	.8577	.8599	.8621
1.1	.8643	.8665	.8686	.8708	.8729	.8749	.8770	.8790	.8810	.8830
1.2	.8849	.8869	.8888	.8907	.8925	.8944	.8962	.8980	.8997	.9015
1.3	.9032	.9049	.9066	.9082	.9099	.9115	.9131	.9147	.9162	.9177
1.4	.9192	.9207	.9222	.9236	.9251	.9265	.9279	.9292	.9306	.9319
1.5	.9332	.9345	.9357	.9370	.9382	.9394	.9406	.9418	.9429	.9441
1.6	.9452	.9463	.9474	.9484	.9495	.9505	.9515	.9525	.9535	.9545
1.7	.9554	.9564	.9573	.9582	.9591	.9599	.9608	.9616	.9625	.9633
1.8	.9641	.9649	.9656	.9664	.9671	.9678	.9686	.9693	.9699	.9706
1.9	.9713	.9719	.9726	.9732	.9738	.9744	.9750	.9756	.9761	.9767

Standard Normal Probabilities

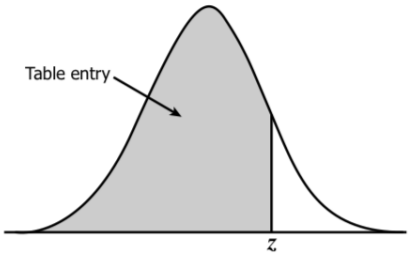


Table entry for z is the area under the standard normal curve to the left of z.

z	.00	.01	.02	.03	.04	.05	.06	.07	.08	.09
0.0	.5000	.5040	.5080	.5120	.5160	.5199	.5239	.5279	.5319	.5359
0.1	.5398	.5438	.5478	.5517	.5557	.5596	.5636	.5675	.5714	.5753
0.2	.5793	.5832	.5871	.5910	.5948	.5987	.6026	.6064	.6103	.6141
0.3	.6179	.6217	.6255	.6293	.6331	.6368	.6406	.6443	.6480	.6517
0.4	.6554	.6591	.6628	.6664	.6700	.6736	.6772	.6808	.6844	.6879
0.5	.6915	.6950	.6985	.7019	.7054	.7088	.7123	.7157	.7190	.7224
0.6	.7257	.7291	.7324	.7357	.7389	.7422	.7454	.7486	.7517	.7549
0.7	.7580	.7611	.7642	.7673	.7704	.7734	.7764	.7794	.7823	.7852
0.8	.7881	.7910	.7939	.7967	.7995	.8023	.8051	.8078	.8106	.8133
0.9	.8159	.8186	.8212	.8238	.8264	.8289	.8315	.8340	.8365	.8389
1.0	.8413	.8438	.8461	.8485	.8508	.8531	.8554	.8577	.8599	.8621
1.1	.8643	.8665	.8686	.8708	.8729	.8749	.8770	.8790	.8810	.8830
1.2	.8849	.8869	.8888	.8907	.8925	.8944	.8962	.8980	.8997	.9015
1.3	.9032	.9049	.9066	.9082	.9099	.9115	.9131	.9147	.9162	.9177
1.4	.9192	.9207	.9222	.9236	.9251	.9265	.9279	.9292	.9306	.9319
1.5	.9332	.9345	.9357	.9370	.9382	.9394	.9406	.9418	.9429	.9441
1.6	.9452	.9463	.9474	.9484	.9495	.9505	.9515	.9525	.9535	.9545
1.7	.9554	.9564	.9573	.9582	.9591	.9599	.9608	.9616	.9625	.9633
1.8	.9641	.9649	.9656	.9664	.9671	.9678	.9686	.9693	.9699	.9706
1.9	.9713	.9719	.9726	.9732	.9738	.9744	.9750	.9756	.9761	.9767

Standard Normal Probabilities

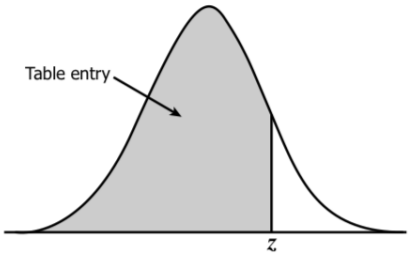


Table entry for z is the area under the standard normal curve to the left of z.

z	.00	.01	.02	.03	.04	.05	.06	.07	.08	.09
0.0	.5000	.5040	.5080	.5120	.5160	.5199	.5239	.5279	.5319	.5359
0.1	.5398	.5438	.5478	.5517	.5557	.5596	.5636	.5675	.5714	.5753
0.2	.5793	.5832	.5871	.5910	.5948	.5987	.6026	.6064	.6103	.6141
0.3	.6179	.6217	.6255	.6293	.6331	.6368	.6406	.6443	.6480	.6517
0.4	.6554	.6591	.6628	.6664	.6700	.6736	.6772	.6808	.6844	.6879
0.5	.6915	.6950	.6985	.7019	.7054	.7088	.7123	.7157	.7190	.7224
0.6	.7257	.7291	.7324	.7357	.7389	.7422	.7454	.7486	.7517	.7549
0.7	.7580	.7611	.7642	.7673	.7704	.7734	.7764	.7794	.7823	.7852
0.8	.7881	.7910	.7939	.7967	.7995	.8023	.8051	.8078	.8106	.8133
0.9	.8159	.8186	.8212	.8238	.8264	.8289	.8315	.8340	.8365	.8389
1.0	.8413	.8438	.8461	.8485	.8508	.8531	.8554	.8577	.8599	.8621
1.1	.8643	.8665	.8686	.8708	.8729	.8749	.8770	.8790	.8810	.8830
1.2	.8849	.8869	.8888	.8907	.8925	.8944	.8962	.8980	.8997	.9015
1.3	.9032	.9049	.9066	.9082	.9099	.9115	.9131	.9147	.9162	.9177
1.4	.9192	.9207	.9222	.9236	.9251	.9265	.9279	.9292	.9306	.9319
1.5	.9332	.9345	.9357	.9370	.9382	.9394	.9406	.9418	.9429	.9441
1.6	.9452	.9463	.9474	.9484	.9495	.9505	.9515	.9525	.9535	.9545
1.7	.9554	.9564	.9573	.9582	.9591	.9599	.9608	.9616	.9625	.9633
1.8	.9641	.9649	.9656	.9664	.9671	.9678	.9686	.9693	.9699	.9706
1.9	.9713	.9719	.9726	.9732	.9738	.9744	.9750	.9756	.9761	.9767

