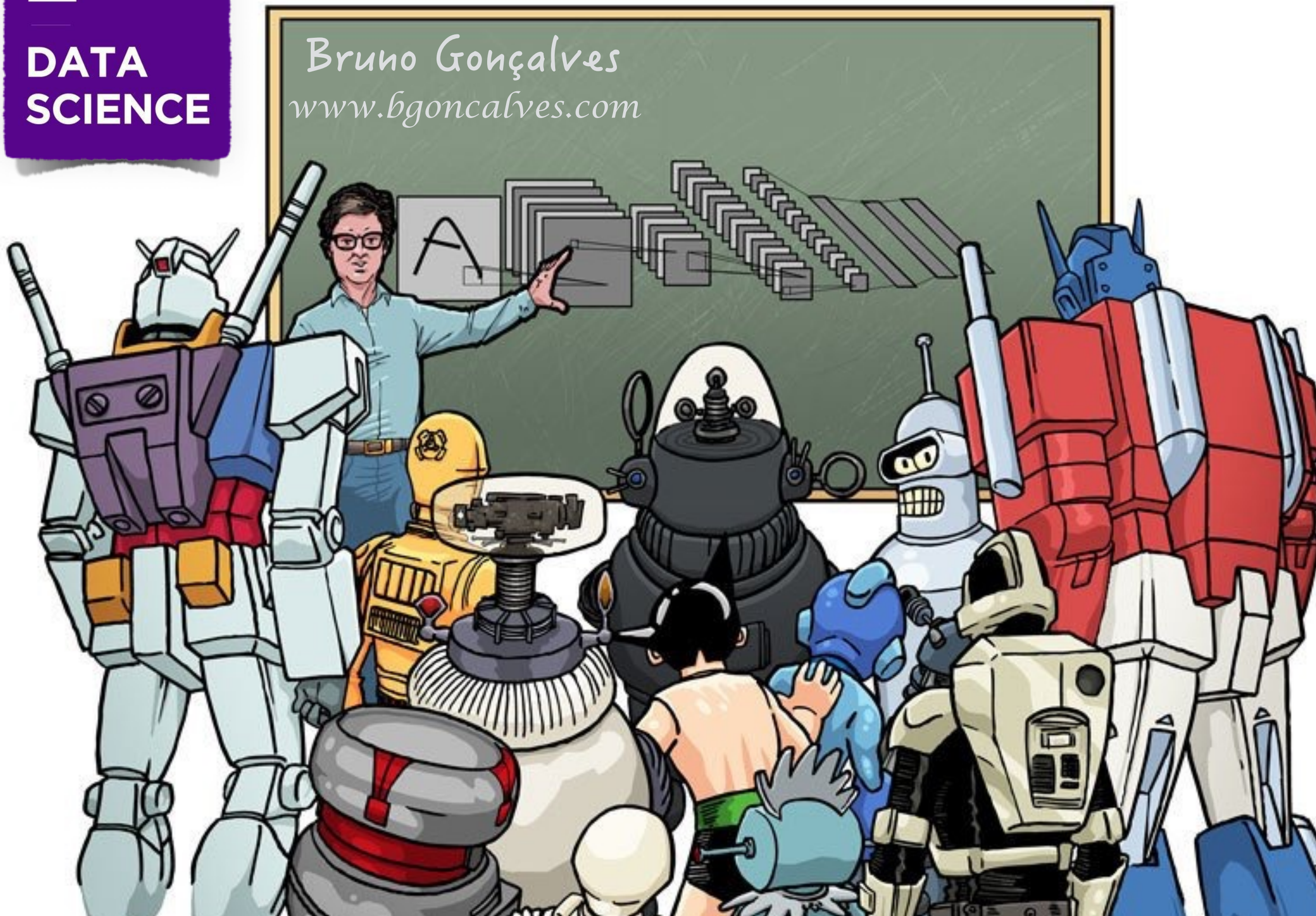
NYU

DATA SCIENCE

Machine(s) Learning and Data Science

Bruno Gonçalves
www.bgoncalves.com

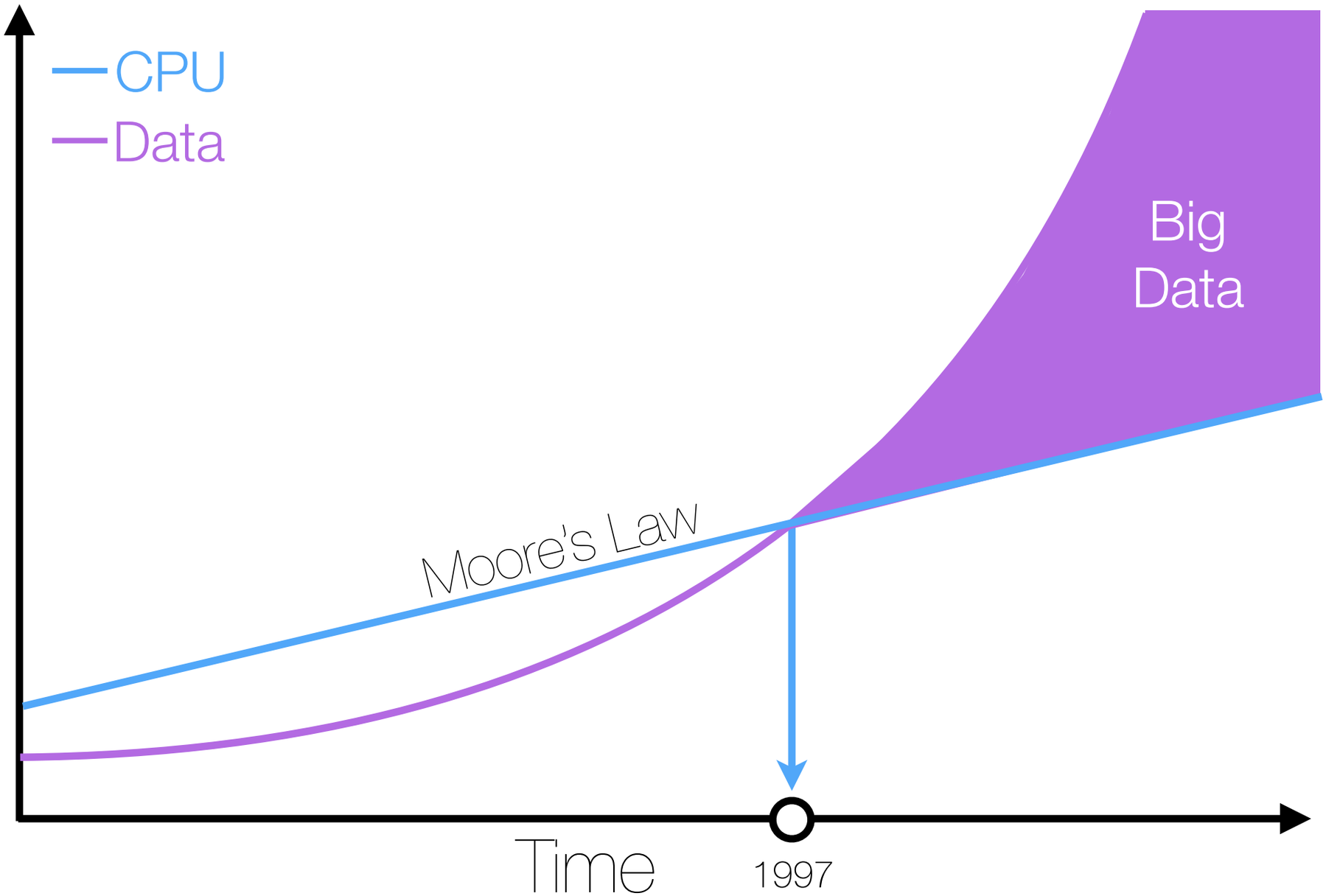


Requirements

http://github.com/bmtgoncalves/UDD

BIG DATA

Big Data



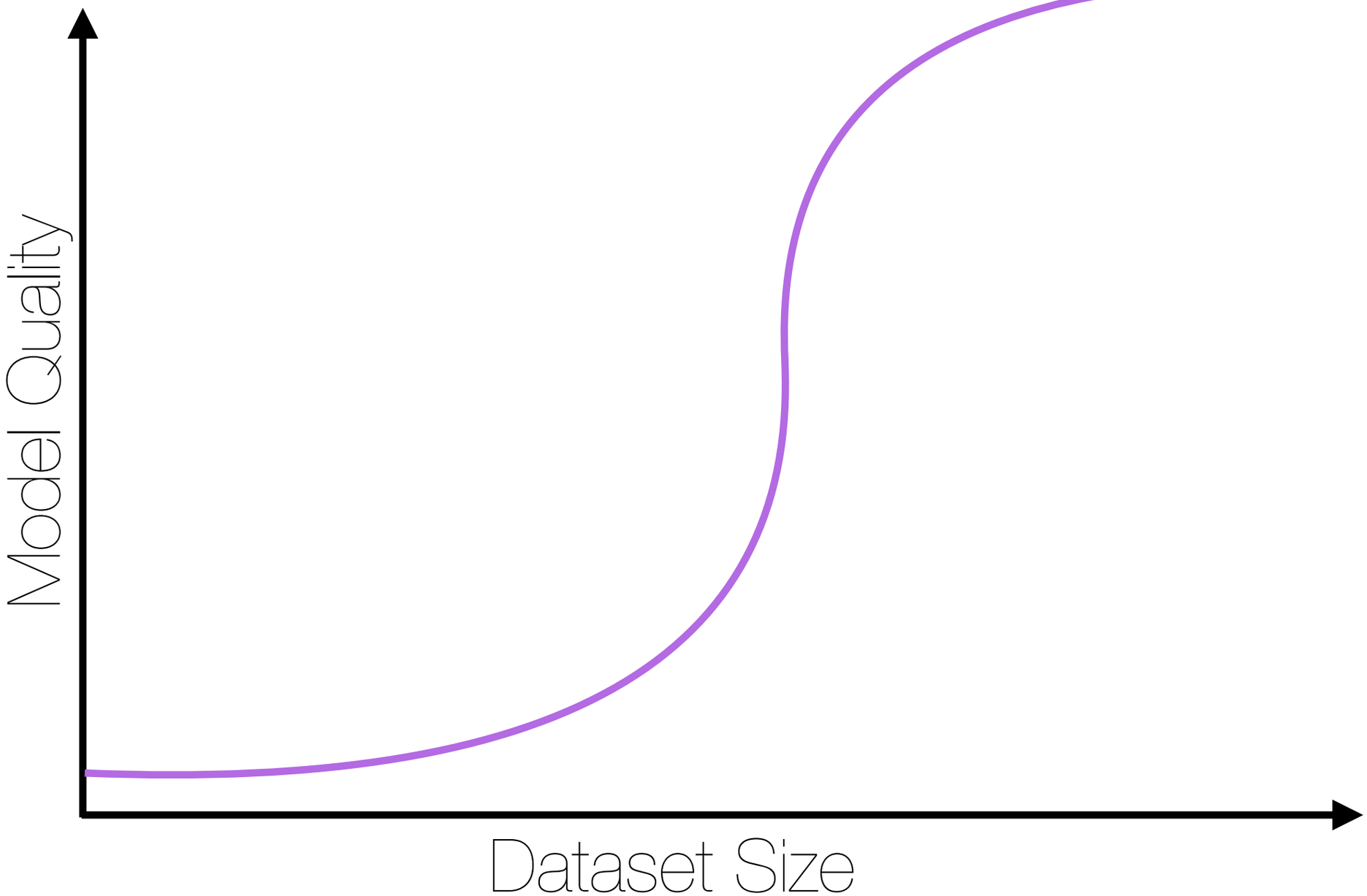
EXPERT OPINION

Contact Editor: **Brian Brannon,** bbrannon@computer.org

The Unreasonable Effectiveness of Data

Alon Halevy, Peter Norvig, and Fernando Pereira, *Google*

Big Data



https://www.wired.com/2008/06/pb-theory/

CHRIS ANDERSON SCIENCE 06.23.08 12:00 PM

THE END OF THEORY: THE DATA DELUGE MAKES THE SCIENTIFIC METHOD OBSOLETE

Illustration: Marian Bantjes

"**All models are wrong**, but some are useful."

https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century

Data Scientist:

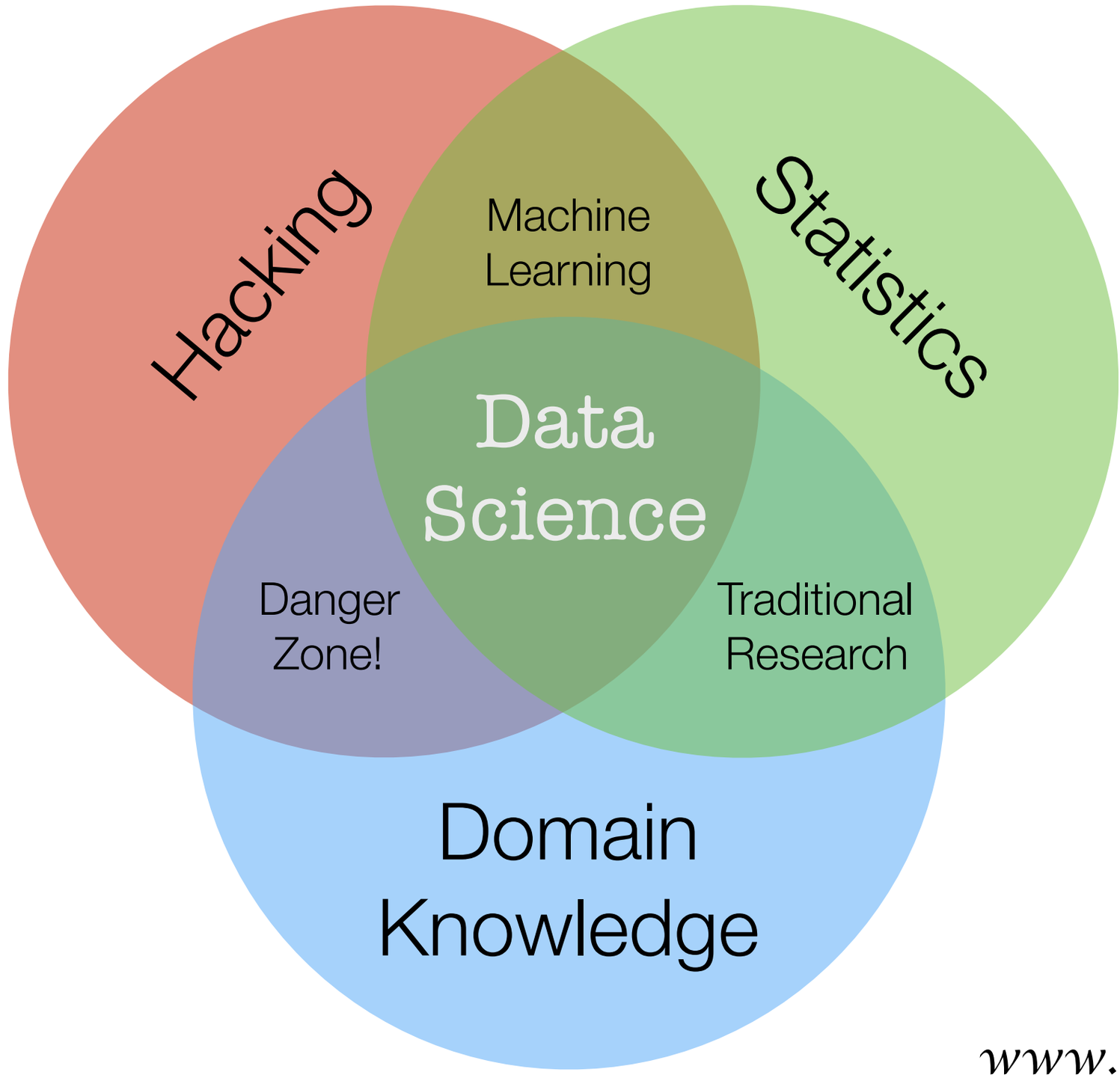
The Sexiest Job of the 21st Century

Meet the people who can coax treasure out of messy, unstructured data.

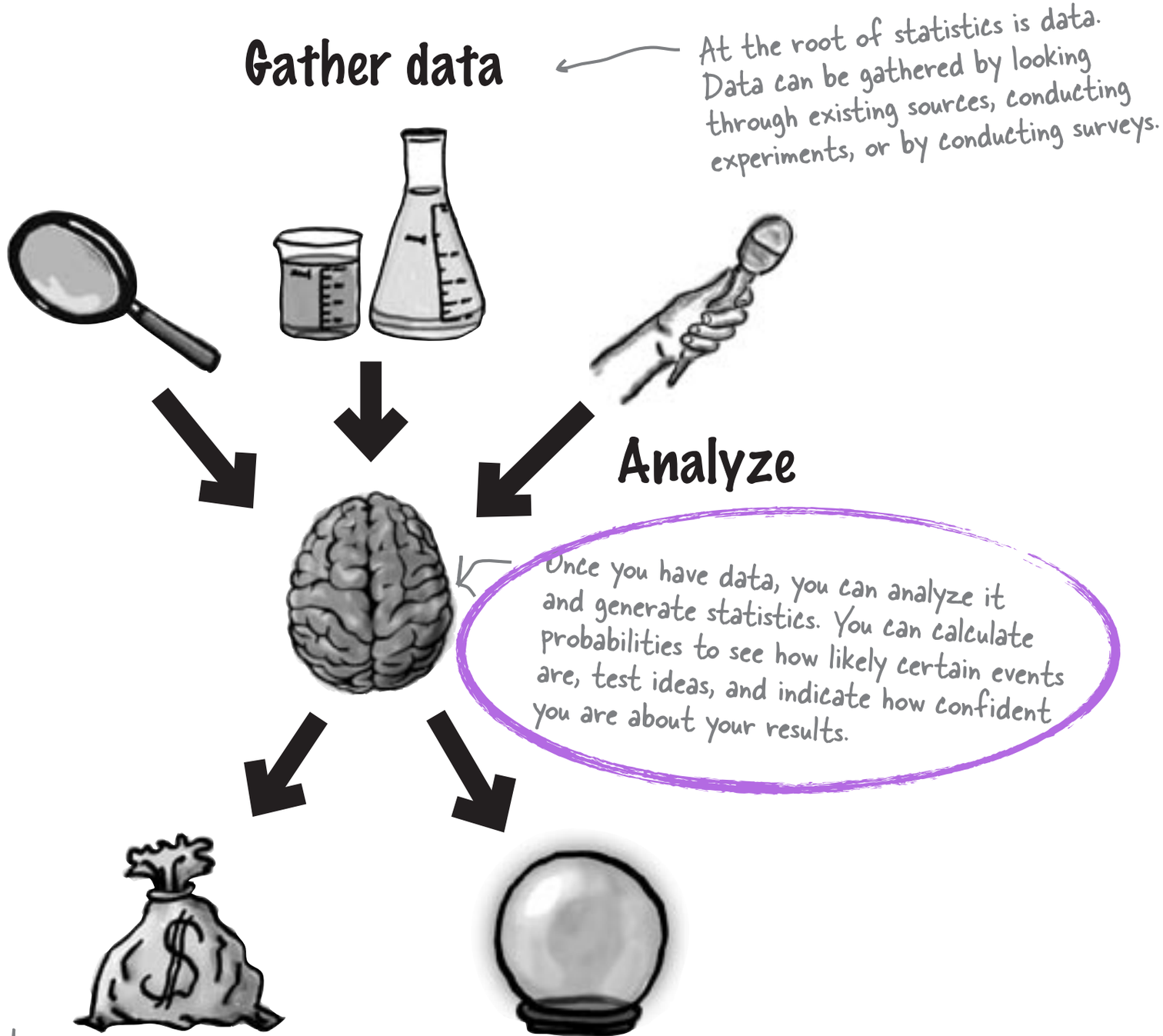
by Thomas H. Davenport and D.J. Patil

When Jonathan Goldman arrived for work in June 2006 at LinkedIn, the business networking site, the place still felt like a start-up. The company had just under 8 million accounts, and the number was growing quickly as existing members invited their friends and colleagues to join. But users weren't seeking out connections with the people who were already on the site at the rate executives had expected. Something was apparently missing in the social experience. As one LinkedIn manager put it, "It was like arriving at a conference reception and realizing you don't know anyone. So you just stand in the corner sipping your drink—and you probably leave early."