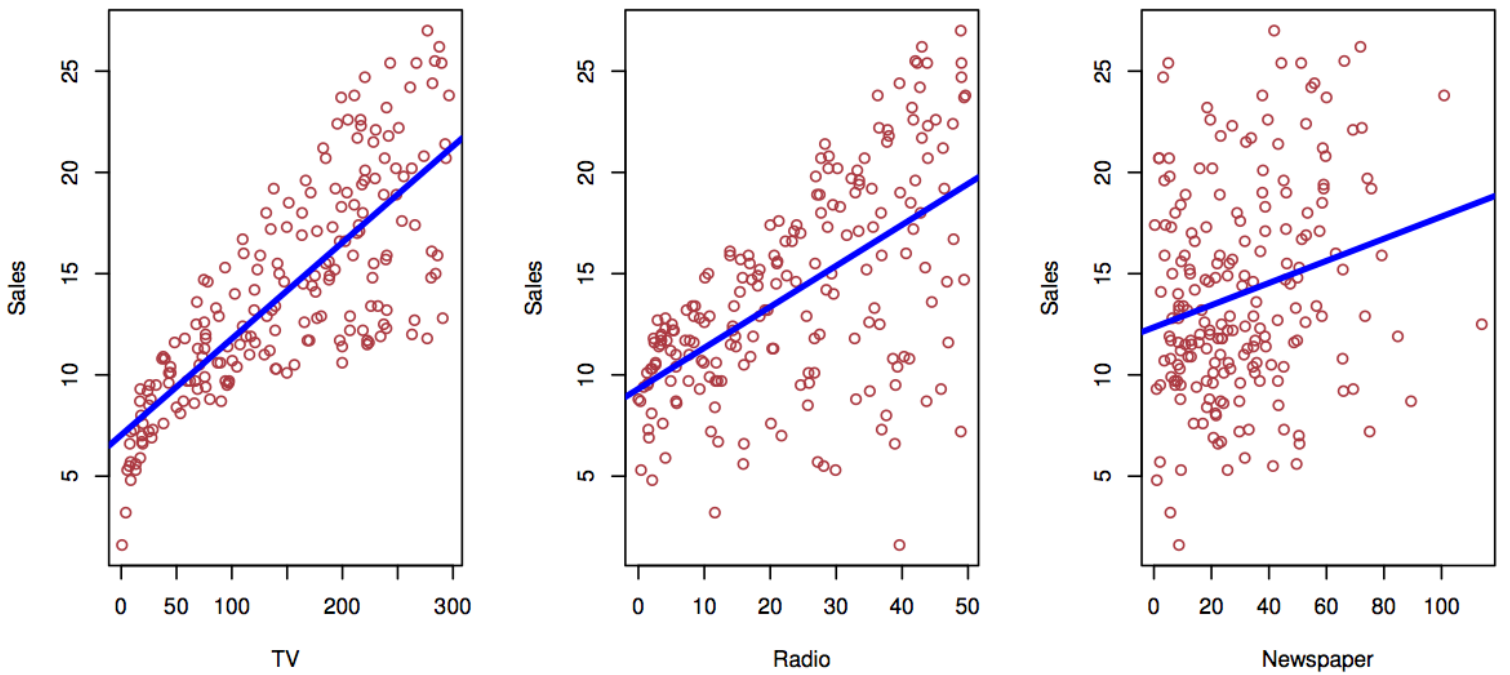
"In the past century, data collection often was carried out by statistical sampling since data was manually collected and therefore expensive. The prudent researcher would design the process to maximize the information that could be extracted from the data. In this century, data are arriving by firehose and without design."

"Algorithms for Data Science", Steele et al

"Since statistical science revolves around the analysis of relatively small and information-rich data, statistical philosophy and principles are at times marginally relevant to the task at hand."

"Steele, Brian; Chandler, John; Reddy, Swarna. Algorithms for Data Science (Kindle Locations 711-712). Springer International Publishing. Kindle Edition."

Advertising Effectiveness



"An Introduction to Statistical Learning", James et al

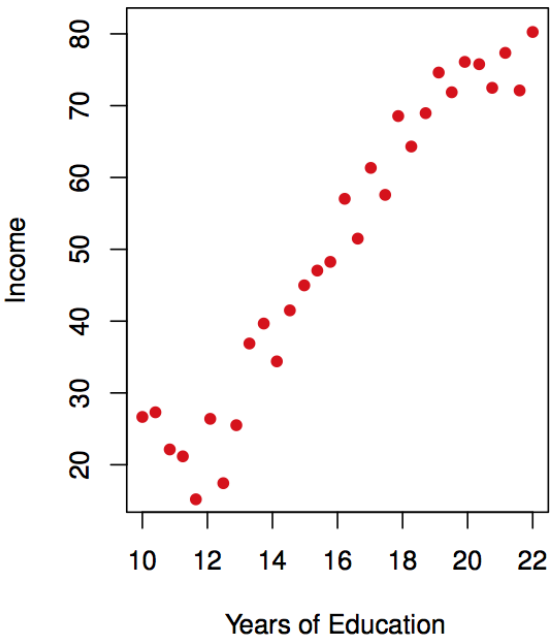
Reality

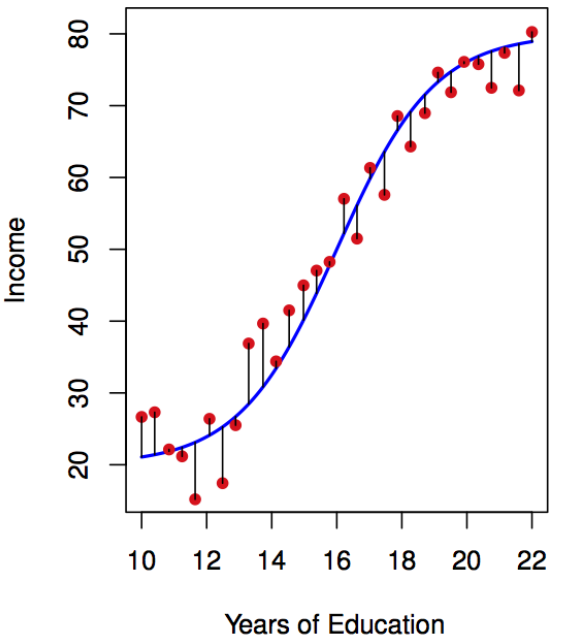
$$Y = f(X) + \epsilon$$

Some function maps the inputs to the output

Why is there an error term?

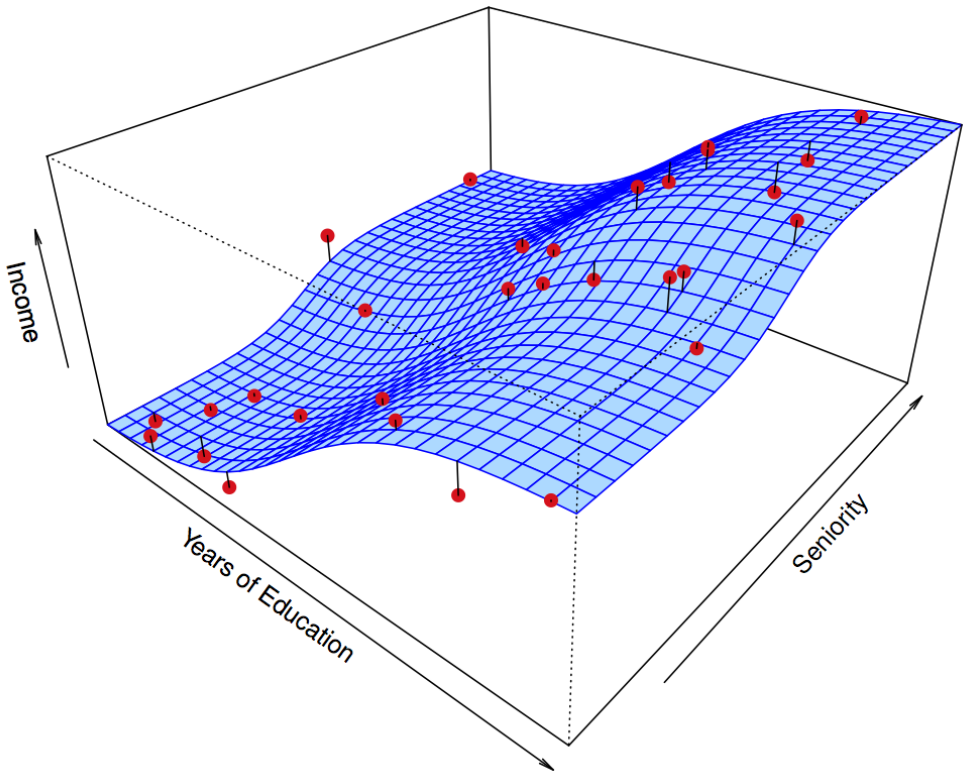
Does Income Relate to Years of Education?





"An Introduction to Statistical Learning", James et al

What if we add another dimension?



"An Introduction to Statistical Learning", James et al

Prediction

$$\hat{Y} = \hat{f}(X)$$

We build a model which approximates the true function

Why *isn't* there an error term here?

Types of Errors

- Reducible
  - How do we reduce this?
- Irreducible

Compare Reality to Our Predictions

$$E(Y - \hat{Y})^2 = E[f(X) + \epsilon - \hat{f}(X)]^2$$

$$= \underbrace{[f(X) - \hat{f}(X)]^2}_{Reducible} + \underbrace{Var(\epsilon)}_{Irreducible}$$

Estimating $f()$

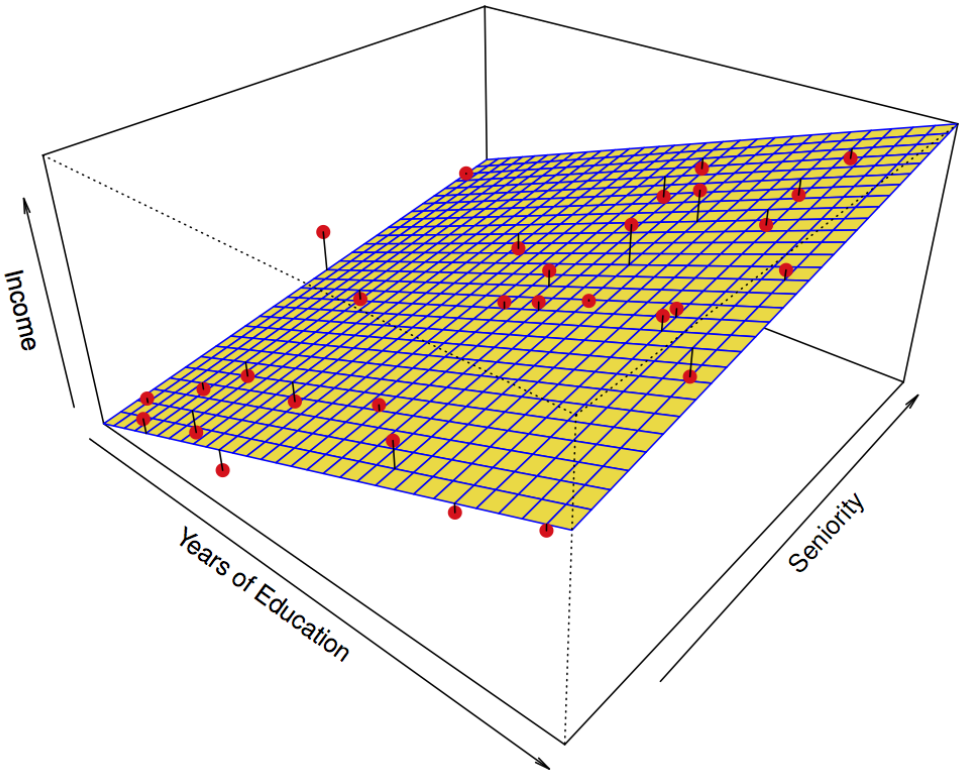
- Parametric methods
- Non-parametric methods

Multivariate Linear Regression

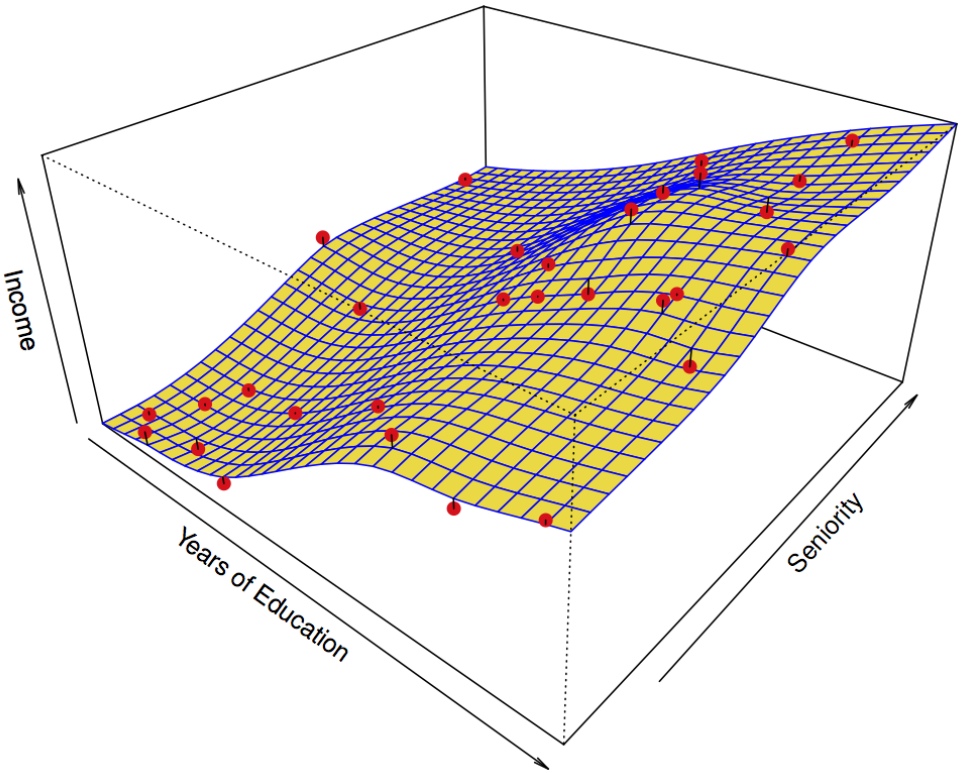
$$f(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p$$