Data Science

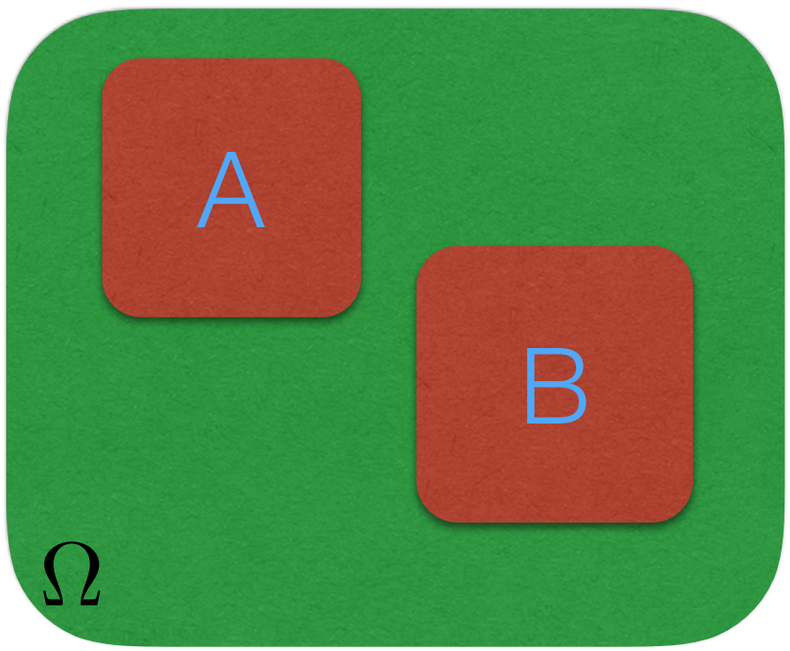


From Data To Information



Probability

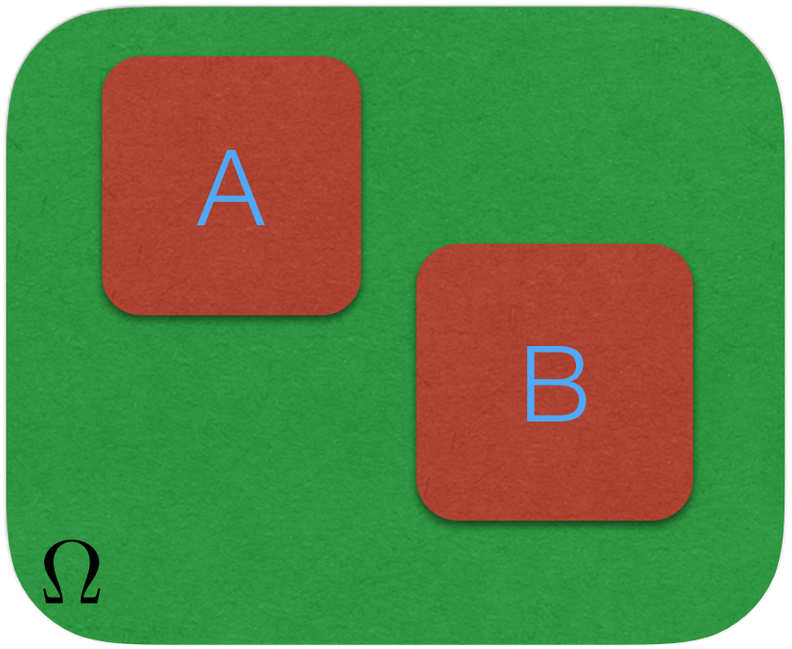
$$P(A) = \text{"Area" of A}$$
$$P(\Omega) = 1 \text{ (Normalization)}$$

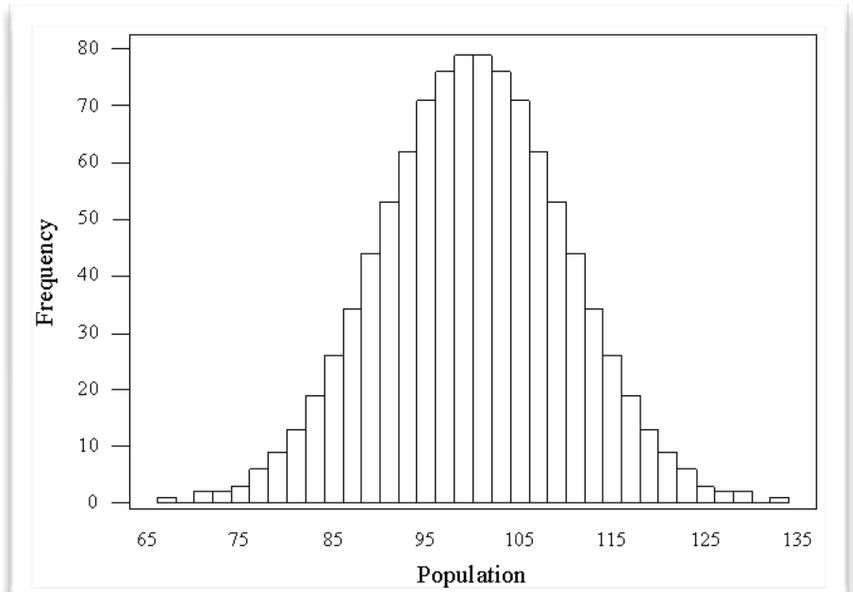


$$P(A \text{ or } B) = P(A) + P(B)$$

Probability

$$P(A) = \text{"Area" of A}$$
$$P(\Omega) = 1 \text{ (Normalization)}$$



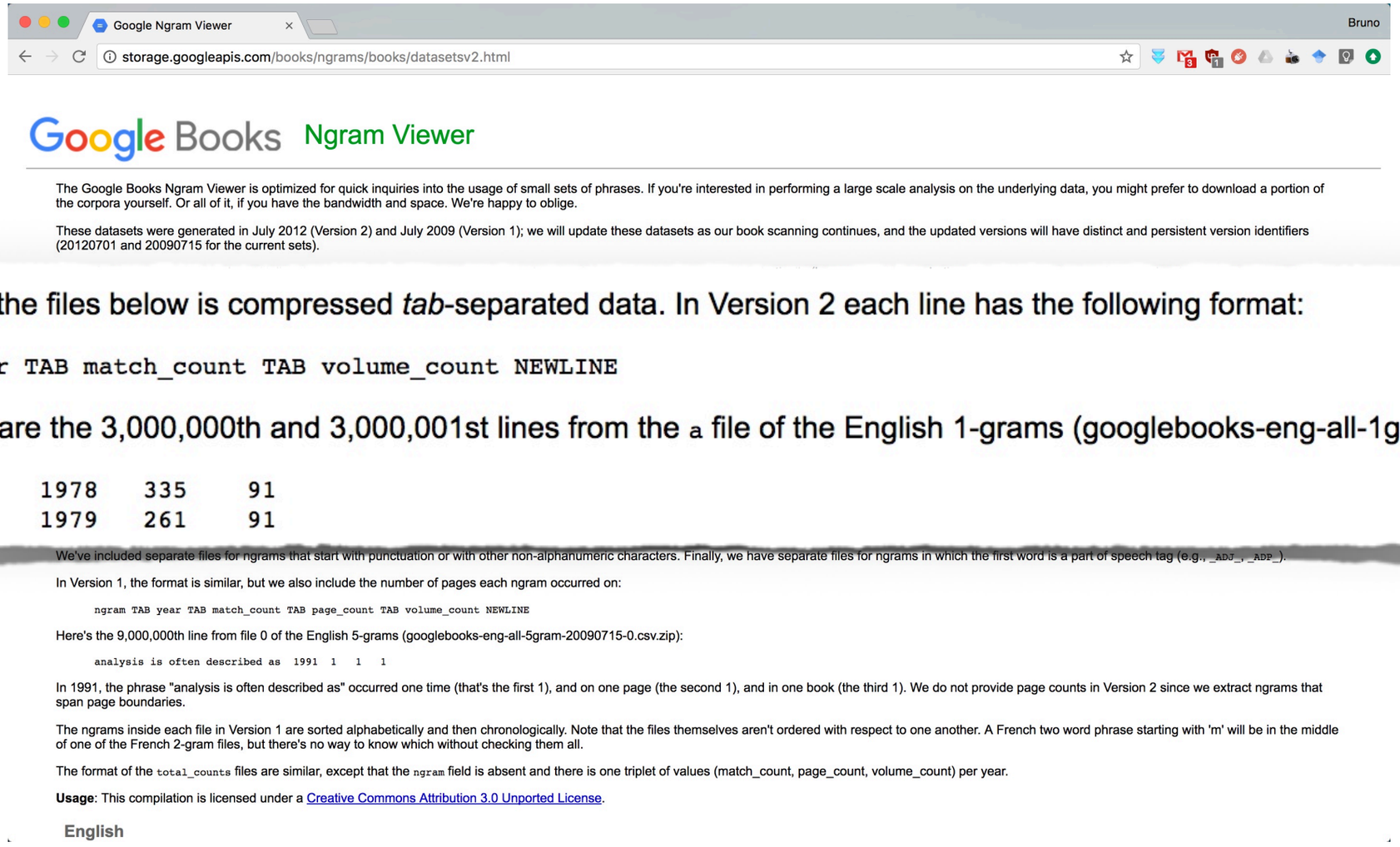


$$P(A \text{ or } B) = P(A) + P(B)$$

Character Probabilities

http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

As an example, let's calculate the probability of each letter occurring in the English language using Google Books 1-gram dataset.



Google Ngram Viewer

storage.googleapis.com/books/ngrams/books/datasetsv2.html

Bruno

Google Books Ngram Viewer

The Google Books Ngram Viewer is optimized for quick inquiries into the usage of small sets of phrases. If you're interested in performing a large scale analysis on the underlying data, you might prefer to download a portion of the corpora yourself. Or all of it, if you have the bandwidth and space. We're happy to oblige.

These datasets were generated in July 2012 (Version 2) and July 2009 (Version 1); we will update these datasets as our book scanning continues, and the updated versions will have distinct and persistent version identifiers (20120701 and 20090715 for the current sets).

File format: Each of the files below is compressed *tab*-separated data. In Version 2 each line has the following format:

```
ngram TAB year TAB match_count TAB volume_count NEWLINE
```

As an example, here are the 3,000,000th and 3,000,001st lines from the `a` file of the English 1-grams (googlebooks-eng-all-1gram-20120701-a.gz):

```
circumvallate   1978   335    91
circumvallate   1979   261    91
```

We've included separate files for ngrams that start with punctuation or with other non-alphanumeric characters. Finally, we have separate files for ngrams in which the first word is a part of speech tag (e.g., `_ADJ_`, `_ADP_`).

In Version 1, the format is similar, but we also include the number of pages each ngram occurred on:

```
ngram TAB year TAB match_count TAB page_count TAB volume_count NEWLINE
```

Here's the 9,000,000th line from file 0 of the English 5-grams (googlebooks-eng-all-5gram-20090715-0.csv.zip):

```
analysis is often described as  1991  1   1   1
```

In 1991, the phrase "analysis is often described as" occurred one time (that's the first 1), and on one page (the second 1), and in one book (the third 1). We do not provide page counts in Version 2 since we extract ngrams that span page boundaries.

The ngrams inside each file in Version 1 are sorted alphabetically and then chronologically. Note that the files themselves aren't ordered with respect to one another. A French two word phrase starting with 'm' will be in the middle of one of the French 2-gram files, but there's no way to know which without checking them all.

The format of the `total_counts` files are similar, except that the `ngram` field is absent and there is one triplet of values (match_count, page_count, volume_count) per year.

Usage: This compilation is licensed under a Creative Commons Attribution 3.0 Unported License.

English

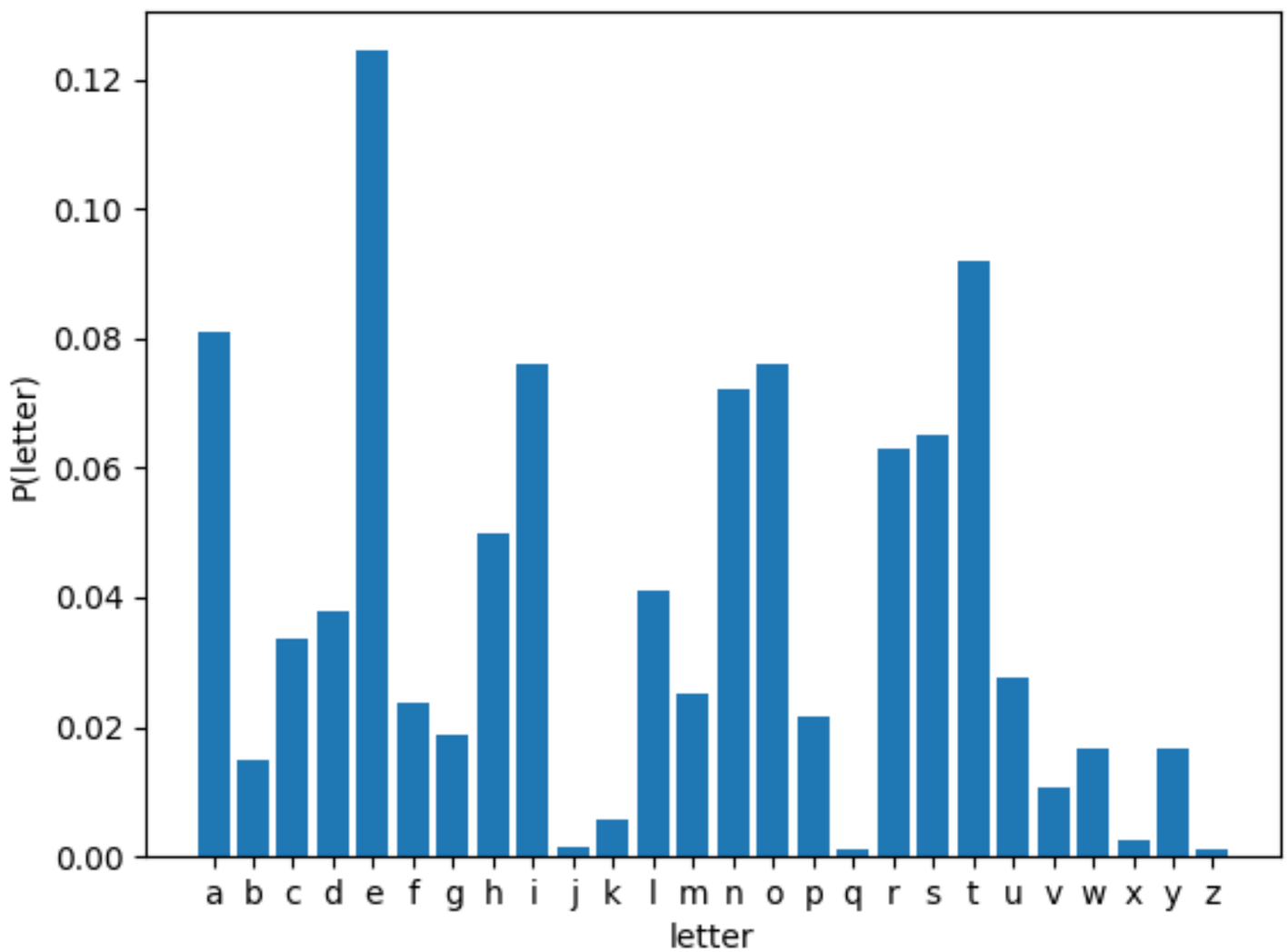
Character Probabilities

```
pos = dict(zip(characters, range(len(characters))))
counts = np.zeros(len(characters), dtype='uint64')

line_count = 0

for filename in sys.argv[1:]:
    for line in gzip.open(filename, "rt"):
        fields = line.lower().strip().split()

        count = int(fields[2])
        word = fields[0]

        if "_" in word:
            continue

        letters = letter_regex.findall(word)

        if len(letters) != len(word):
            continue

        for letter in letters:
            if letter not in pos:
                continue

            counts[pos[letter]] += count

total = np.sum(counts)
pos = list(pos.items())
pos.sort(key=lambda x: x[1])

for key, value in enumerate(pos):
    print(value[0], counts[key]/total)
```

build_lang_table.py

Character Probabilities

```
pos = dict(zip(characters, range(len(characters))))
counts = np.zeros(len(characters), dtype='uint64')
```

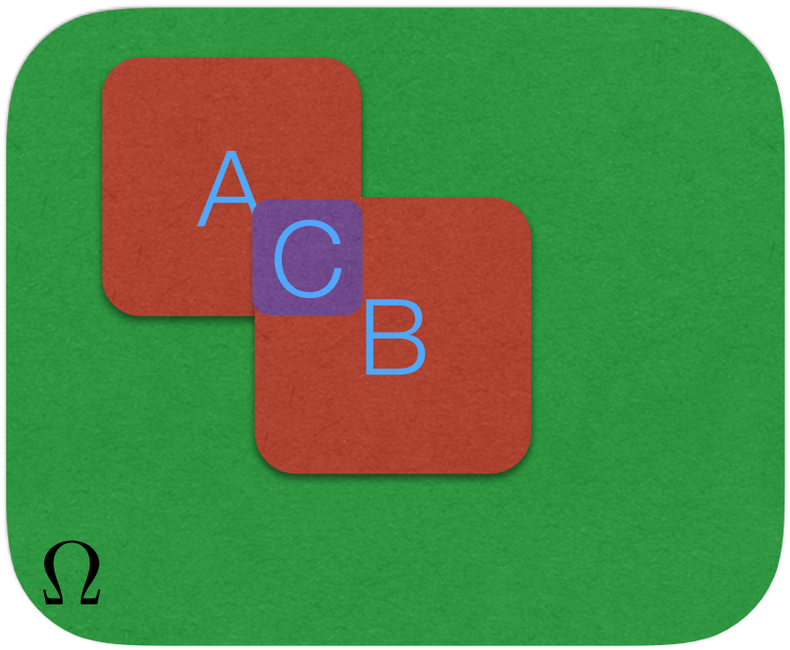



```
for key, value in enumerate(pos):
    print(value[0], counts[key]/total)
```

build_lang_table.py

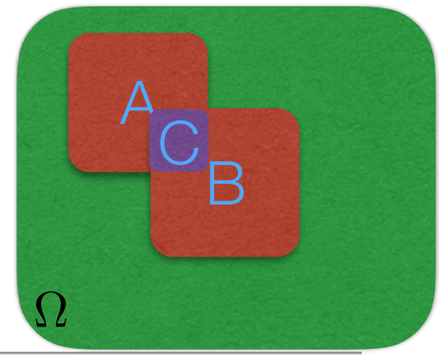
Probability

$$P(A) = \text{"Area" of A}$$
$$P(\Omega) = 1 \text{ (Normalization)}$$



$$P(A \text{ or } B) = P(A) + P(B) - P(A \text{ and } B)$$
$$P(C) = P(A \text{ and } B) = \text{overlap of A and B}$$

Probability



$$P(A) = \text{"Area" of A}$$
$$P(\Omega) = 1 \text{ (Normalization)}$$
$$P(A \text{ or } B) = P(A) + P(B) - P(A \text{ and } B)$$
$$P(C) = P(A \text{ and } B) = \text{overlap of A and B}$$

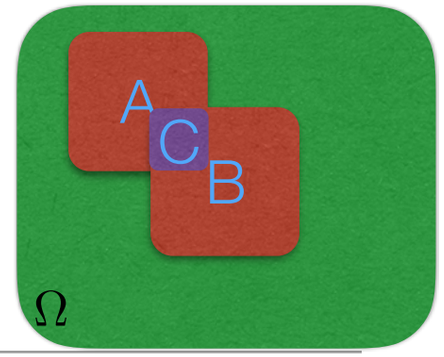
What's the probability that I'm in B given that I'm in A?

What fraction of A is occupied by B?

$$P(B|A) = \frac{P(C)}{P(A)} \rightarrow P(C) = P(B|A)\,P(A)$$

$$P(C) = P(C)$$

$$P(C) = P(C)$$



$$P(B|A) = \frac{P(C)}{P(A)} \rightarrow P(C) = P(B|A)\,P(A)$$

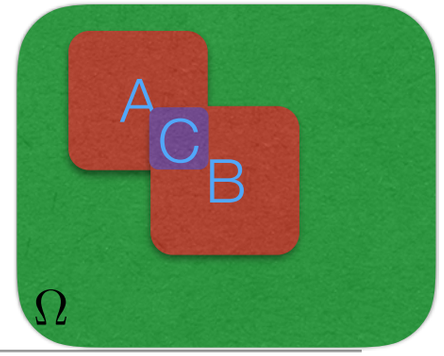
$$P(A|B) = \frac{P(C)}{P(A)} \rightarrow P(C) = P(A|B)\,P(B)$$

$$P(B|A)\,P(A) = P(A|B)\,P(B)$$

$$P(B|A) = \frac{P(A|B)\,P(B)}{P(A)}$$

Bayes Theorem

Medical Tests



Your doctor thinks you might have a rare disease that affects 1 person in 10,000. A test that is 99% accurate comes out positive. What's the probability of you having the disease?

Bayes Theorem:

$$P\left(disease|positive\ test\right) = \frac{P\left(positive\ test|disease\right)P\left(disease\right)}{P\left(positive\ test\right)}$$

Total Probability:

$$P\left(positive\ test\right) = P\left(positive\ test|disease\right)P\left(disease\right) + P\left(positive\ test|no\ disease\right)P\left(no\ disease\right)$$

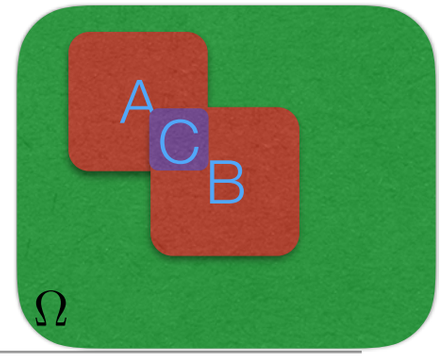
Finally:

$$P\left(disease|positive\ test\right) = 0.0098$$

Base Rate Fallacy

Low Base Rate Value
+
Non-zero False Positive Rate

Medical Tests

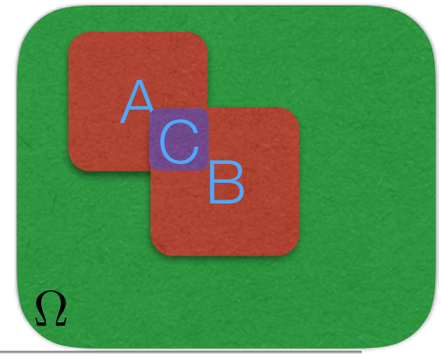


Consider a population of 1,000,000 individuals. The numbers we should expect are:

Marginals

	disease	no disease	
positive	99	9,999	10,098
negative	1	989,901	989,902
Marginals →	100	999,900	1,000,000

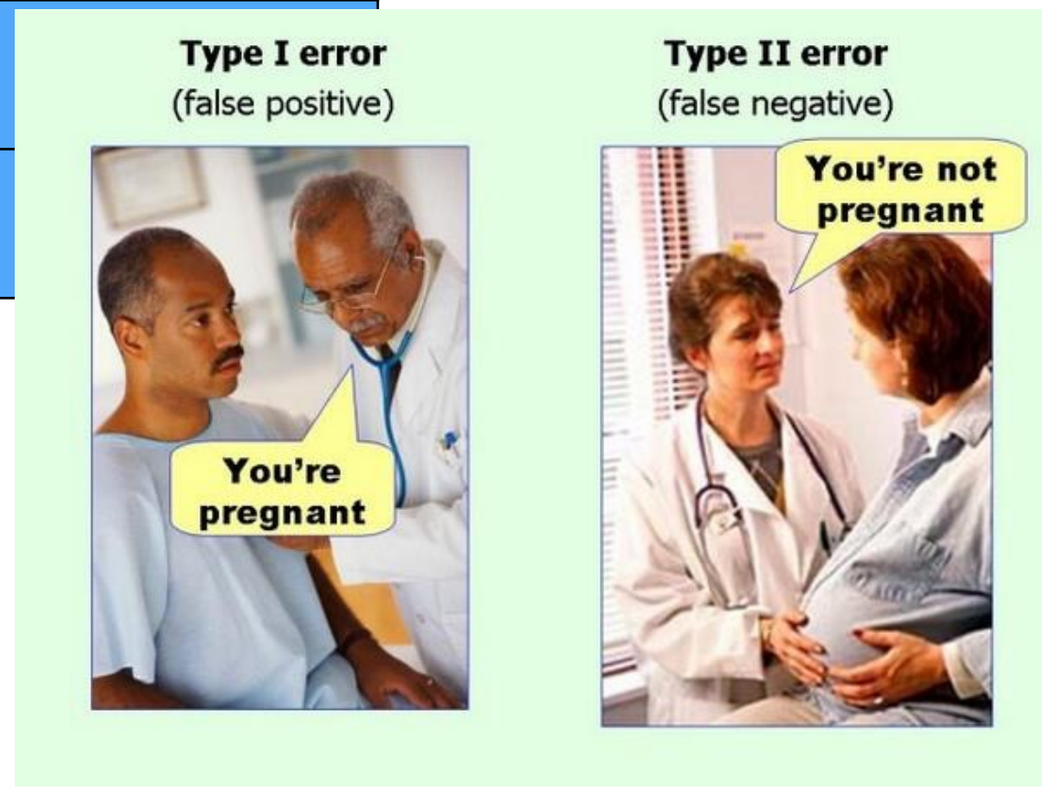
Medical Tests



Consider a population of 1,000,000 individuals. The numbers we should expect are:

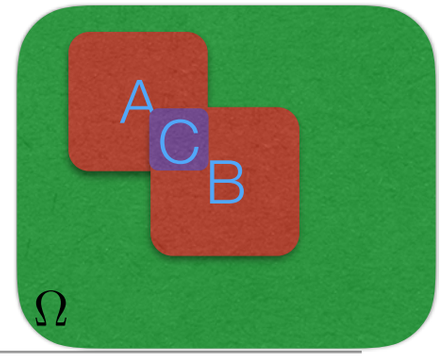
Marginals

	disease	no disease	
positive	99	9,999	10,098
negative	1	989,901	
Marginals →	100	999,900	



@bgoncalves

Medical Tests



Consider a population of 1,000,000 individuals. The numbers we should expect are:

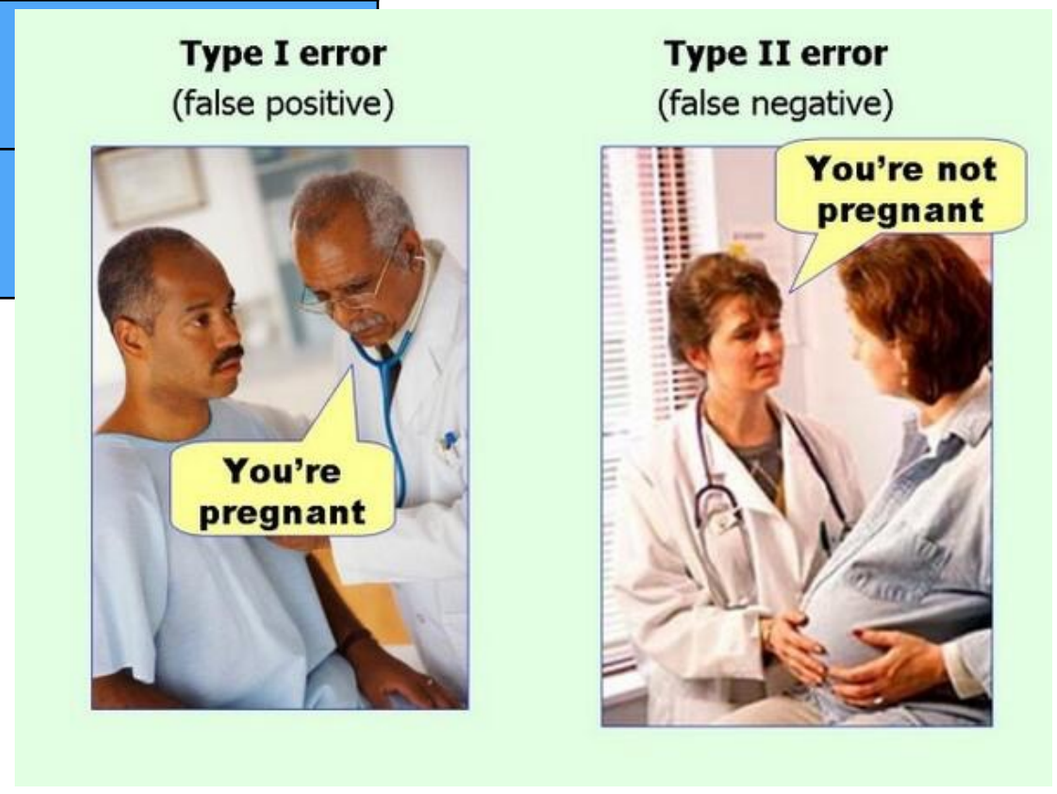
Marginals

	disease	no disease	
positive	99	9,999	10,098
negative	1	989,901	
Marginals →	100	999,900	

$$P\left(disease|positive\ test\right) = \frac{TP}{TP + FP} = 0.0098$$

$$P\left(no\ disease|negative\ test\right) = \frac{TN}{TN + FN} = 0.99999$$

@bgoncalves



(Confusion Matrix)

Feature / Test	positive	negative
positive	TP	FP
negative	FN	TN

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$specificity = \frac{TN}{FP + TN}$$

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

$$sensitivity = \frac{TP}{TP + FN}$$

harmonic mean

$$F1 = \frac{2TP}{2TP + FP + FN}$$

A second Test

Bayes Theorem still looks the same:

$$P\left(disease|positive\ test\right) = \frac{P\left(positive\ test|disease\right)P\left(disease\right)}{P\left(positive\ test\right)}$$

but now the probability that we have the disease has been updated:

$$P^{\dagger}\left(disease\right) = 0.0098$$

So this time we find:

$$P^{\dagger}\left(disease|positive\ test\right) = 0.4949$$

Each test is providing new evidence, and Bayes theorem is simply telling us how to use it to update our beliefs.

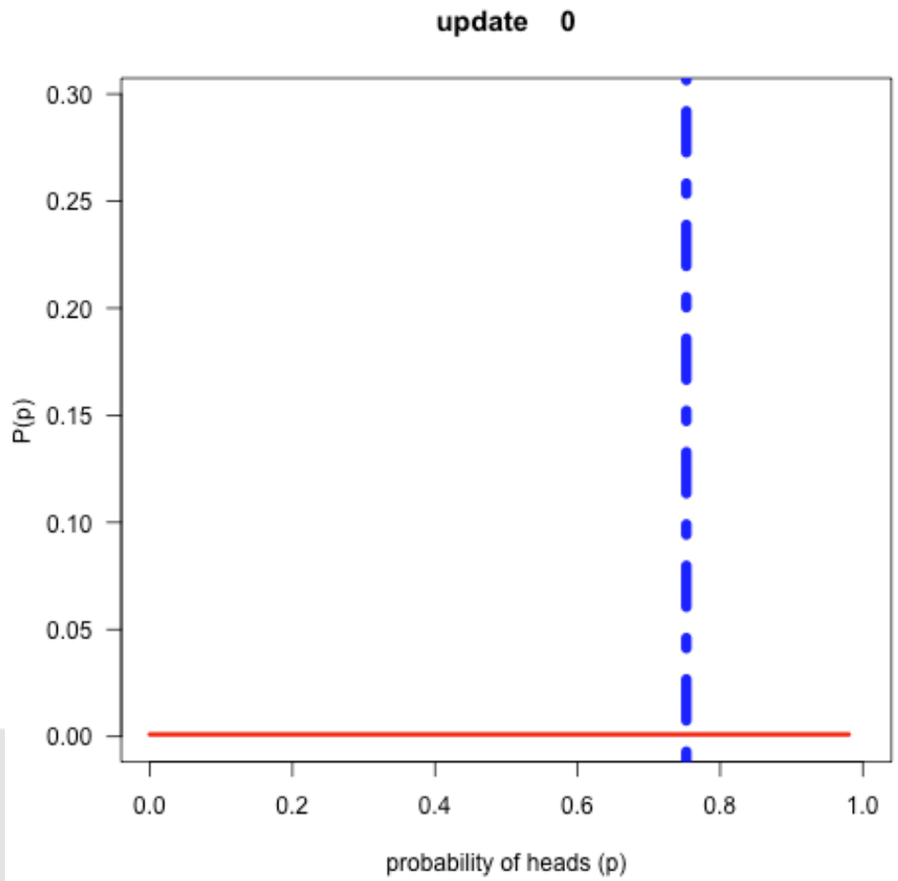
Bayesian Coin Flips

http://youtu.be/GTx0D8VY0CY

- Biased coin with unknown probability of heads (p)
- Perform N flips and update our belief after each flip using Bayes Theorem

$$P\left(p|heads\right) = \frac{P\left(heads|p\right)P\left(p\right)}{P\left(heads\right)}$$

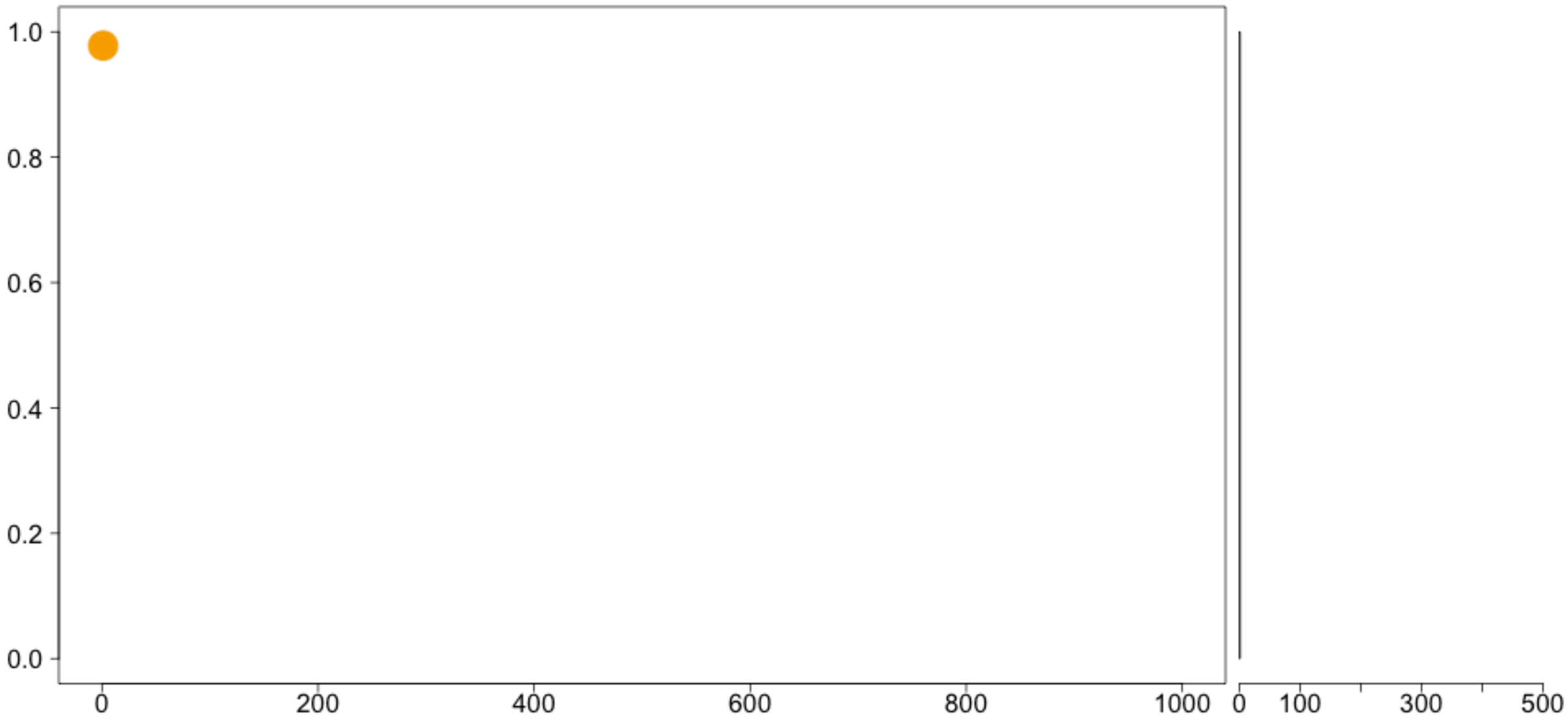
$$P\left(p|tails\right) = \frac{P\left(tails|p\right)P\left(p\right)}{P\left(tails\right)}$$




```
# Uninformative prior
prior = np.ones(bins, dtype='float')/bins
likelihood_heads = np.arange(bins)/float(bins)
likelihood_tails = 1-likelihood_heads
flips = np.random.choice(a=[True, False], size=flips, p=[0.75, 0.25])

for coin in flips:
    if coin:  # Heads
        posterior = prior * likelihood_heads
    else:  # Tails
        posterior = prior * likelihood_tails

    # Normalize
    posterior /= np.sum(posterior)

    # The posterior is now the new prior
    prior = posterior
```

coins.py

Law of Large Numbers



Central Limit Theorem

- As $n \to \infty$ the random variables:

$$\sqrt{n}\left(S_n - \mu\right)$$

- with:

$$S_n = \frac{1}{n}\sum_i x_i$$

- converge to a normal distribution:

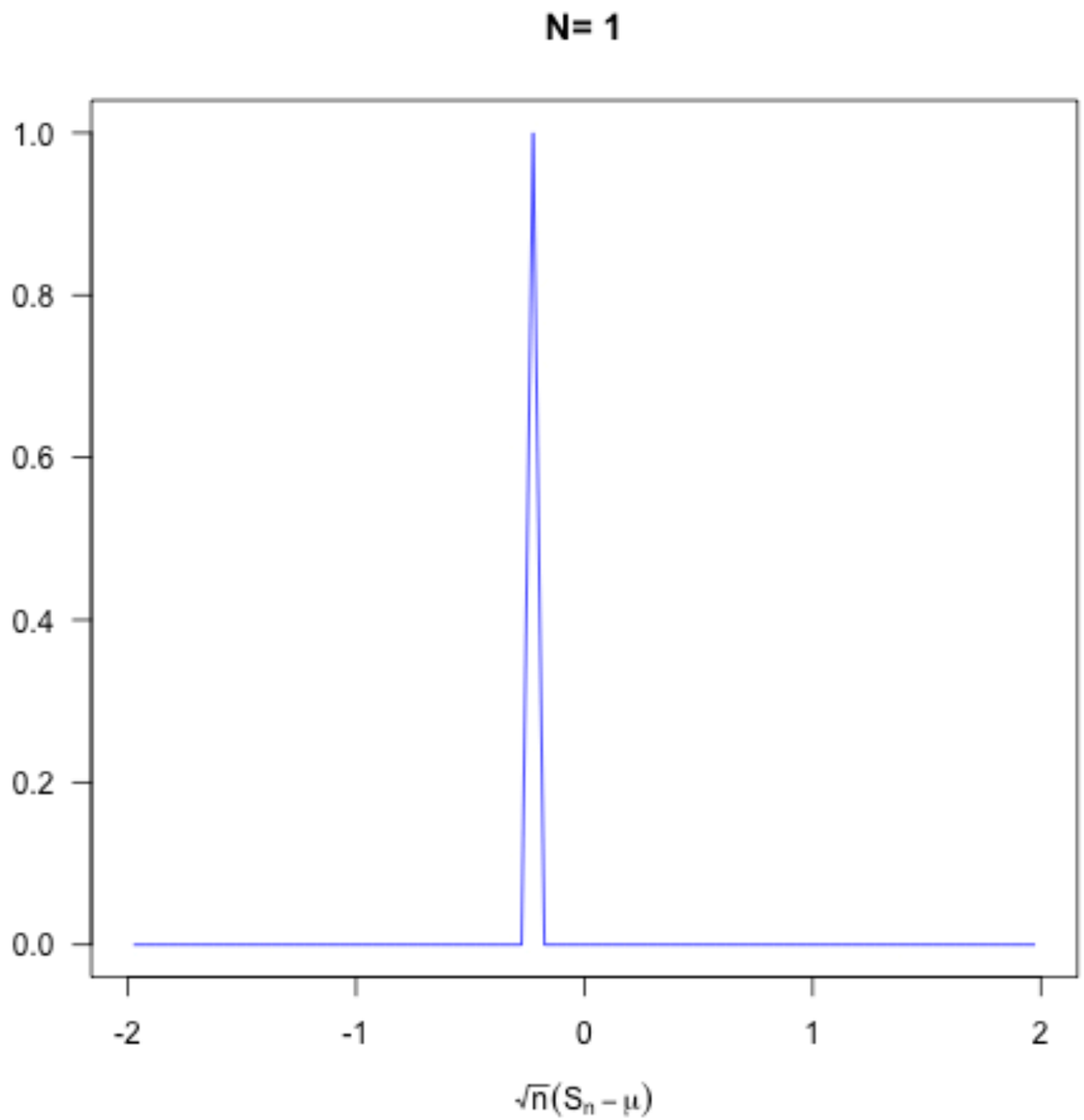
$$\mathcal{N}\left(0, \sigma^2\right)$$

- after some manipulations, we find:

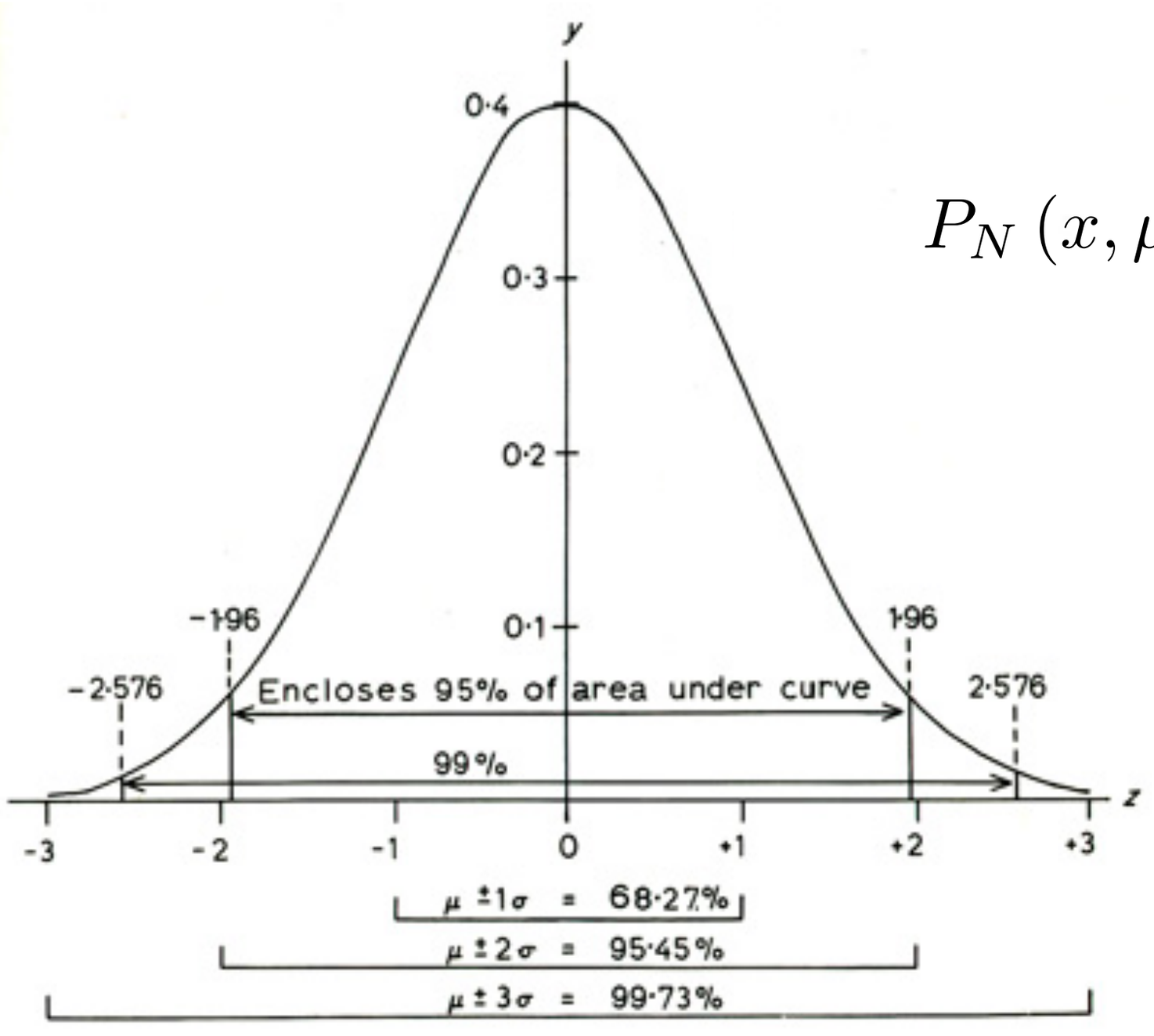
$$S_n \sim \mu + \frac{\mathcal{N}\left(0, \sigma^2\right)}{\sqrt{n}} \to SE = \frac{\sigma}{\sqrt{n}}$$

The estimation of the mean converges to the true mean with the square root of the number of samples

Central Limit Theorem



Gaussian Distribution



$$P_N\left(x,\mu,\sigma\right)=\frac{1}{\sqrt{2}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Experimental Measurements

- Experimental errors commonly assumed gaussian distributed

- Many experimental measurements are actually averages:

    - Instruments have a finite response time and the quantity of interest varies quickly over time

    - Stochastic Environmental factors

    - Etc

MLE - Fitting a theoretical function to experimental data

- In an experimental measurement, we expect (CLT) the experimental values to be normally distributed around the theoretical value with a certain variance. Mathematically, this means:

$$y - f\left(x\right) \approx \frac{1}{\sqrt{2\sigma^2}} \exp\left[-\frac{\left(y - f\left(x\right)\right)^2}{2\sigma^2}\right]$$

- where y are the experimental values and $f\left(x\right)$ the theoretical ones. The likelihood is then:

$$\mathcal{L} = -\frac{N}{2}\log\left[2\sigma^2\right] - \sum_i \left[\frac{\left(y_i - f\left(x_i\right)\right)^2}{2\sigma^2}\right]$$

- Where we see that to maximize the likelihood we must minimize the sum of squares

Least Squares Fitting

MLE - Linear Regression

- Let's say we want to fit a straight line to a set of points:

$$y = w \cdot x + b$$

- The Likelihood function then becomes:

$$\mathcal{L} = -\frac{N}{2}\log\left[2\sigma^2\right] - \sum_i \left[\frac{\left(y_i - w \cdot x_i - b\right)^2}{2\sigma^2}\right]$$

- With partial derivatives:

$$\frac{\partial \mathcal{L}}{\partial w} = \sum_i \left[2x_i\left(y_i - w \cdot x_i - b\right)\right]$$

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_i \left[\left(y_i - w \cdot x_i - b\right)\right]$$