$$Y \approx \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p$$

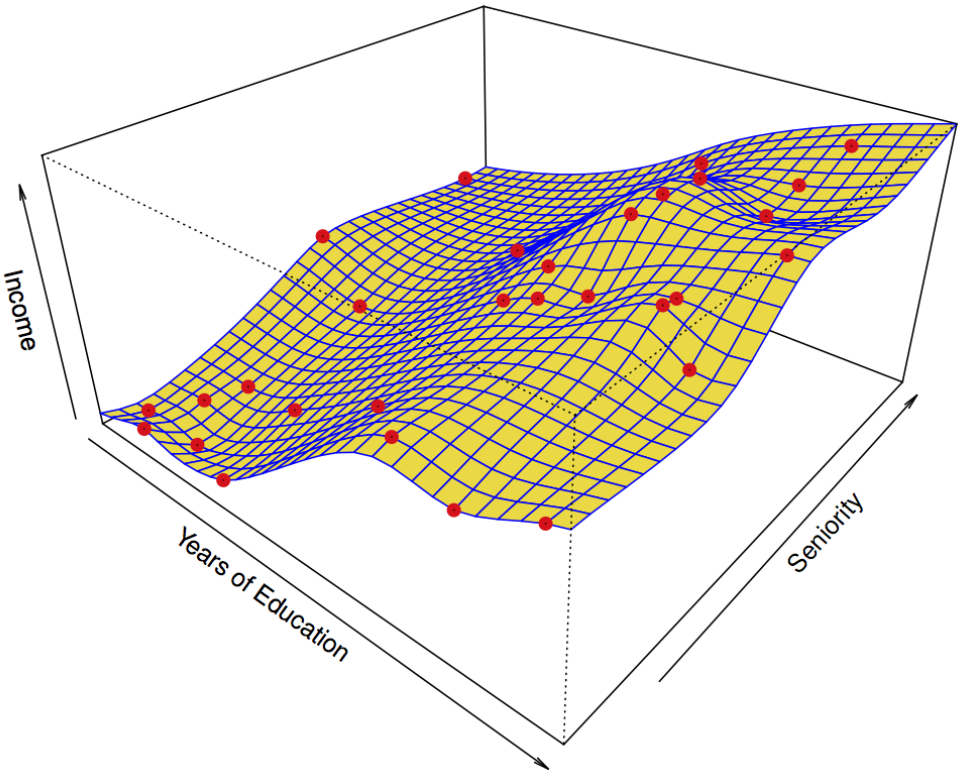
$$income \approx \beta_0 + \beta_1 \times education + \beta_2 \times seniority$$



"An Introduction to Statistical Learning", James et al

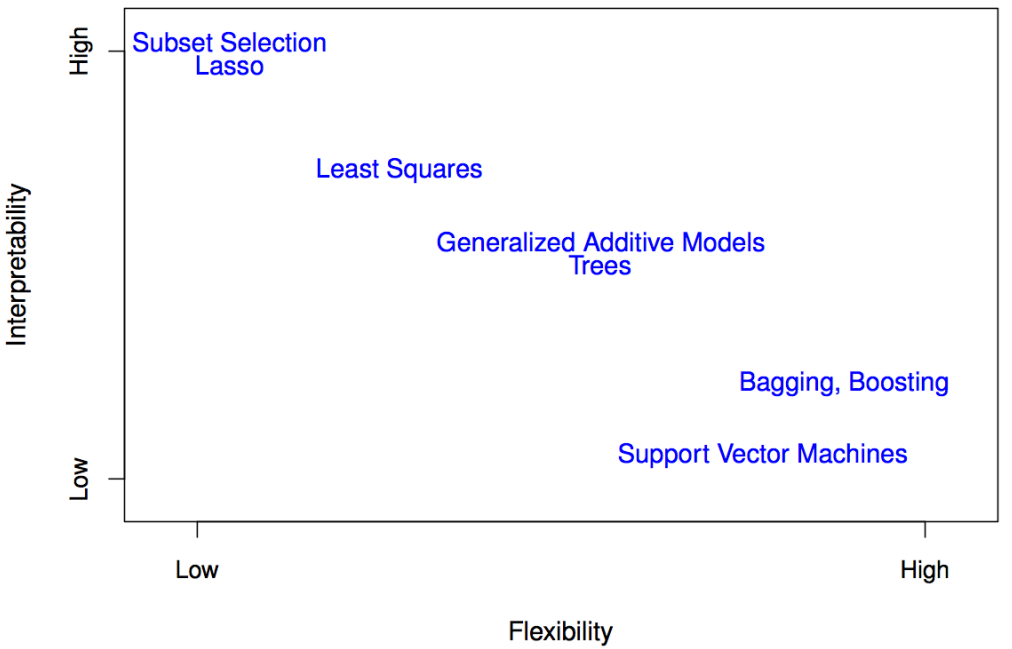


"An Introduction to Statistical Learning", James et al



"An Introduction to Statistical Learning", James et al

Interpretability vs. Flexibility Tradeoff



"An Introduction to Statistical Learning", James et al

Measuring Fitness: Mean Squared Error

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{f}(x_i))^2$$

Training MSE

Training data $= (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$

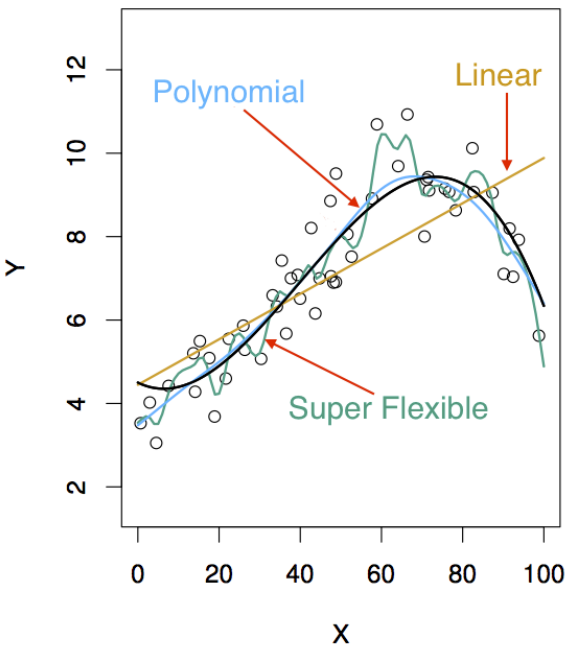
Compute $\hat{f}(x_1), \hat{f}(x_2), \ldots, \hat{f}(x_n)$

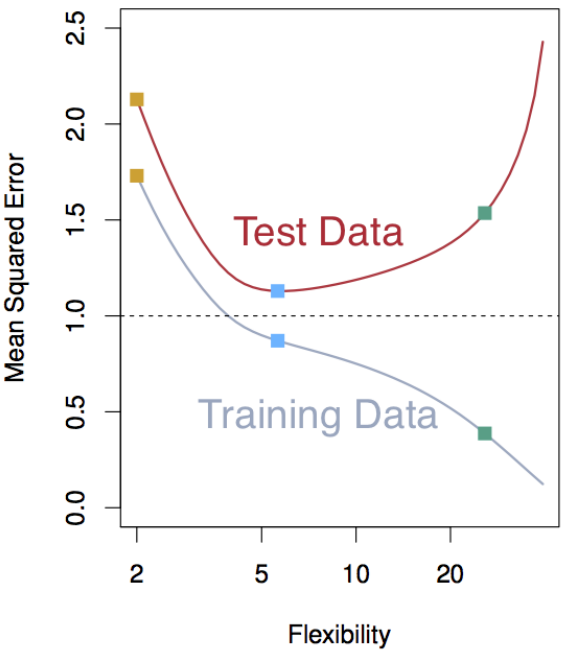
$\hat{f}(x_i) \approx y_i$ means training MSE is small