- Setting to zero and solving for $\hat{w}$ and $\hat{b}$:

$$\hat{w} = \frac{\sum_i \left(x_i - \langle x \rangle\right)\left(y_i - \langle y \rangle\right)}{\sum_i \left(x_i - \langle x \rangle\right)^2}$$

$$\hat{b} = \langle y \rangle - \hat{w}\langle x \rangle$$

MLE for Linear Regression

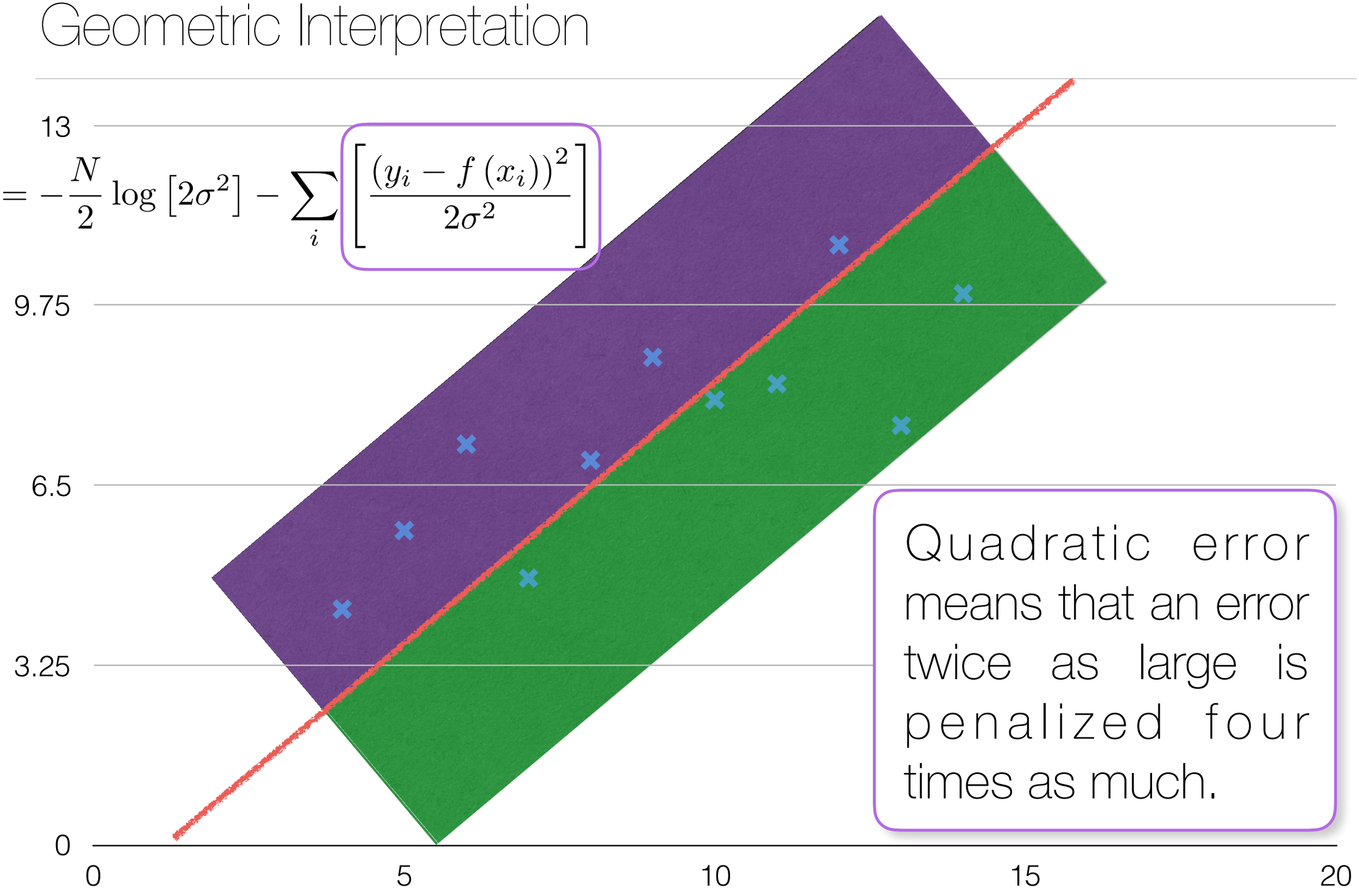
```python
from __future__ import print_function
import sys
import numpy as np
from scipy import optimize

data = np.loadtxt(sys.argv[1])

x = data.T[0]
y = data.T[1]

meanx = np.mean(x)
meany = np.mean(y)

w = np.sum((x-meanx)*(y-meany))/np.sum((x-meanx)**2)
b = meany-w*meanx

print(w, b)

#We can also optimize the Likelihood expression directly
def likelihood(w):
    global x, y
    sigma = 1.0
    w, b = w

    return np.sum((y-w*x-b)**2)/(2*sigma)

w, b = optimize.fmin_bfgs(likelihood, [1.0, 1.0])

print(w, b)
```

MLElinear.py

@bgoncalves

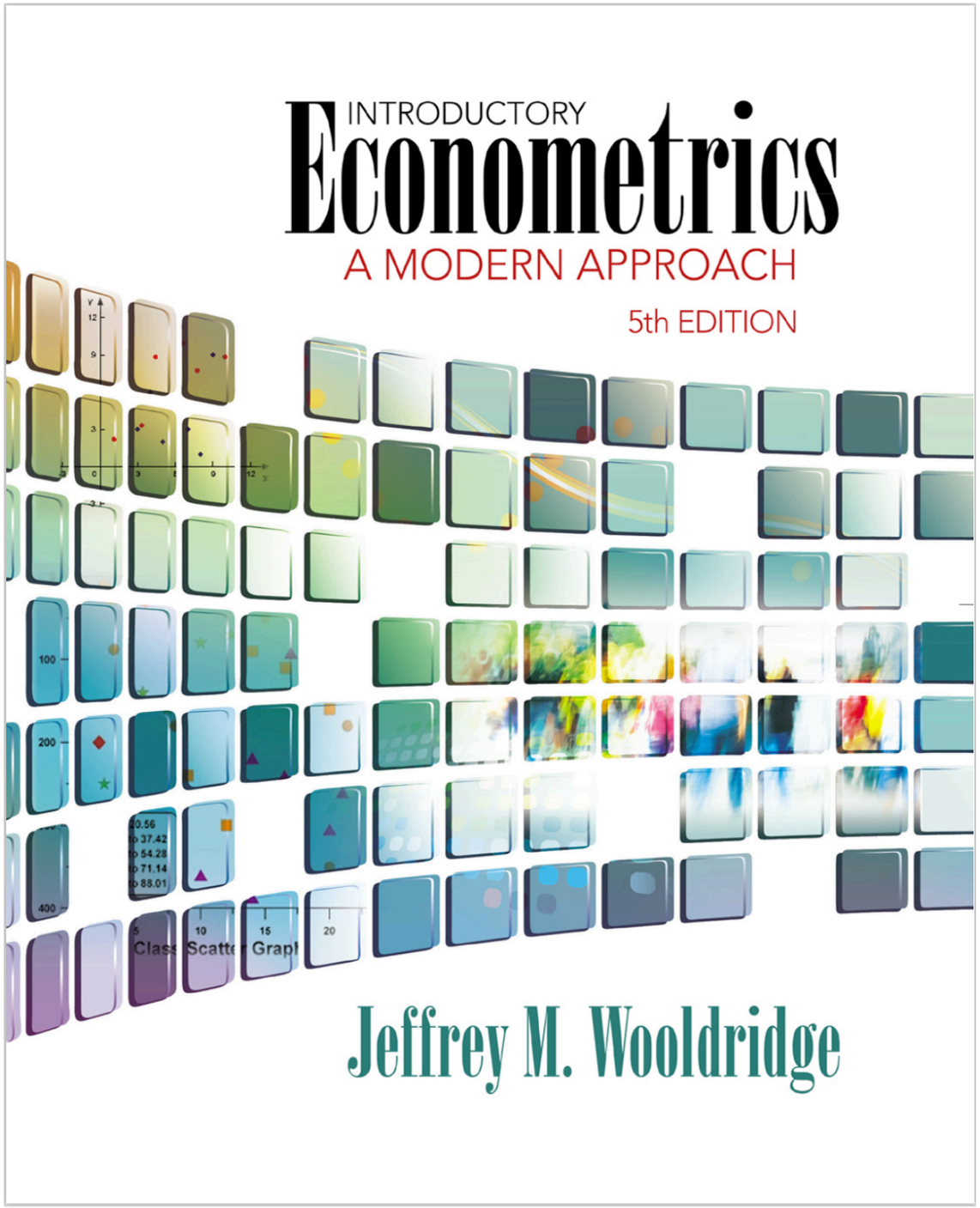
Geometric Interpretation

$$\mathcal{L} = -\frac{N}{2}\log\left[2\sigma^2\right] - \sum_i \left[\frac{\left(y_i - f\left(x_i\right)\right)^2}{2\sigma^2}\right]$$



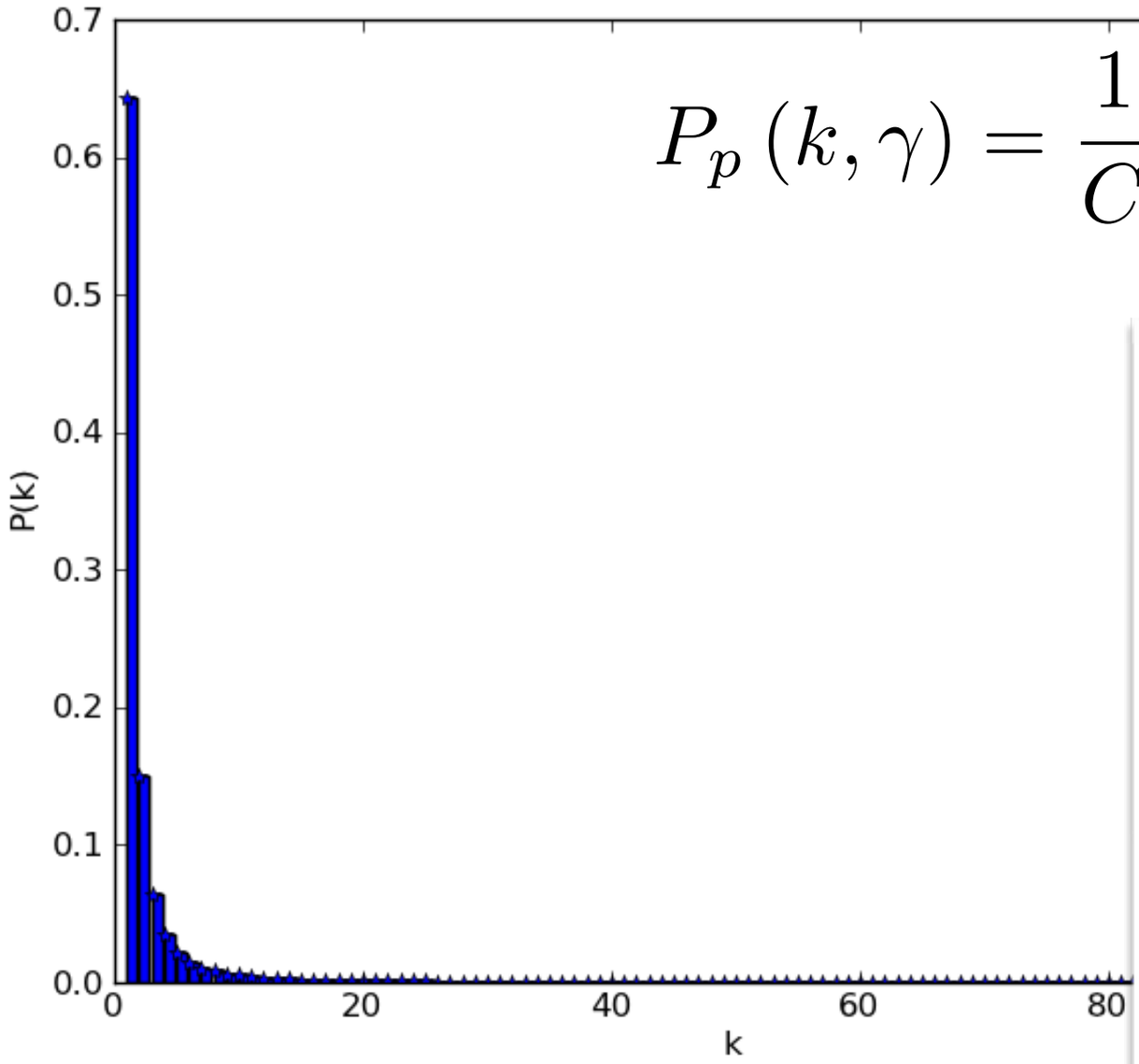
Quadratic error means that an error twice as large is penalized four times as much.

Multiple Regression...

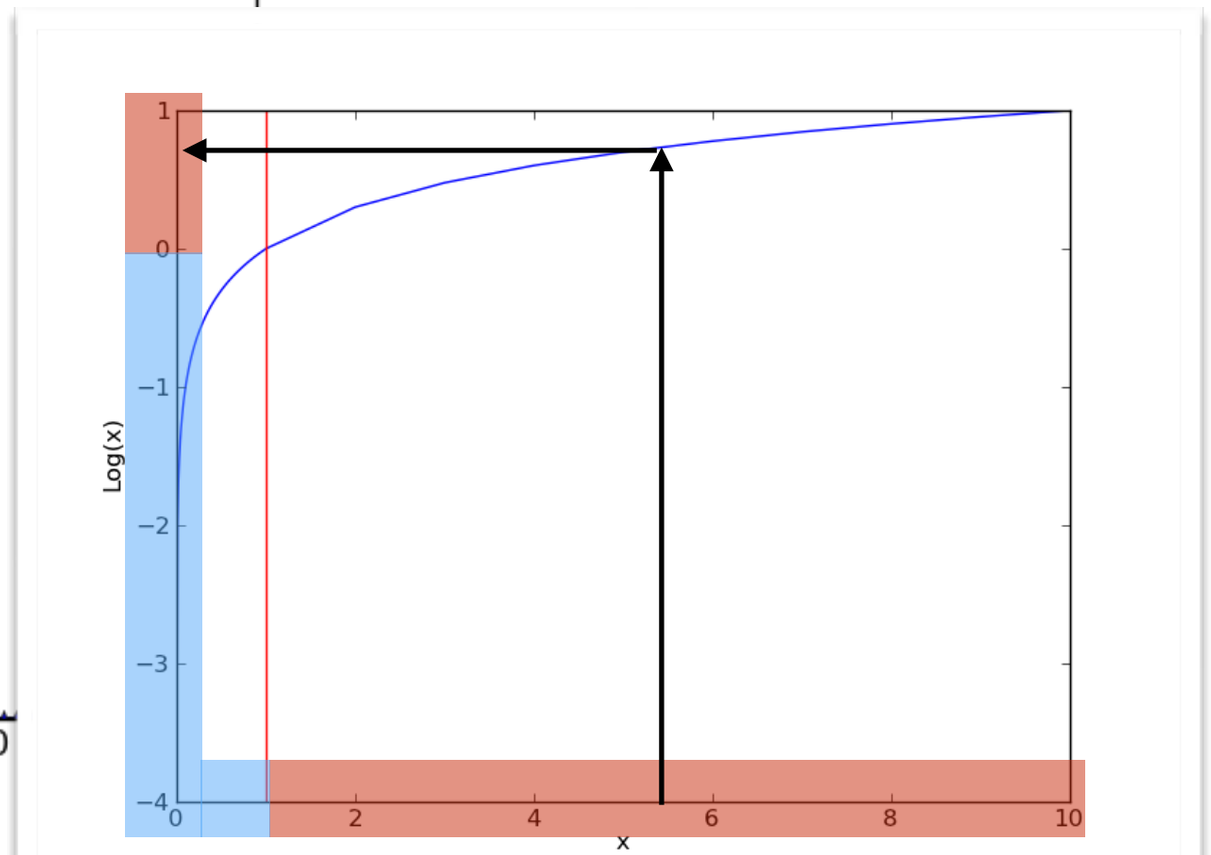


BRIEF CONTENTS

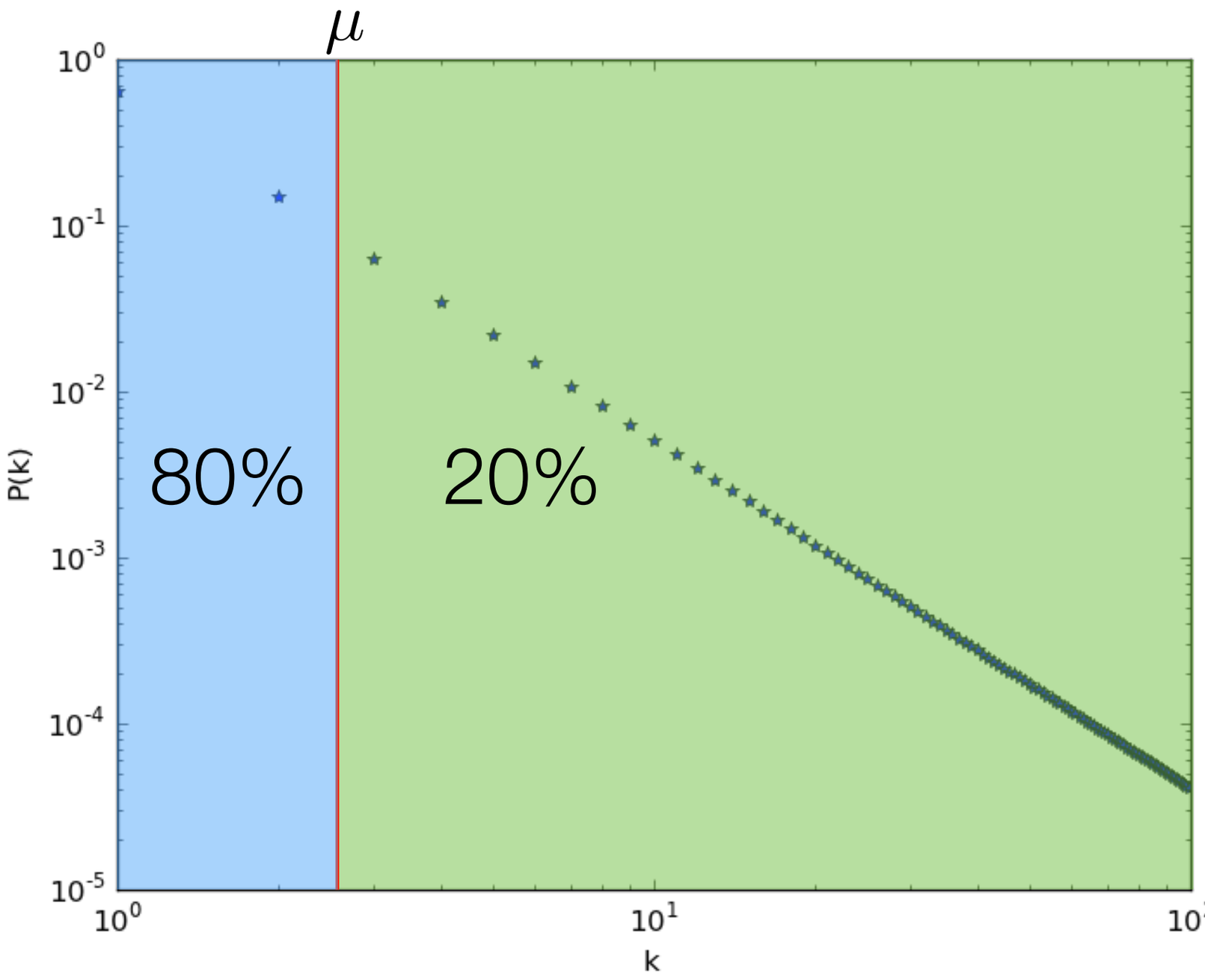
Broad-tailed distributions

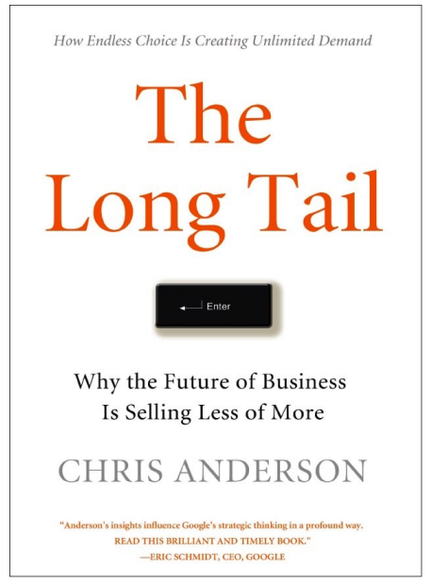


$$P_p\left(k,\gamma\right) = \frac{1}{C}k^{-\gamma}$$



(Almost) Everyone is below average!





MLE - Fitting a power-law to experimental data

SIAM Rev. 51, 661 (2009)

- We often find what look like power-law distributions in empirical data:

$$P(k) = \frac{\gamma - 1}{k_{min}} \left(\frac{k}{k_{min}} \right)^{-\gamma}$$

and we would like to find the right parameter values.

- The likelihood of any set of points is:

$$\mathcal{L} = \sum_i \log \left[\frac{\gamma - 1}{k_{min}} \left(\frac{k_i}{k_{min}} \right)^{-\gamma} \right]$$

- And maximizing, we find:

$$\gamma = 1 + n \left[\sum_i \log \left(\frac{k_i}{k_{min}} \right) \right]^{-1}$$

- with a standard error of:

$$SE = \frac{\gamma - 1}{\sqrt{n}}$$

3 Types of Machine Learning

- Unsupervised Learning
  - Autonomously learn an good representation of the dataset
  - Find clusters in input
- Supervised Learning
  - Predict output given input
  - Training set of known inputs and outputs is provided
- ~~Reinforcement Learning~~
  - Learn sequence of actions to maximize payoff
  - Discount factor for delayed rewards

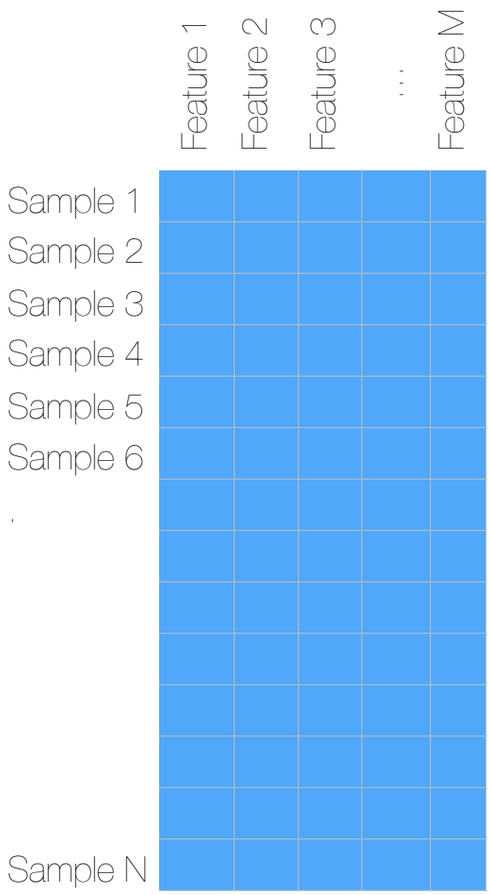
Data Normalization

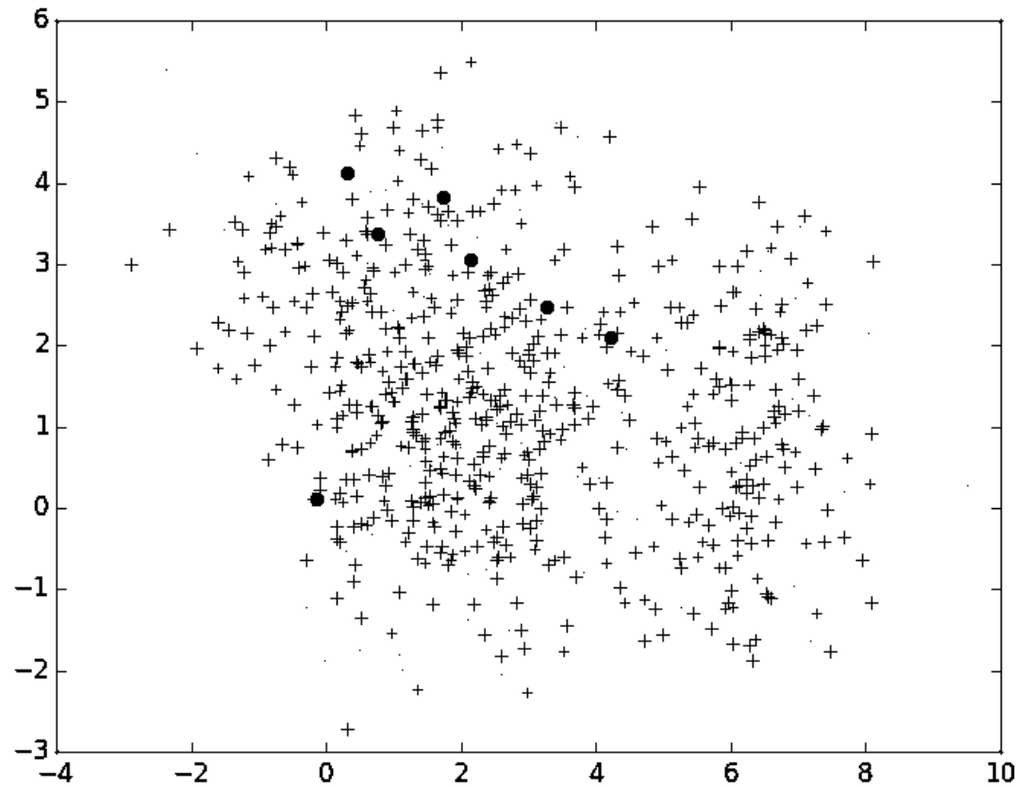
- The range of raw data values can vary widely.
- Using feature with very different ranges in the same analysis can cause numerical problems. Many algorithms are linear or use euclidean distances that are heavily influenced by the numerical values used (cm vs km, for example)
- To avoid difficulties, it's common to rescale the range of all features in such a way that each feature follows within the same range.
- Several possibilities:
  - Rescaling - $\hat{x} = \frac{x - x_{min}}{x_{max} - x_{min}}$
  - Standardization - $\hat{x} = \frac{x - \mu_x}{\sigma_x}$
  - Normalization - $\hat{x} = \frac{x}{||x||}$
  - In the rest of the discussion we will assume that the data has been normalized in some suitable way

Unsupervised Learning

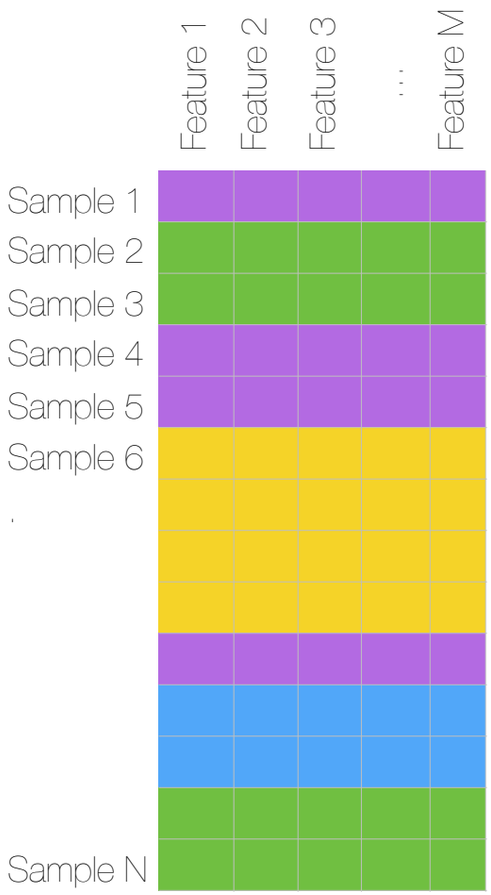
- Extracting patters from data
  - K-Means
  - Expectation Maximization
  - PCA

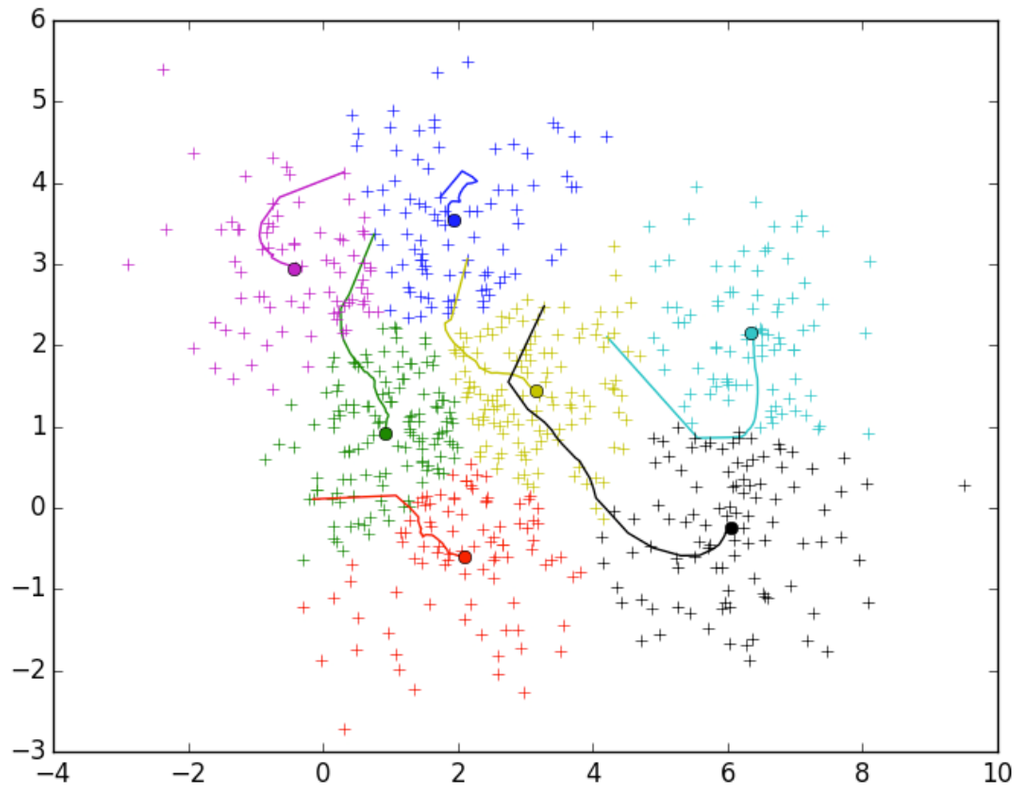
Clustering


Feature 1
Feature 2
Feature 3
. . .
Feature M
Sample 1
Sample 2
Sample 3
Sample 4
Sample 5
Sample 6
Sample N

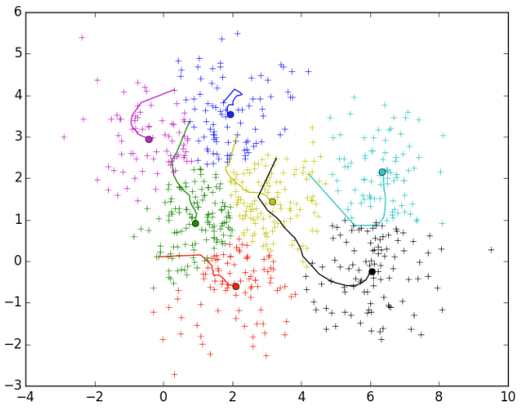

6
5
4
3
2
1
0
−1
−2
−3
−4
−2
0
2
4
6
8
10

Clustering


Feature 1
Feature 2
Feature 3
. . .
Feature M
Sample 1
Sample 2
Sample 3
Sample 4
Sample 5
Sample 6
.
Sample N


6
5
4
3
2
1
0
−1
−2
−3
−4
−2
0
2
4
6
8
10

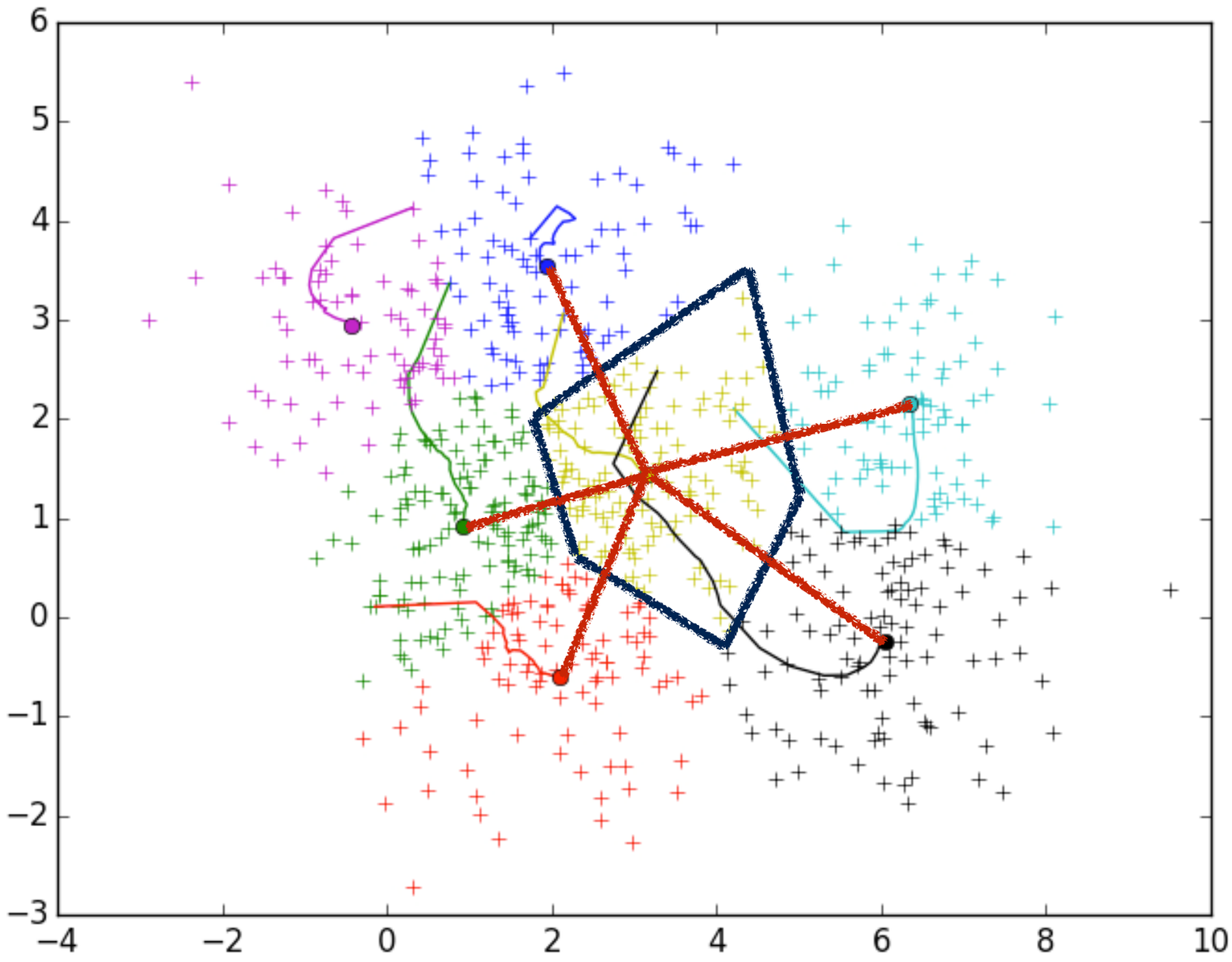
K-Means



- Choose k randomly chosen points to be the centroid of each cluster
- Assign each point to belong the cluster whose centroid is closest
- Recompute the centroid positions (mean cluster position)
- Repeat until convergence

K-Means: Structure

Voronoi Tesselation



K-Means: Convergence

- How to quantify the "quality" of the solution found at each iteration, n?

- Measure the "Inertia", the square intra-cluster distance:

$$I_n = \sum_{i=0}^{N} \|x_i - \mu_i\|^2$$

where μ_i are the coordinates of the centroid of the cluster to which x_i is assigned.

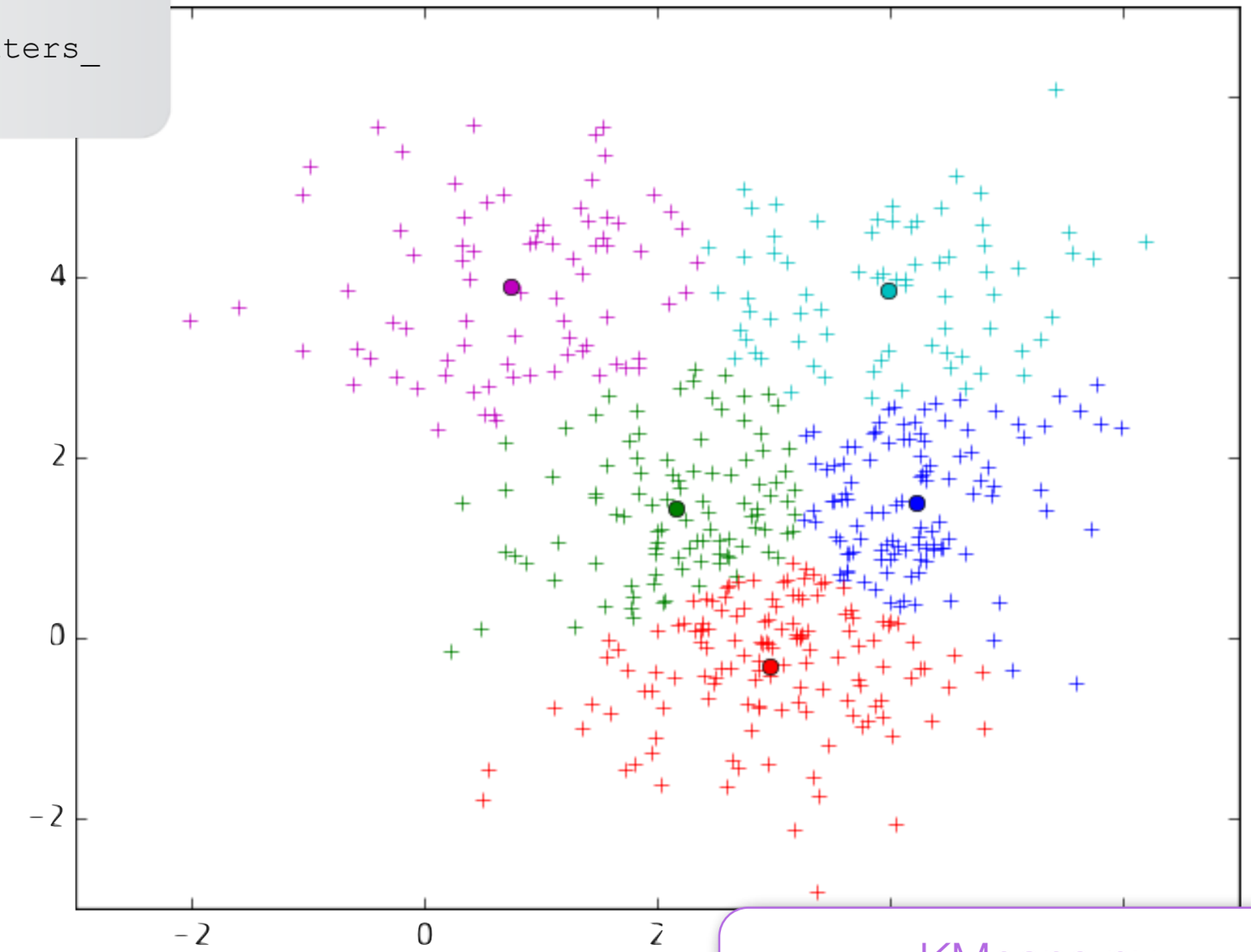
- Smaller values are better

- Can stop when the relative variation is smaller than some value

$$\frac{I_{n+1} - I_n}{I_n} < tol$$

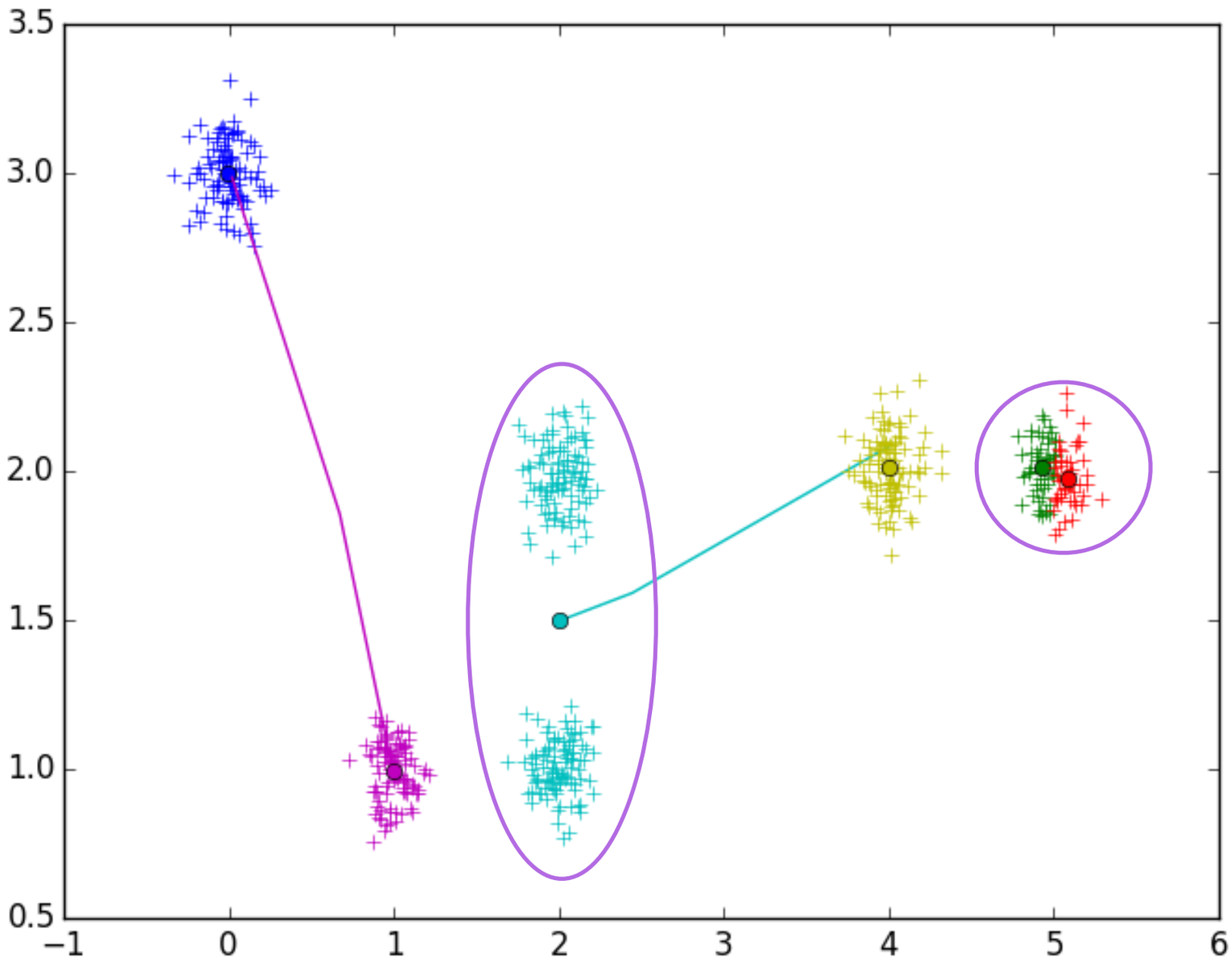
K-Means: sklearn

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=nclusters)
kmeans.fit(data)