$\hat{f}(x_0) \approx y_0$ is more important!

$$Ave(\hat{f}(x_0) - y_0)^2$$

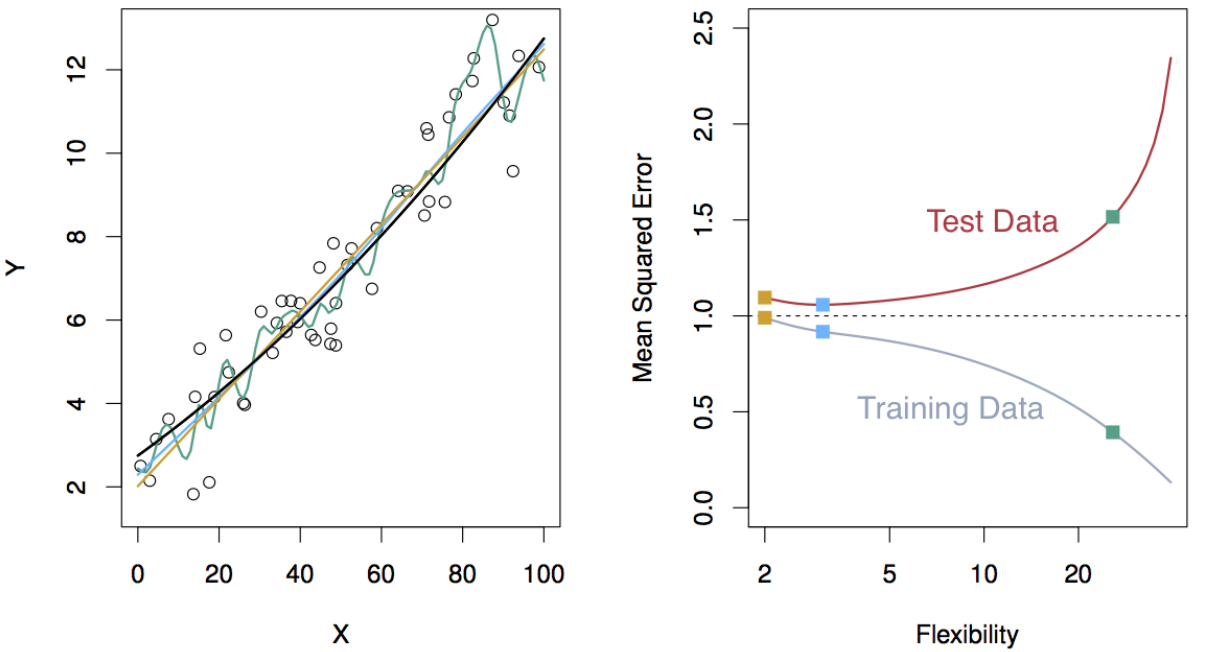
Comparing Models with MSE Curves





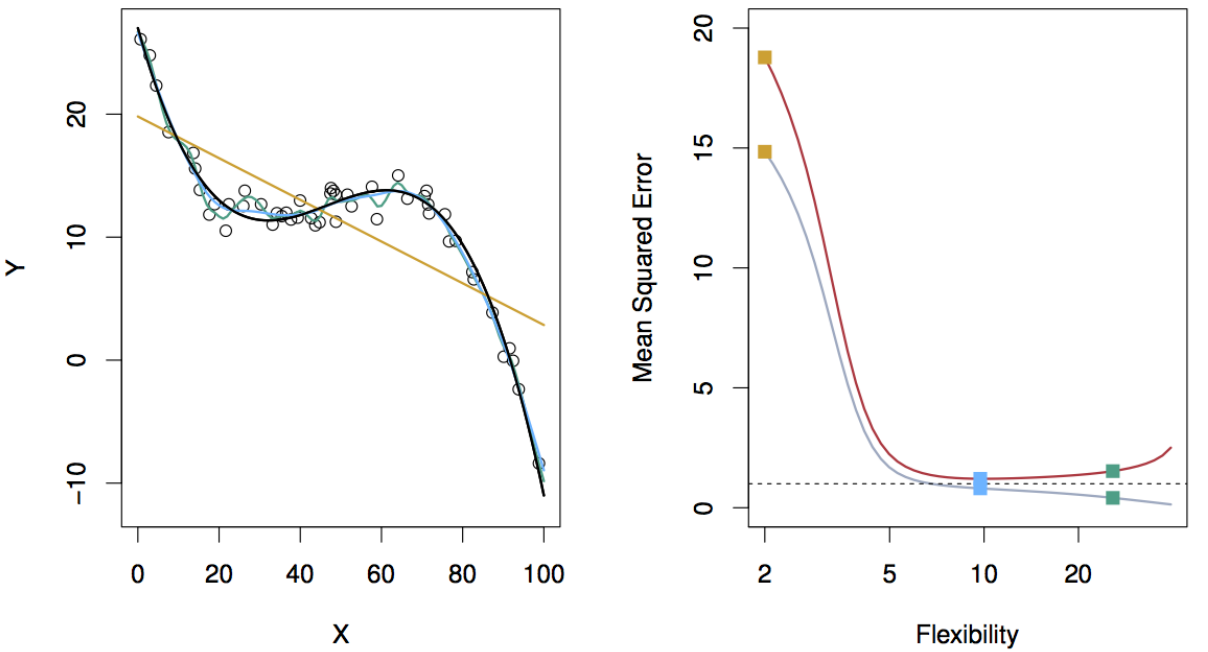
"An Introduction to Statistical Learning", James et al