centroids = kmeans.cluster_centers_
labels = kmeans.labels_
```

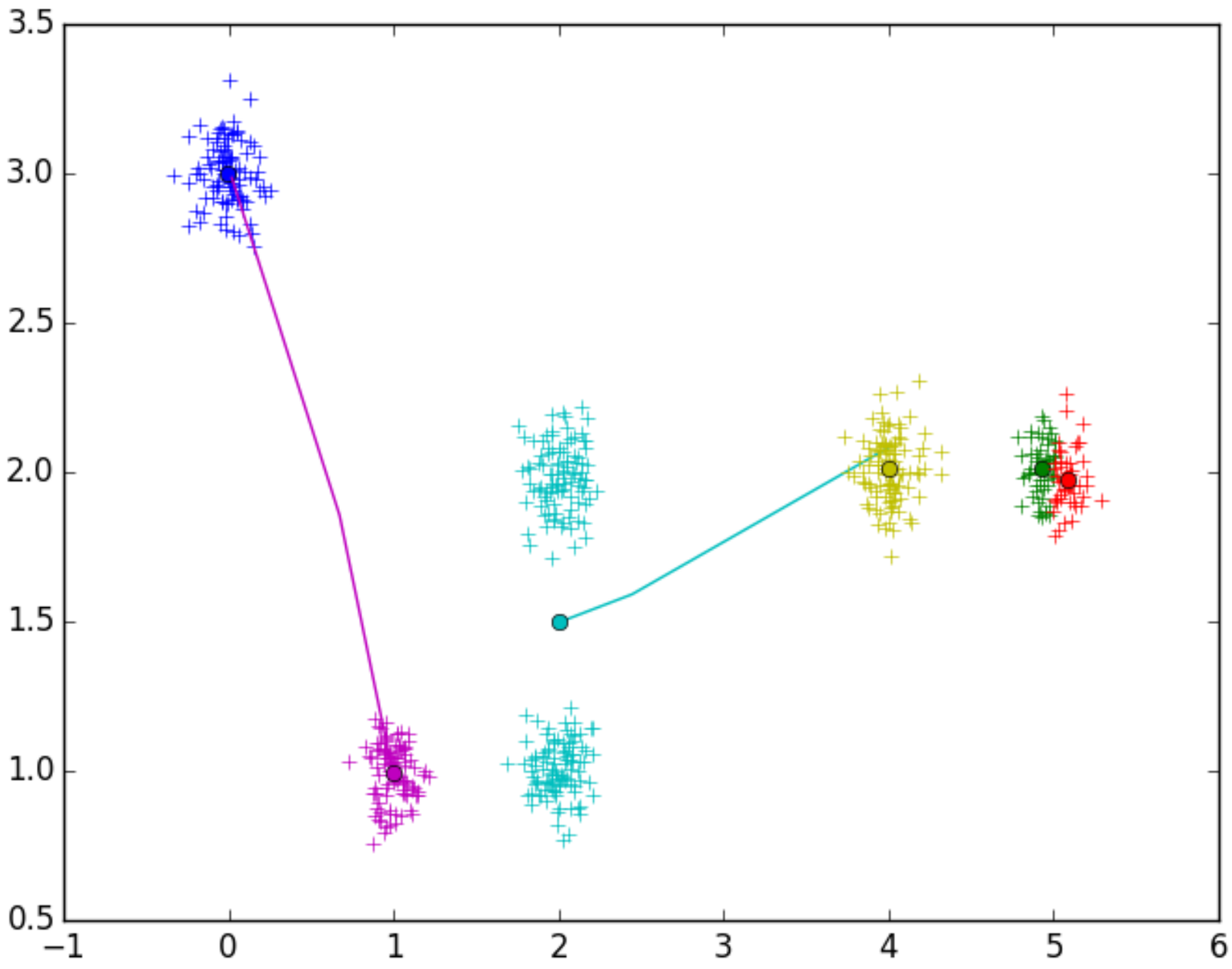


KMeans.py

K-Means: Limitations



K-Means: Limitations



- No guarantees about Finding "Best" solution
- Each run can find different solution
- No clear way to determine "k"

Silhouettes

- For each point x_i define $a_c\left(x_i\right)$ as:

$$a_c\left(x_i\right) = \frac{1}{N_c}\sum_{j \in c}\left\|x_i - x_j\right\|$$

the average distance between point x_i and every other point within cluster c

- Let $b\left(x_i\right)$ be:

$$b\left(x_i\right) = \min_{c \neq c_i} a_c\left(x_i\right)$$

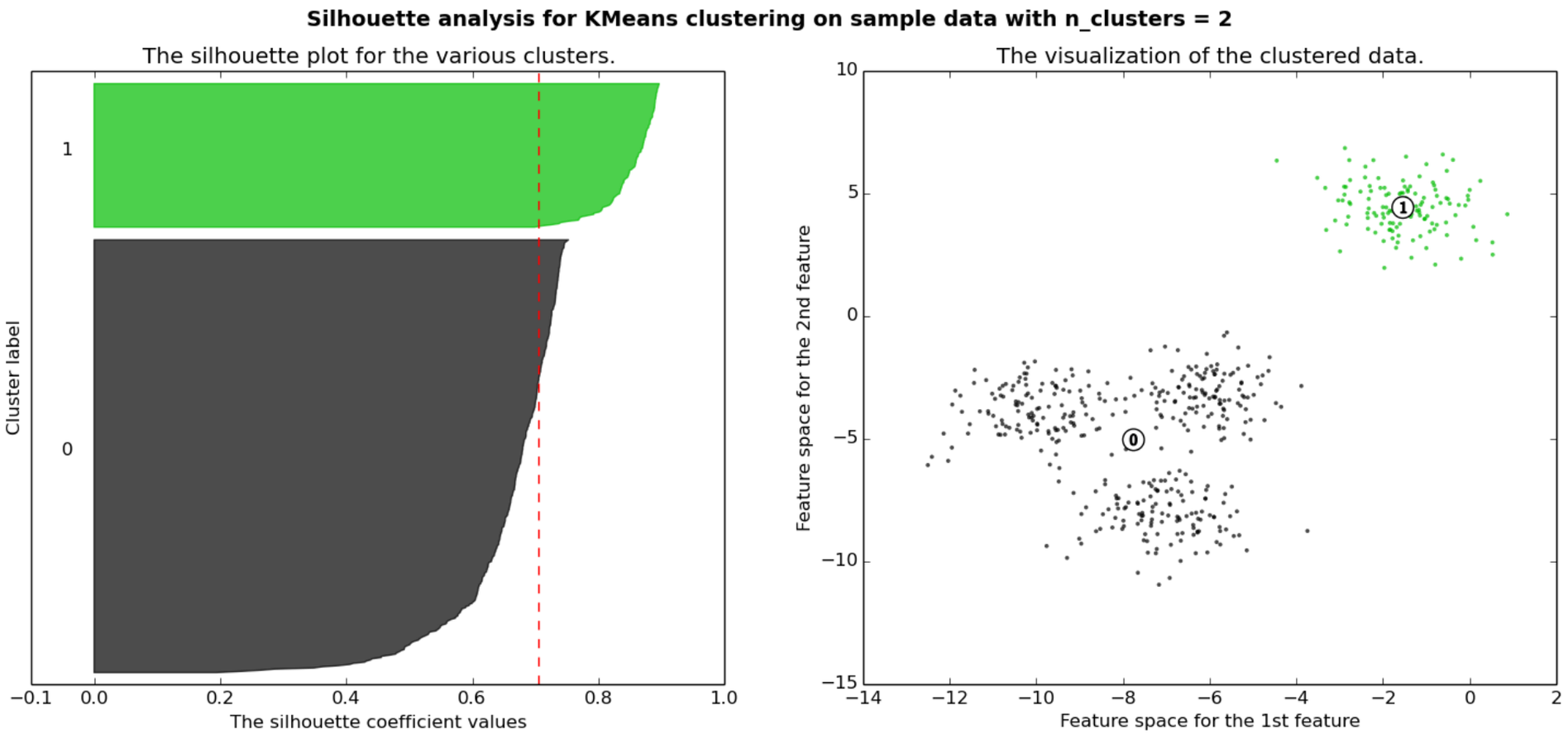
the minimum value of $a_c\left(x_i\right)$ excluding c_i

- The silhouette of x_i is then:

$$s\left(x_i\right) = \frac{b\left(x_i\right) - a_{c_i}\left(x_i\right)}{\max\left\{b\left(x_i\right), a_{c_i}\left(x_i\right)\right\}}$$

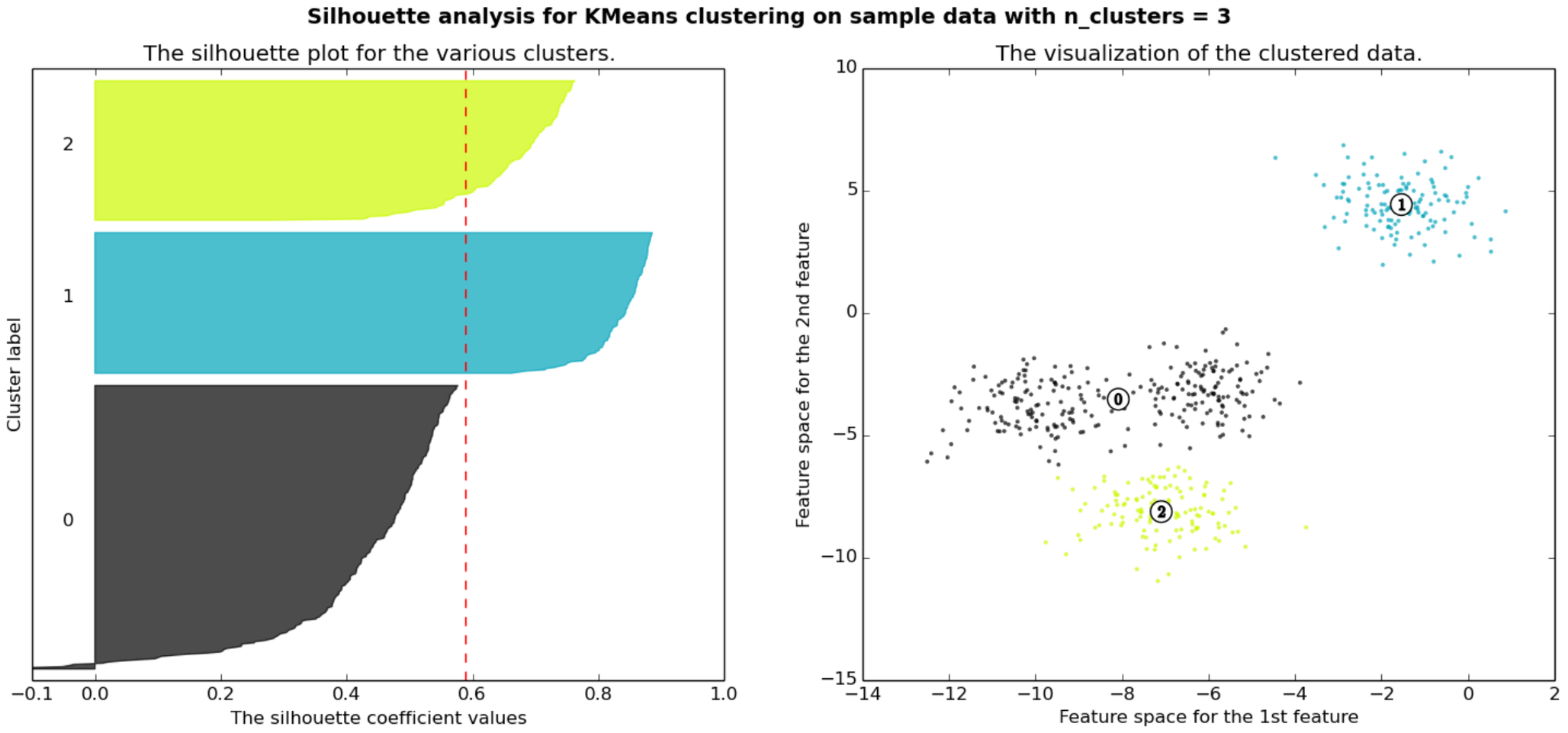
Silhouettes

http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html



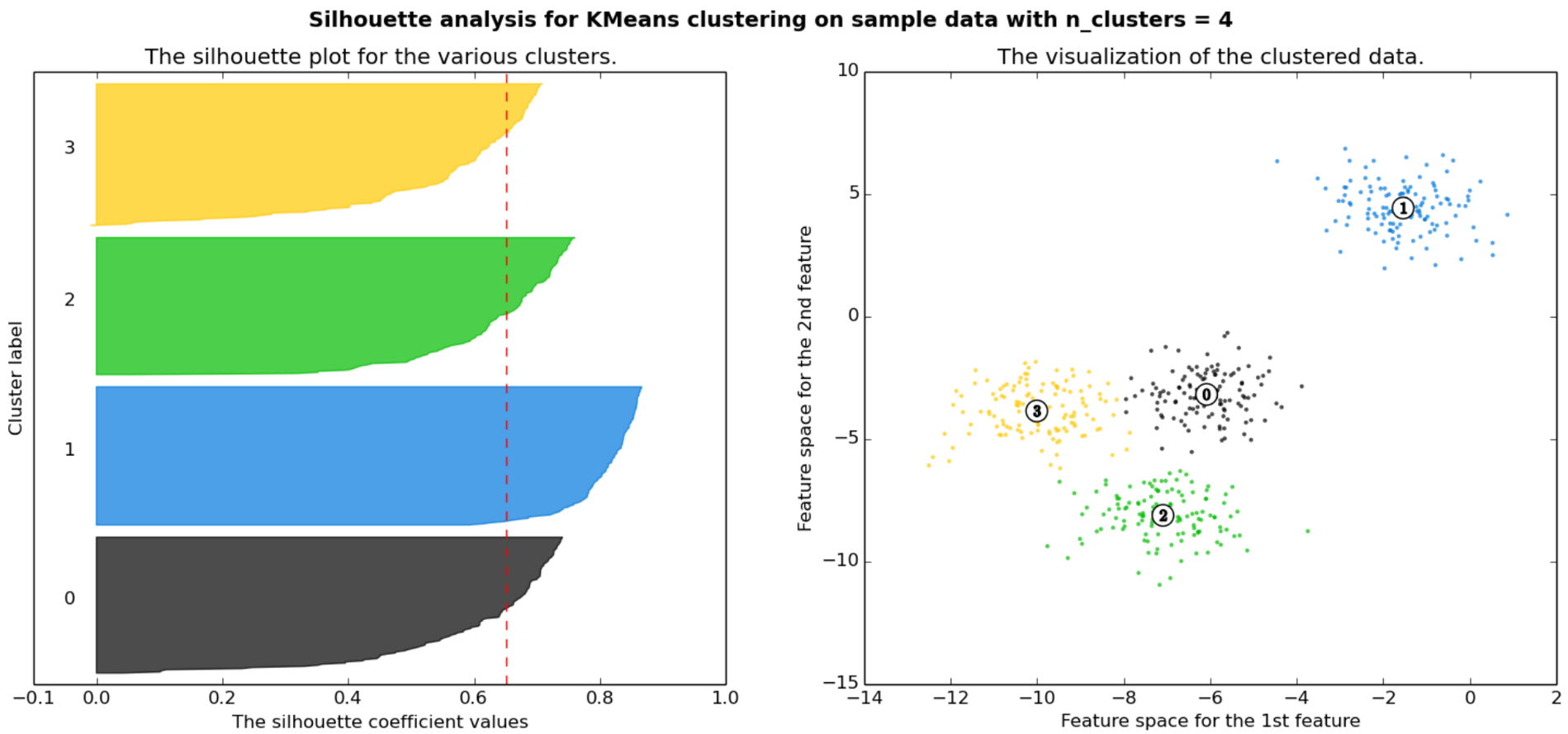
Silhouettes

http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html



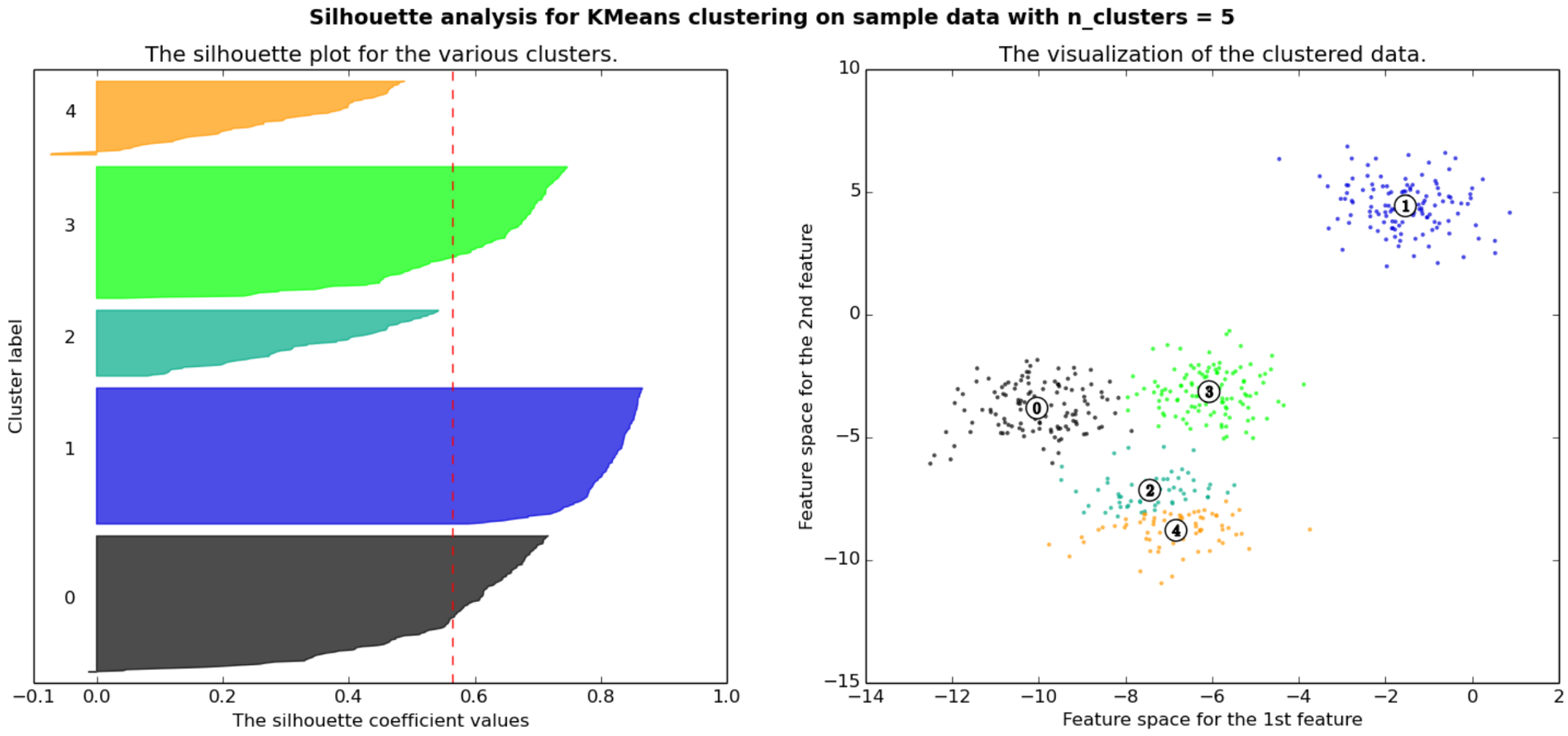
Silhouettes

http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html



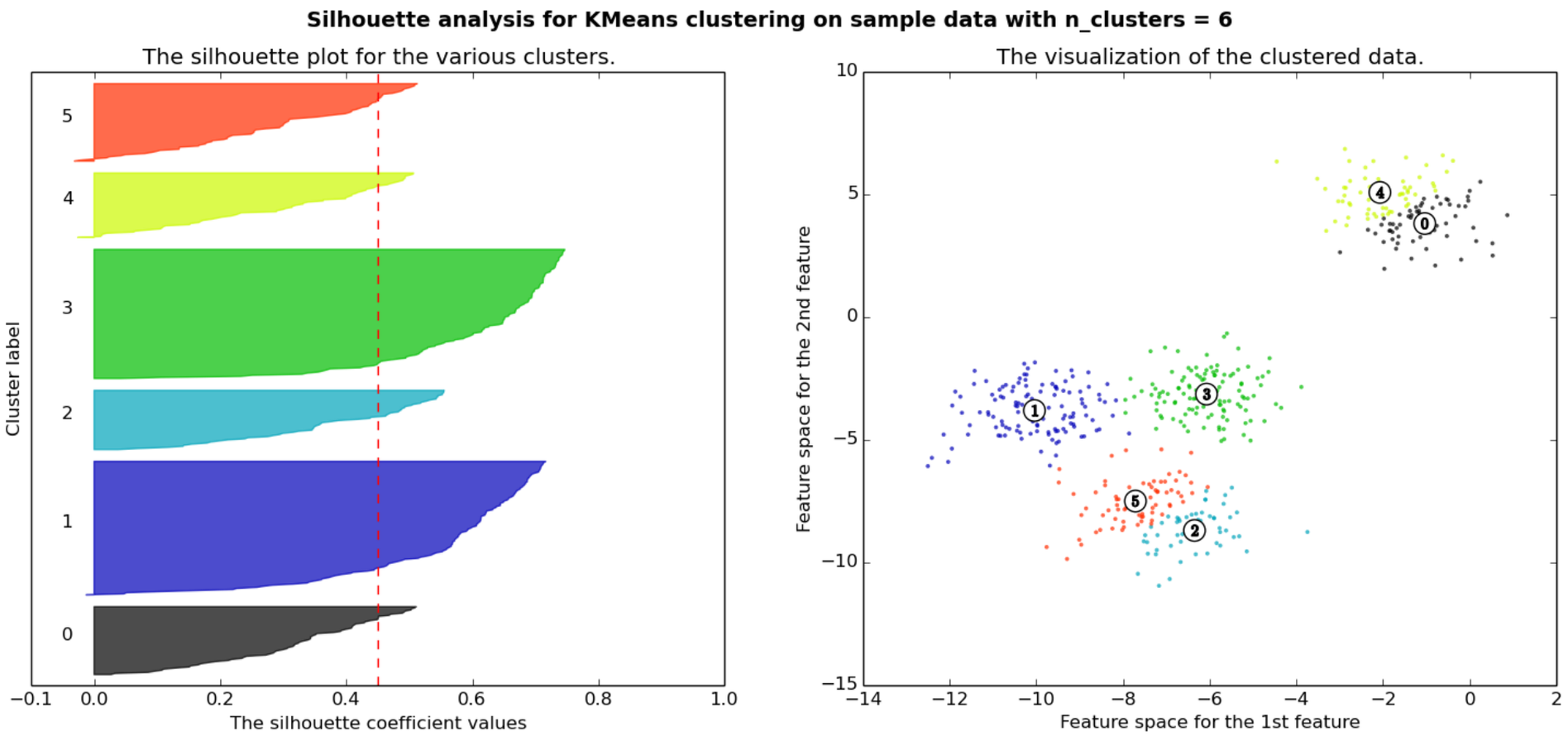
Silhouettes

http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html



Silhouettes

http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html



Expectation Maximization

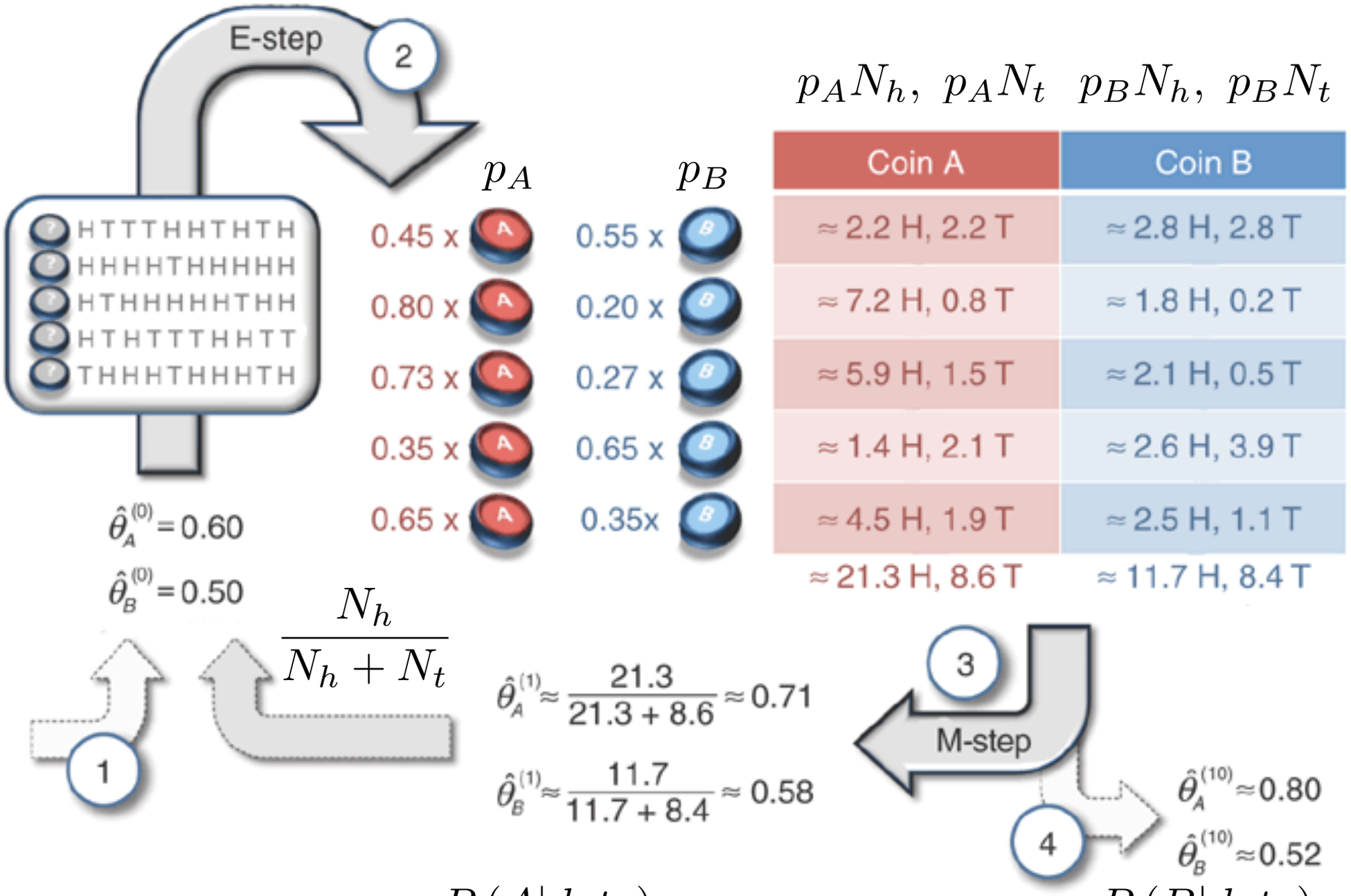
- Iterative algorithm to learn parameter estimates in models with unobserved latent variables

- Two steps for each iteration
  - **Expectation:** Calculate the likelihood of the data given current parameter estimate
  - **Maximization:** Find the parameter values that maximize the likelihood

- Stop when the relative variation of the parameter estimates is smaller than some value

Expectation Maximization

Nature BioTech 26, 897 (2008)



@bgoncalves

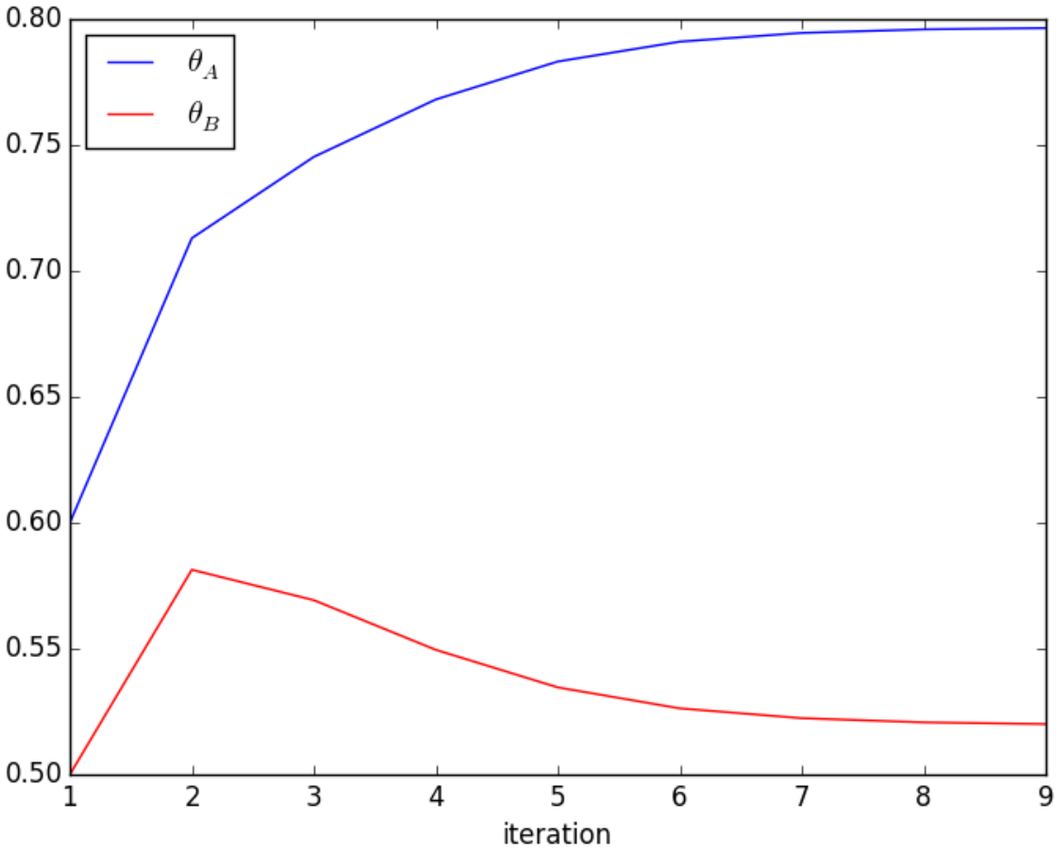
$$p_A = \frac{P(A|data)}{P(A|data) + P(B|data)} \qquad p_B = \frac{P(B|data)}{P(A|data) + P(B|data)}$$

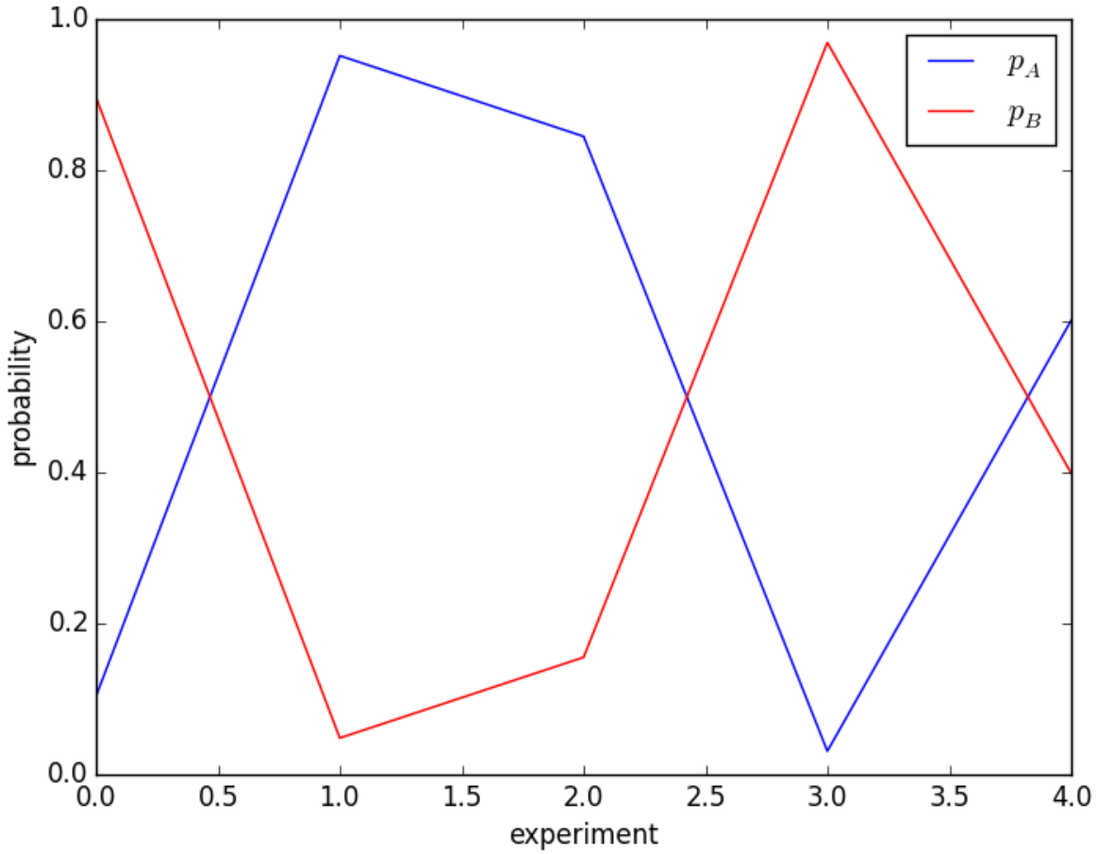
Expectation Maximization