More MSE Curves



"An Introduction to Statistical Learning", James et al

Even More MSE Curves



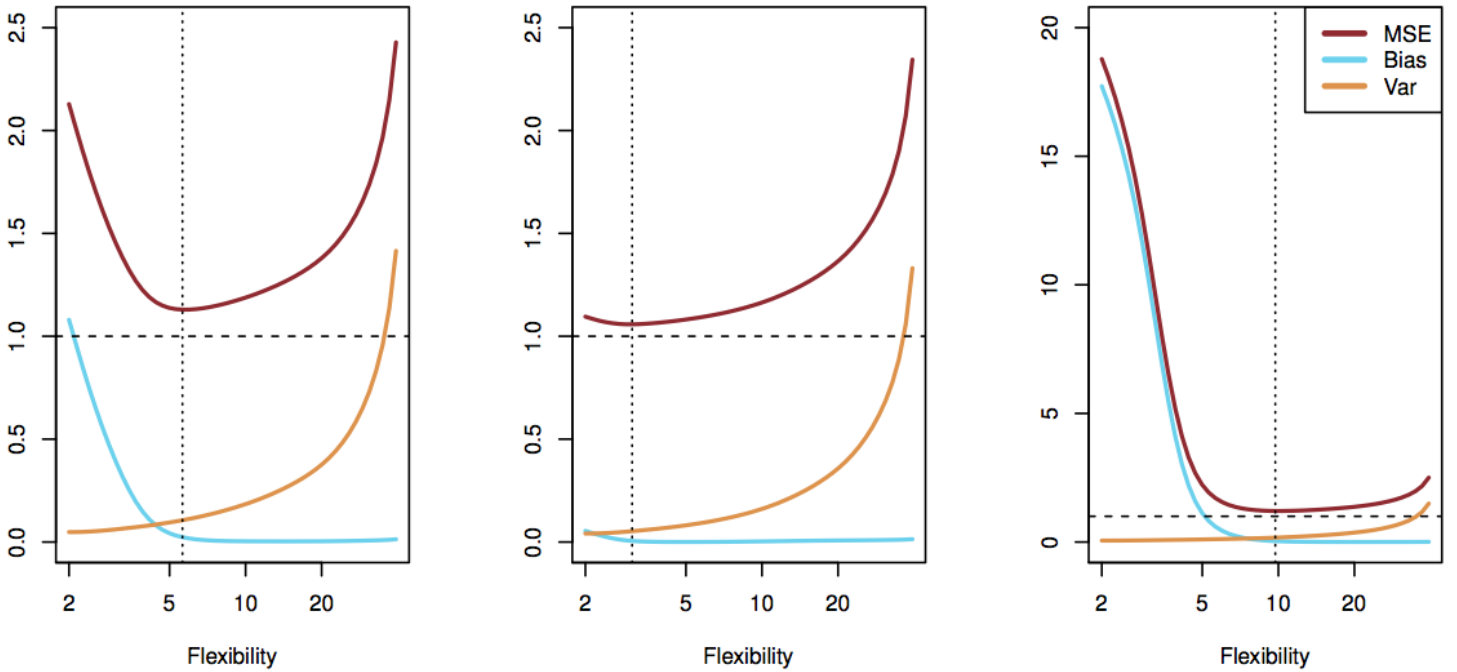
"An Introduction to Statistical Learning", James et al

Bias-Variance Trade-Off

$$E(y_0 - \hat{f}(x_0))^2 = Var(\hat{f}(x_0)) + [Bias(\hat{f}(x_0))]^2 + Var(\epsilon)$$

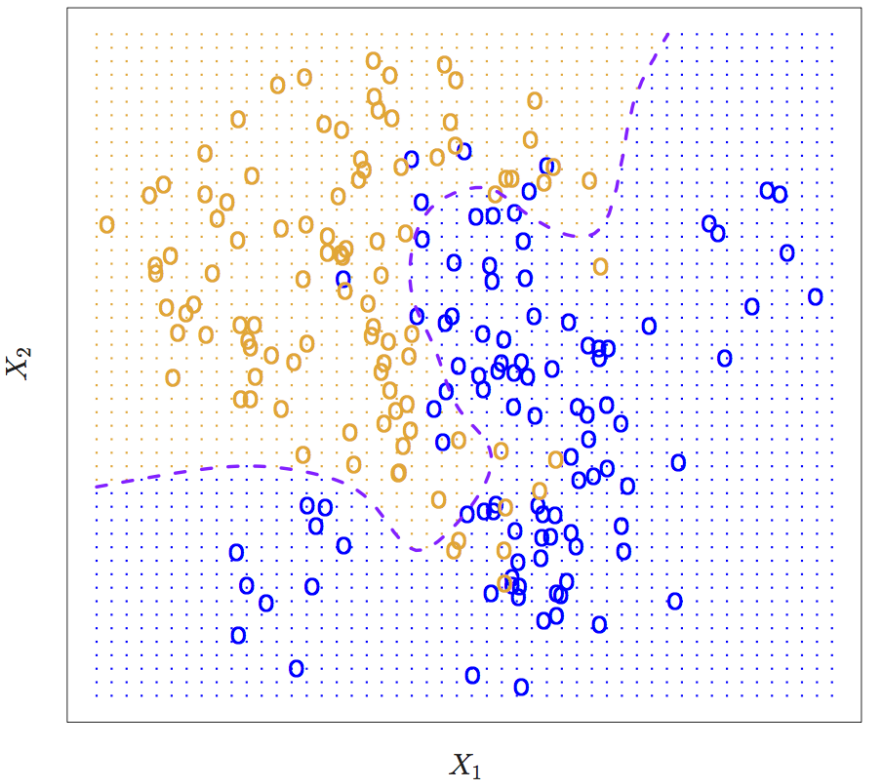
Term	Meaning
$E(y_0 - \hat{f}(x_0))^2$	expected test MSE
$Var(\hat{f}(x_0))$	amount estimate would change with different training data
$[Bias(\hat{f}(x_0))]^2$	error introduced in approximation
$Var(\epsilon)$	irreducible error

MSE, Bias, and Variance Relationships



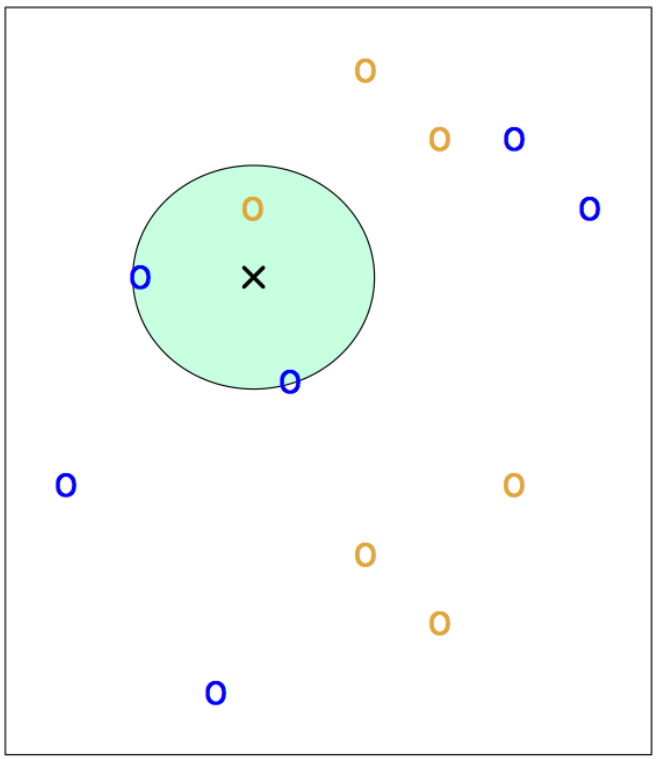
"An Introduction to Statistical Learning", James et al