```
while (improvement > delta):
    expectation_A = np.zeros((5, 2), dtype=float)
    expectation_B = np.zeros((5, 2), dtype=float)

    for i in range(0, len(experiments)):
        e = experiments[i]  # i'th experiment
        ll_A = get_mn_likelihood(e, np.array([tA[-1], 1-tA[-1]]))
        ll_B = get_mn_likelihood(e, np.array([tB[-1], 1-tB[-1]]))

        weightA = ll_A/(ll_A + ll_B)
        weightB = ll_B/(ll_A + ll_B)

        expectation_A[i] = np.dot(weightA, e)
        expectation_B[i] = np.dot(weightB, e)

    tA.append(sum(expectation_A)[0] / sum(sum(expectation_A)))
    tB.append(sum(expectation_B)[0] / sum(sum(expectation_B)))

    improvement = max(abs(np.array([tA[-1], tB[-1]]) - np.array([tA[-2], tB[-2]])))
```


EM.py

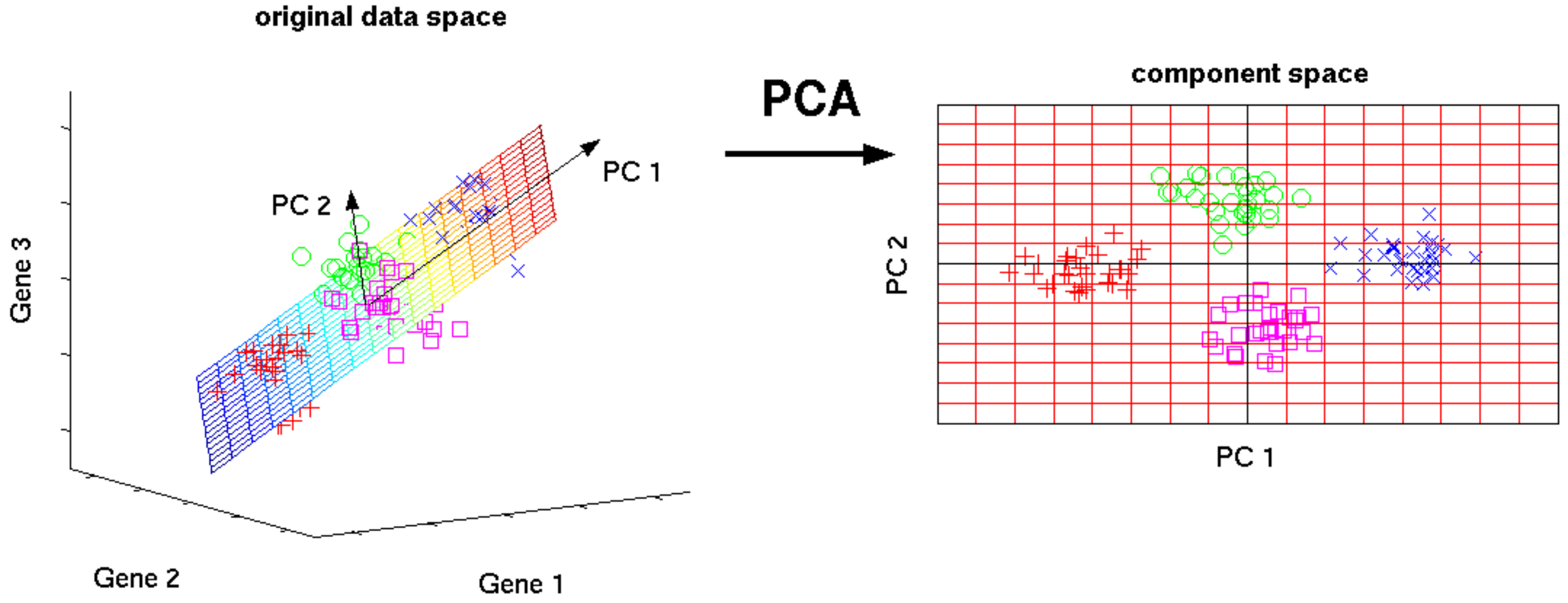
Expectation Maximization





Principle Component Analysis

- Finds the directions of maximum variance of the dataset
- Useful for dimensionality reduction
- Often used as preprocessing of the dataset



Principle Component Analysis

- The Principle Component projection, T, of a matrix A is defined as:

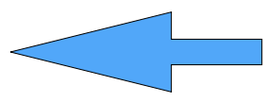
$$T = AW$$

- where W is the eigenvector matrix of:

$$A^T A$$

- and corresponds to the right singular vectors of A obtained by Singular Value Decomposition (SVD):

$$A = U\Sigma W^T$$



Generalization of Eigenvalue/ Eigenvector decomposition for non-square matrices.

- So we can write:

$$T = U\Sigma W^T W \equiv U\Sigma$$

- Showing that the Principle Component projection corresponds to the left singular vectors of A scaled by the respective singular values Σ

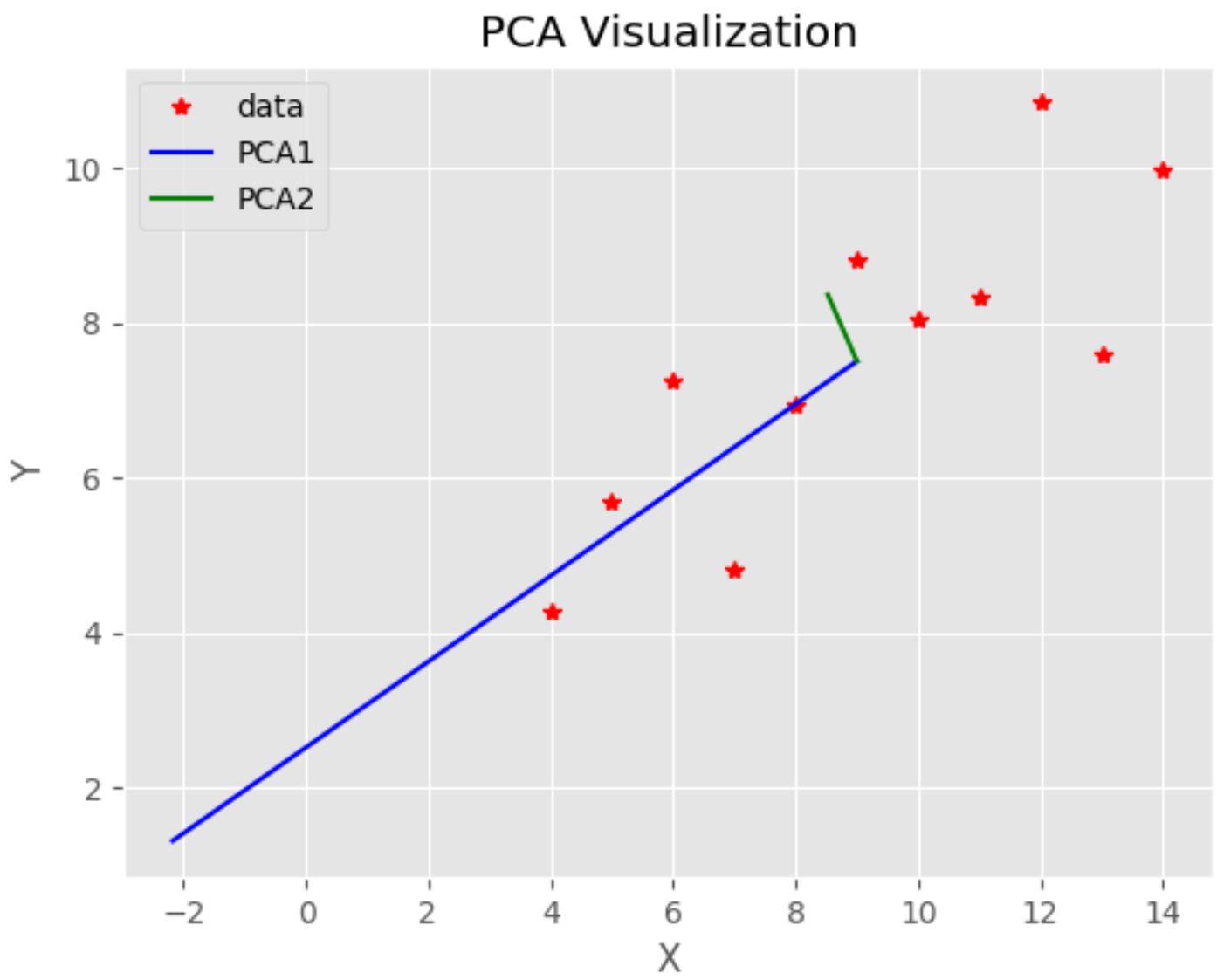
- Columns of T are ordered in order of decreasing variance.

Princ

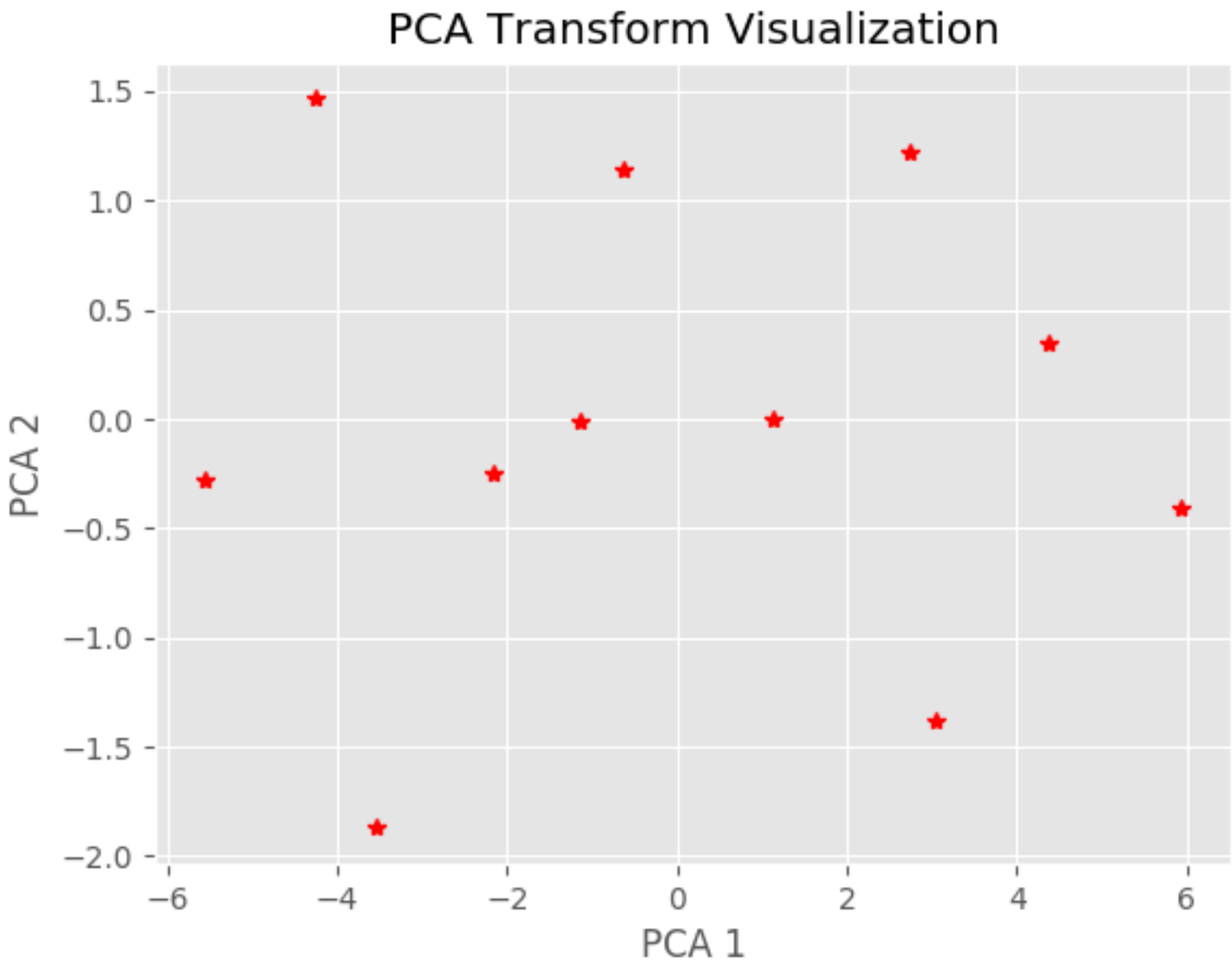
```python
import sys
from sklearn.decomposition import PCA
import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt(sys.argv[1])

x = data.T[0]
y = data.T[1]

pca = PCA()
pca.fit(data)

meanX = np.mean(x)
meanY = np.mean(y)

plt.style.use('ggplot')
plt.plot(x, y, 'r*')

plt.plot([meanX, meanX+pca.components_[0][0]*pca.explained_variance_[0]],
         [meanY, meanY+pca.components_[0][1]*pca.explained_variance_[0]], 'b-')
plt.plot([meanX, meanX+pca.components_[1][0]*pca.explained_variance_[1]],
         [meanY, meanY+pca.components_[1][1]*pca.explained_variance_[1]], 'g-')
plt.title('PCA Visualization')
plt.legend(['data', 'PCA1', 'PCA2'], loc=2)
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('PCA.png')
plt.close()