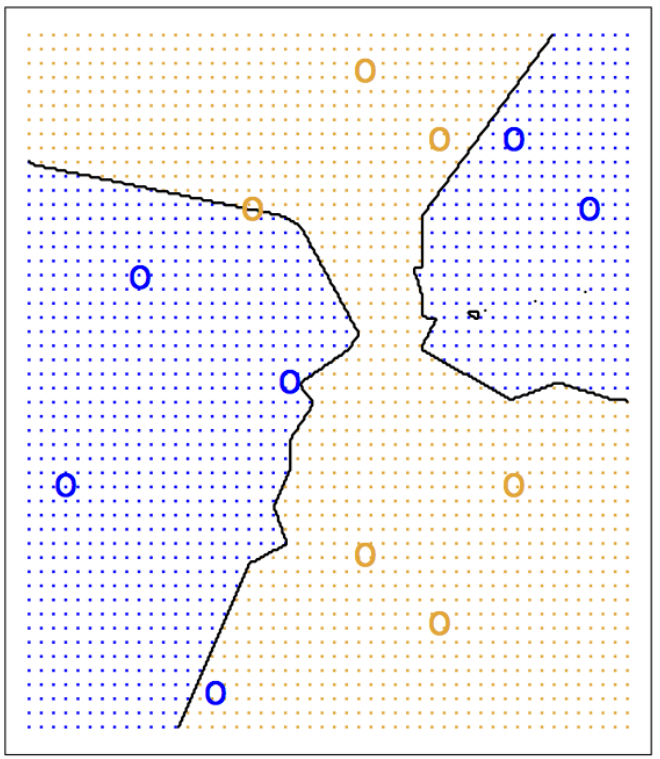
Comparing Classification Models



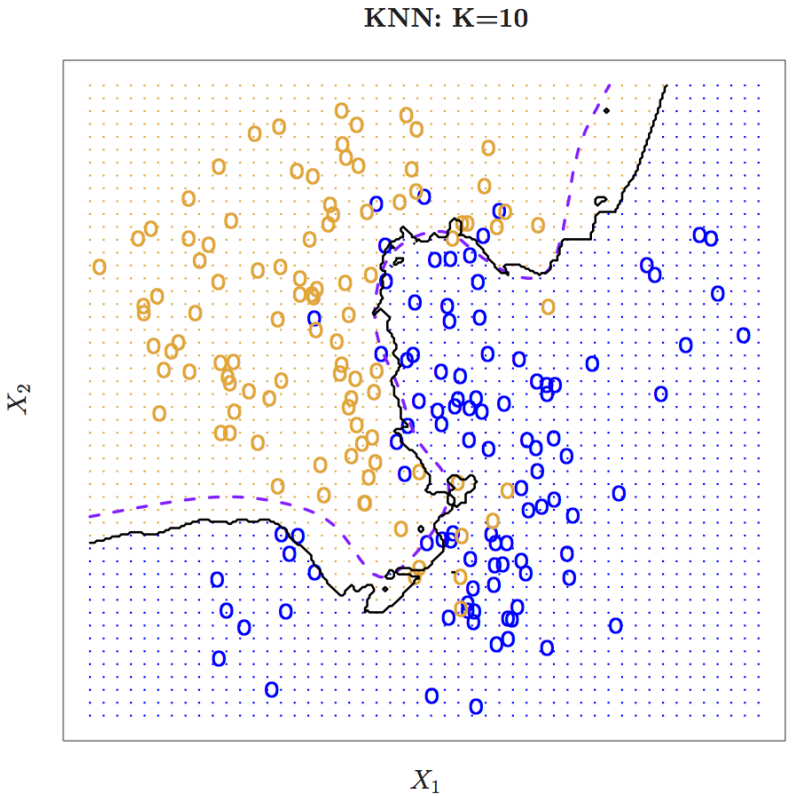
"An Introduction to Statistical Learning", James et al

K-Nearest Neighbors

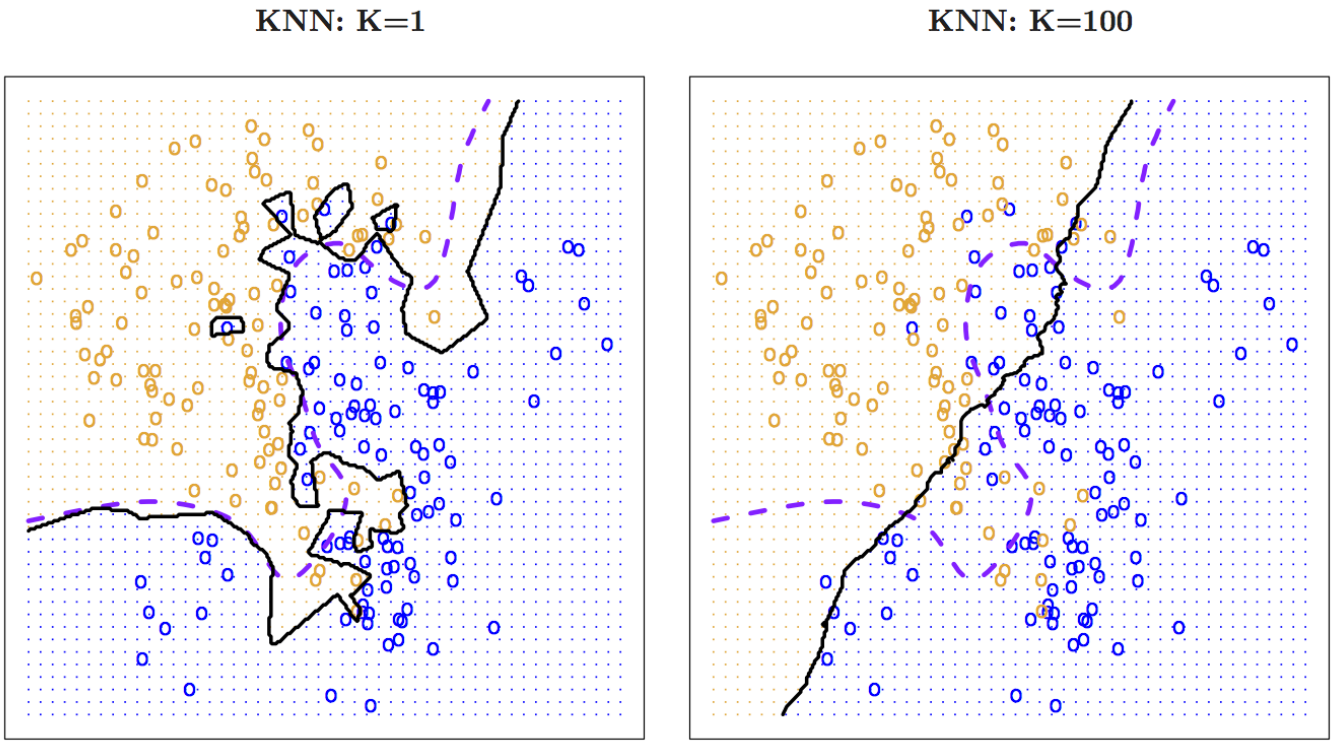




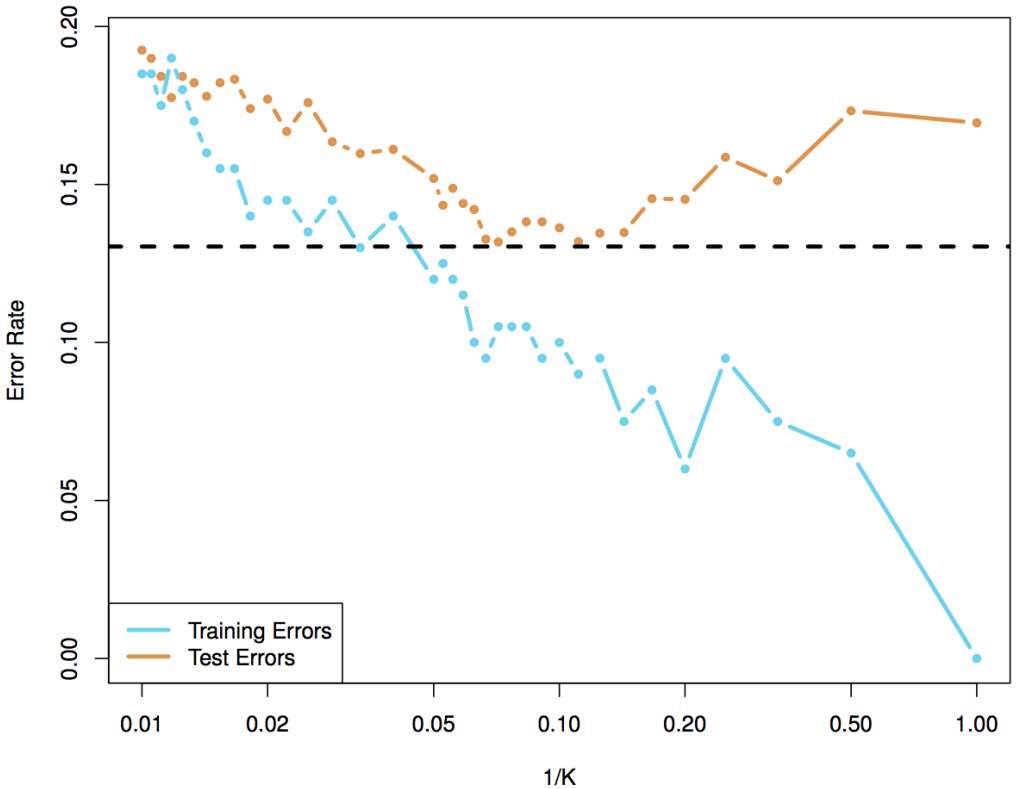
"An Introduction to Statistical Learning", James et al



"An Introduction to Statistical Learning", James et al



"An Introduction to Statistical Learning", James et al



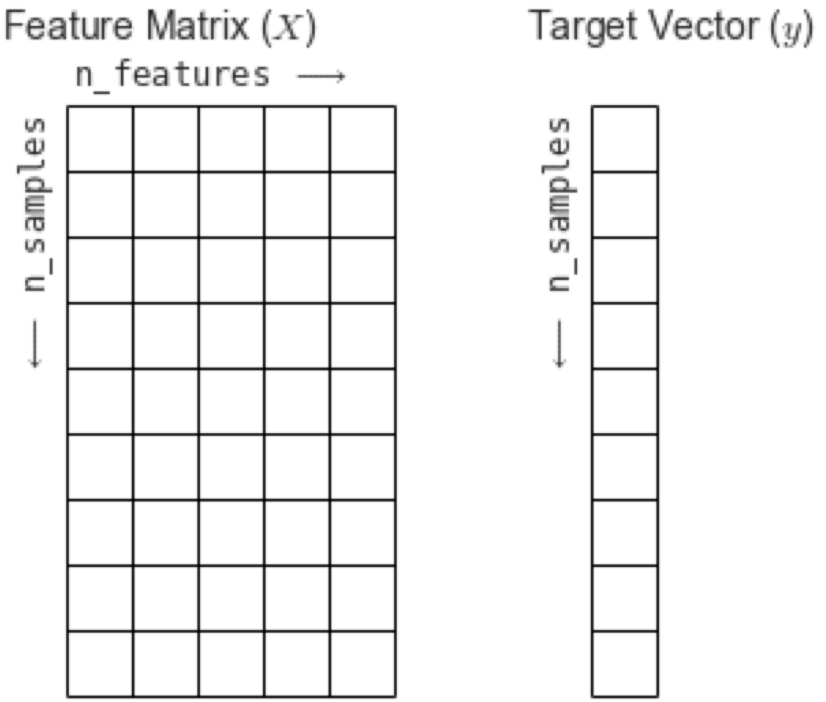
"An Introduction to Statistical Learning", James et al

Demo: SciKit-Learn

SciKit Learn

- popular Python library providing efficient implementation of a large number of machine learning algorithms
- purposely designed to be clean and uniform across tools
- consistent data representation and common interface
- there are screenshots in this presentation to maintain continuity, but let's open the notebook named **`Demo - SciKit Learn.ipynb`** and go through it together
- when done, click here to skip screesnhots

SciKit-Learn Data Representation



"Python Data Science Handbook"

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

np.random.seed(5)

iris = datasets.load_iris()
dat = pd.DataFrame(data=np.c_[iris['data'], iris['target']],
                   columns=iris['feature_names'] + ['target'])
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0.0
1	4.9	3.0	1.4	0.2	0.0
2	4.7	3.2	1.3	0.2	0.0
3	4.6	3.1	1.5	0.2	0.0
4	5.0	3.6	1.4	0.2	0.0
5	5.4	3.9	1.7	0.4	0.0
6	4.6	3.4	1.4	0.3	0.0
7	5.0	3.4	1.5	0.2	0.0
8	4.4	2.9	1.4	0.2	0.0
9	4.9	3.1	1.5	0.1	0.0
10	5.4	3.7	1.5	0.2	0.0
11	4.8	3.4	1.6	0.2	0.0

```
# Set up the feature matrix

>>> X_iris = dat.drop('target', axis=1)
>>> X_iris.shape
(150, 4)
```

```
# Set up the target vector

>>> y_iris = iris['target']
>>> y_iris.shape
(150,)
```

`scikit-learn` Objects

"All objects within scikit-learn share a uniform common basic API consisting of three complementary interfaces: an estimator interface for building and fitting models, a predictor interface for making predictions and a transformer interface for converting data."

API design for machine learning software: experiences from the scikit-learn project (https://arxiv.org/pdf/1309.0238.pdf)

Estimator API

- Driven by a set of principles documented in a paper: https://arxiv.org/pdf/1309.0238.pdf
  - Consistency
  - Allow Inspection
  - Limited object hierarchies
  - Composition
  - Sensible defaults

General Flow

- Choose a model
- Choose model hyperparameters
- Arrange data into a features matrix and target vector
- Fit the model to the data with the fit() method
- Apply the model to test data (predict() or transform())

Examples

```
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = 2 * x - 5 + rng.randn(50)
plt.scatter(x, y);

from sklearn.linear_model import LinearRegression
model = LinearRegression(fit_intercept=True)
model.fit(x[:, np.newaxis], y)
xfit = np.linspace(0, 10, 1000)
yfit = model.predict(xfit[:, np.newaxis])
```

```
from sklearn.tree import DecisionTreeClassifier
tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X_iris, y_iris)
tree_clf.predict([[5, 1.5, 2, 2.0]])
```

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
X, y = make_blobs(random_state=1)
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
kmeans.labels_
```

Exercise: SciKit-Learn