transform = pca.transform(data)
plt.plot(transform.T[0], transform.T[1], 'r*')
plt.title('PCA Transform Visualization')
plt.xlabel('PCA 1')
plt.ylabel('PCA 2')
plt.savefig('PCATransform.png')
plt.close()
```

PCA.py

Principle Component Analysis



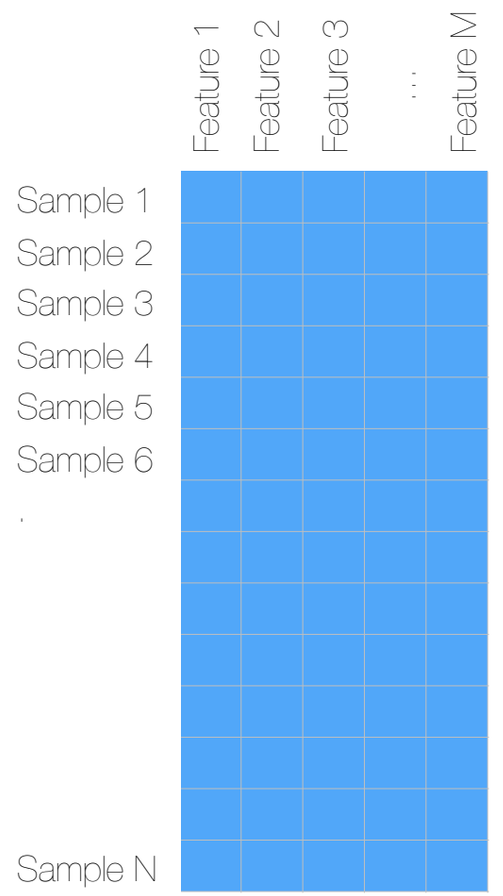
Principle Component Analysis



Supervised Learning

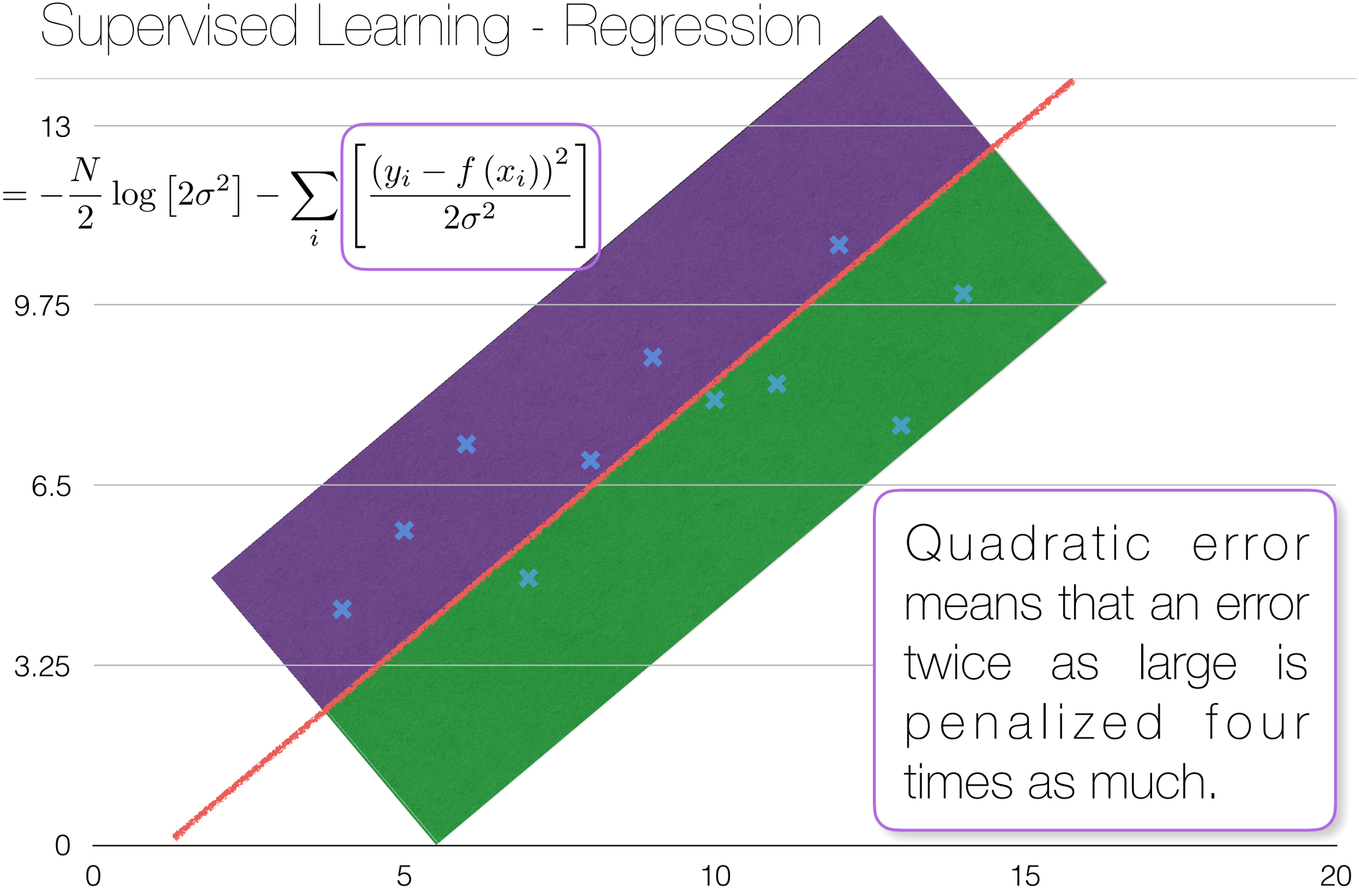
Supervised Learning - Regression

- Two fundamental types of problems:
  - Classification (discrete output value)
  - Regression (continuous output value)
- Dataset formatted as an NxM matrix of N samples and M features



Supervised Learning - Regression

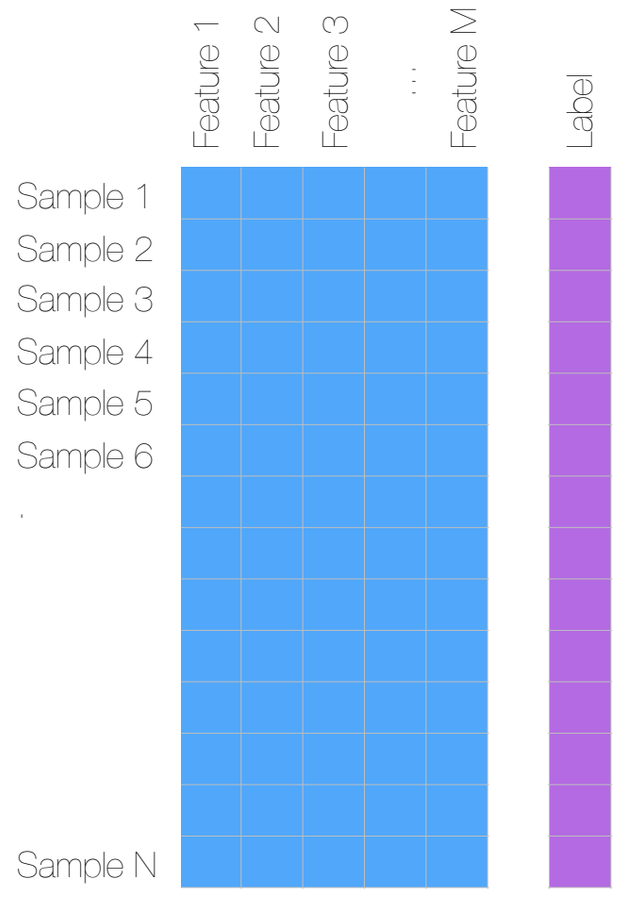
$$\mathcal{L} = -\frac{N}{2}\log\left[2\sigma^2\right] - \sum_i \left[\frac{\left(y_i - f\left(x_i\right)\right)^2}{2\sigma^2}\right]$$



Quadratic error means that an error twice as large is penalized four times as much.

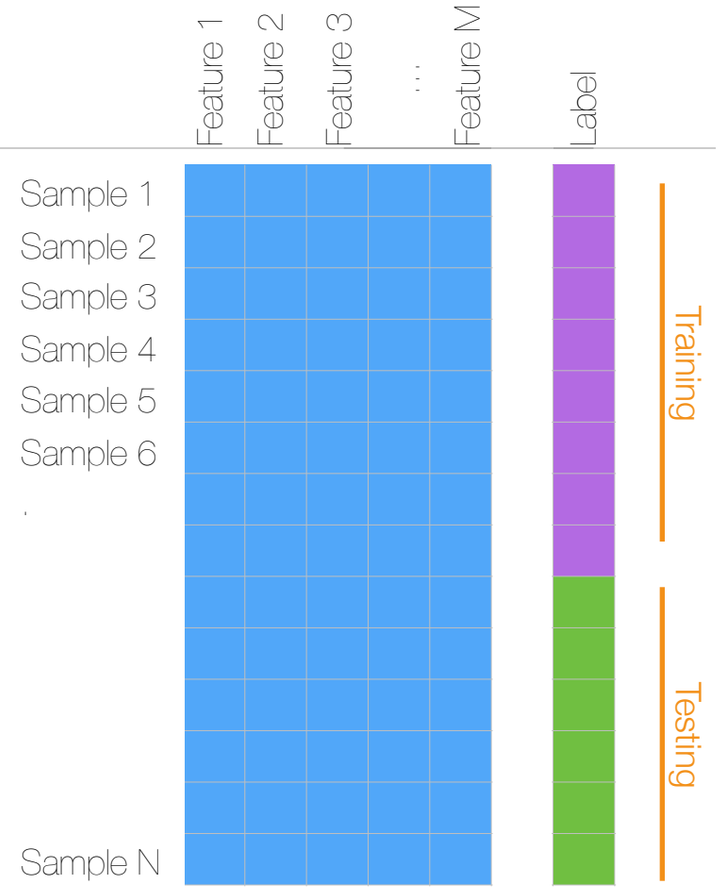
Supervised Learning - Classification

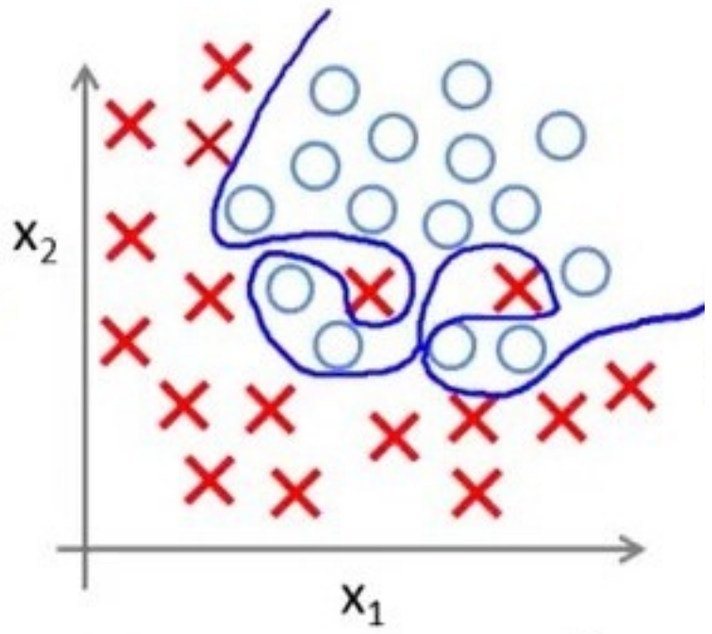
- Two fundamental types of problems:
  - Classification (discrete output value)
  - Regression (continuous output value)
- Dataset formatted as an NxM matrix of N samples and M features
- Each sample belongs to a specific class or has a specific label.
- The goal of classification is to predict to which class a previously unseen sample belongs to by learning defining regularities of each class
  - K-Nearest Neighbor
  - Support Vector Machine
  - Neural Networks



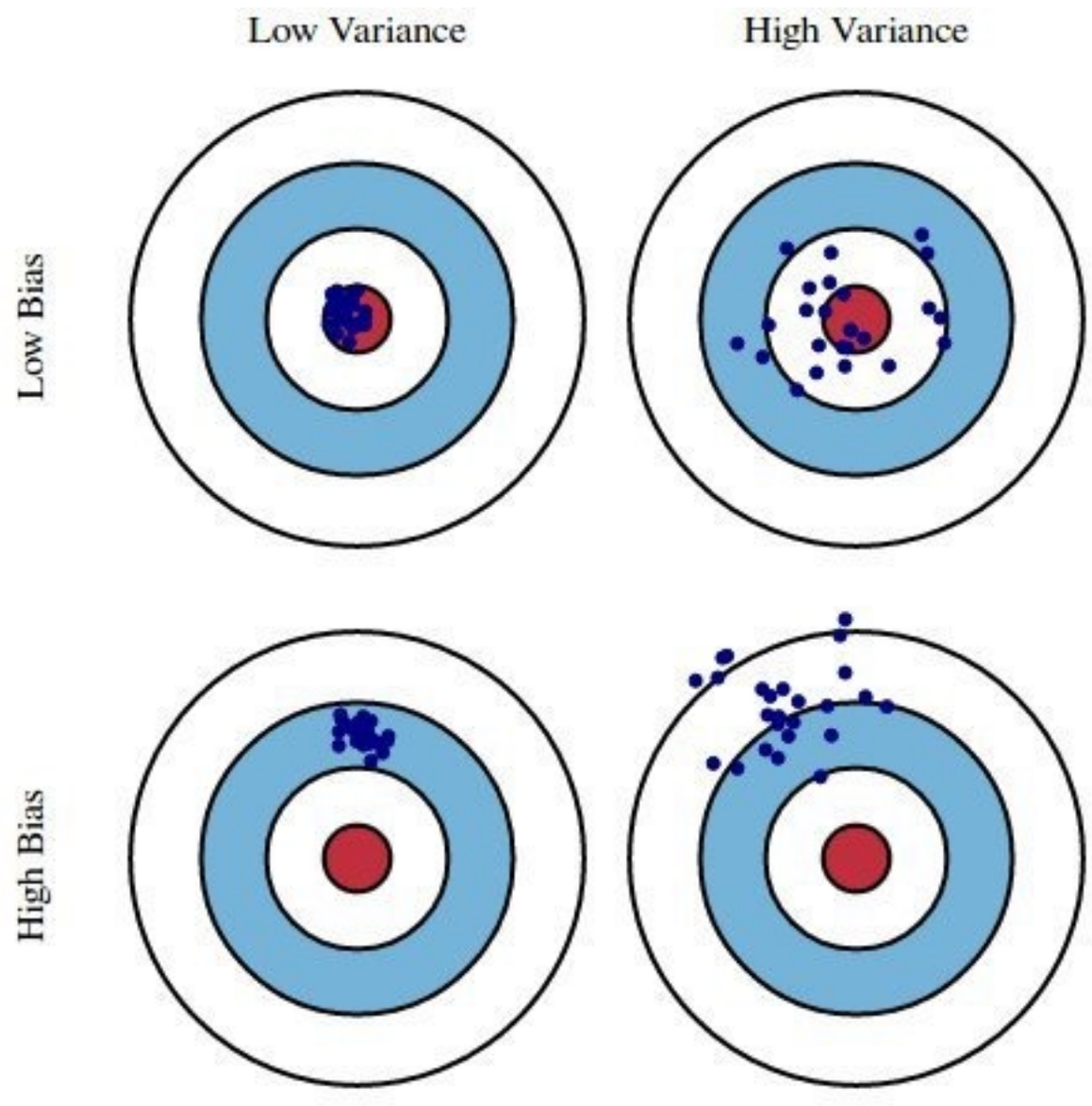
Supervised Learning - Overfitting

- "Learning the noise"
- "Memorization" instead of "generalization"
- How can we prevent it?
    - Split dataset into two subsets: Training and Testing
    - Train model using only the Training dataset and evaluate results in the previously unseen Testing dataset.
- Different heuristics on how to split:
    - Single split
    - k-fold cross validation: split dataset in k parts, train in k-1 and evaluate in 1, repeat k times and average results.

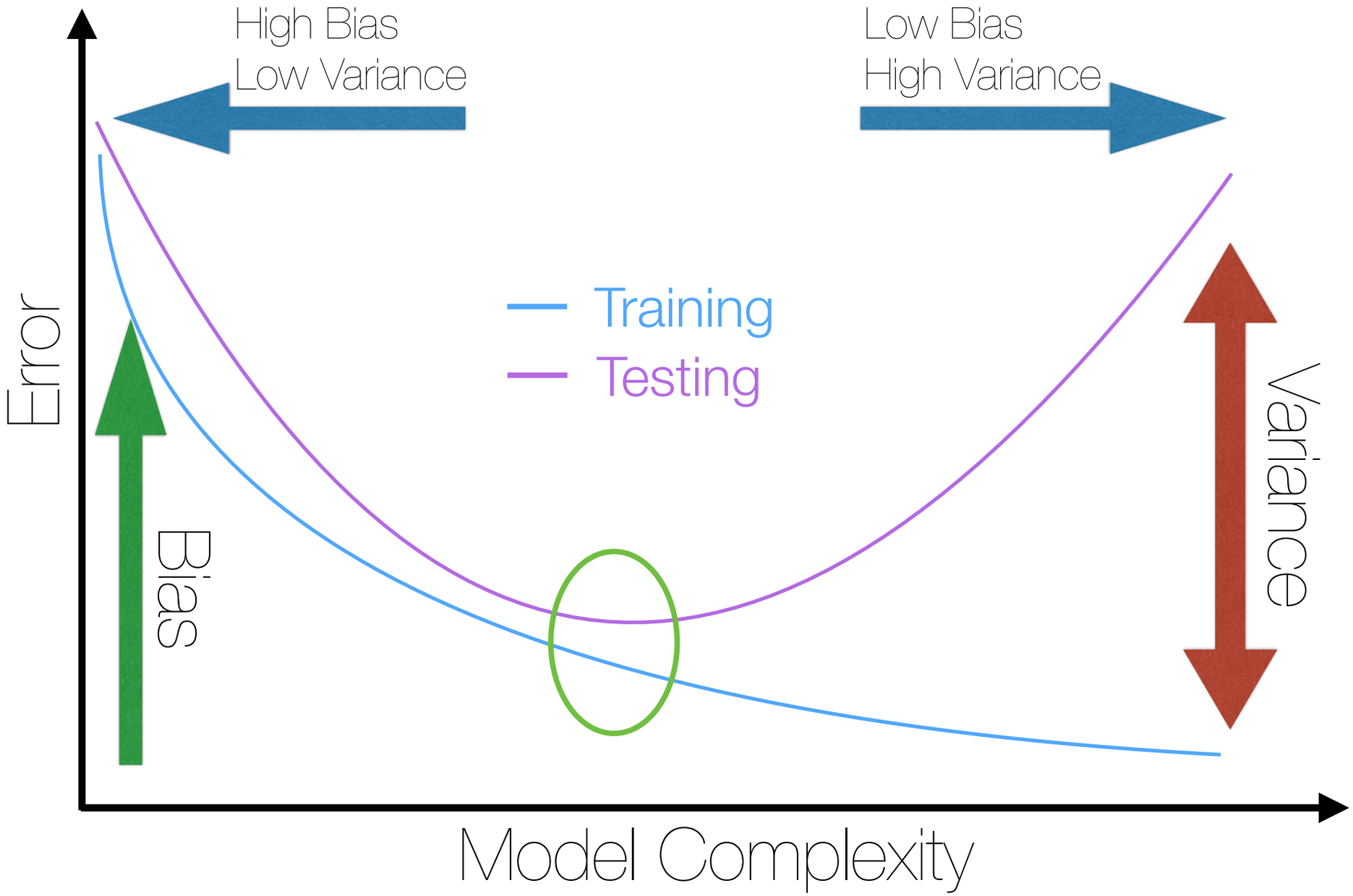




Bias-Variance Tradeoff



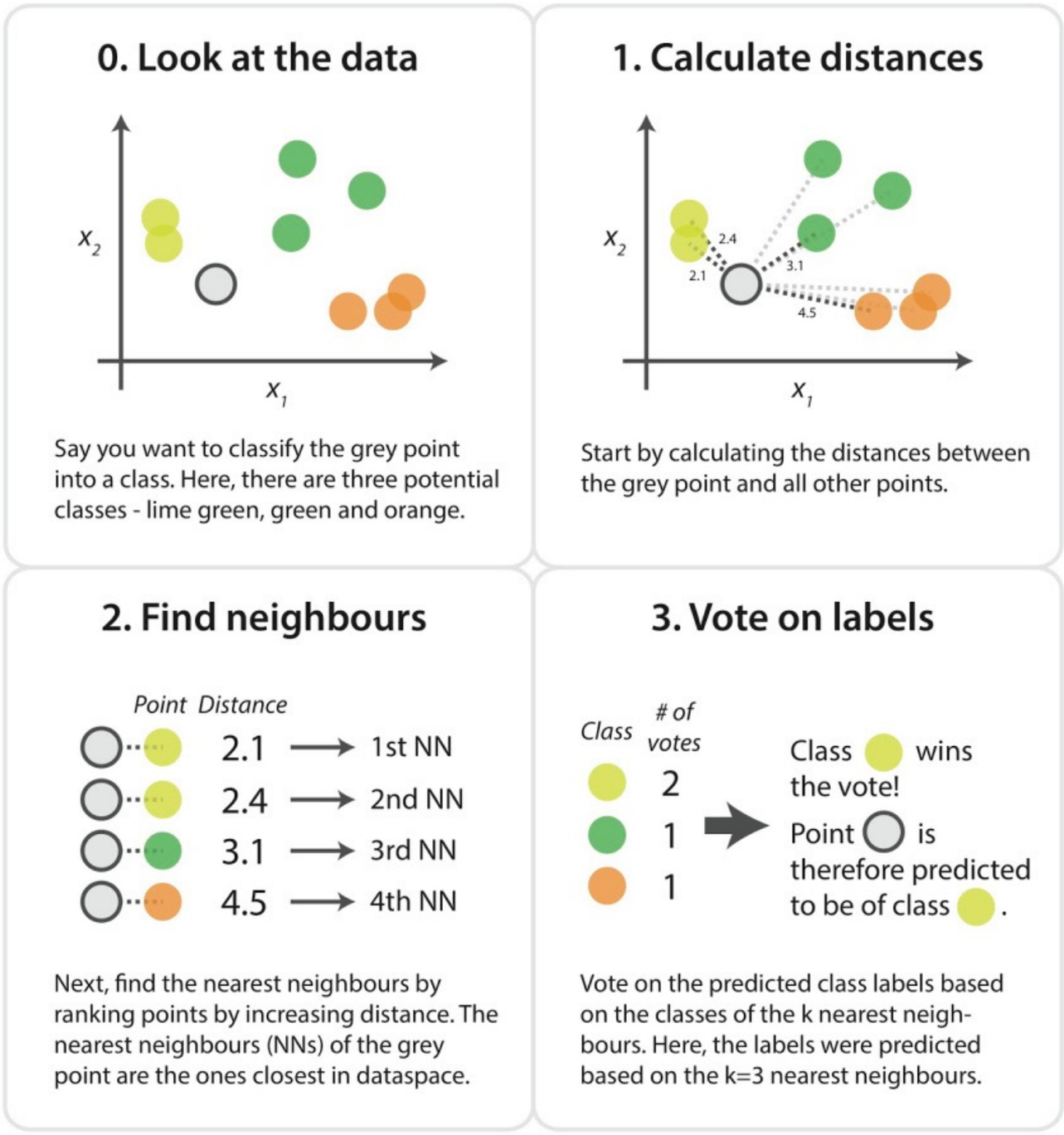
Bias-Variance Tradeoff



K-nearest neighbors

- Perhaps the simplest of supervised learning algorithms
- Effectively memorizes all previously seen data
- Intuitively takes advantage of natural data clustering
- Define that the class of any datapoint is given by the plurality of it's k nearest neighbors
- Many variations using:
    - different distance metrics,
    - weighting procedures,
    - etc…

K-nearest neighbors



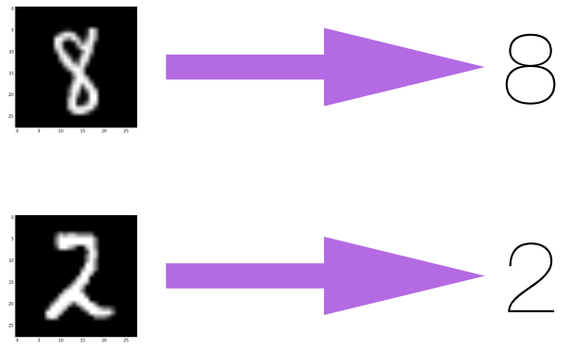
A practical example - MNIST

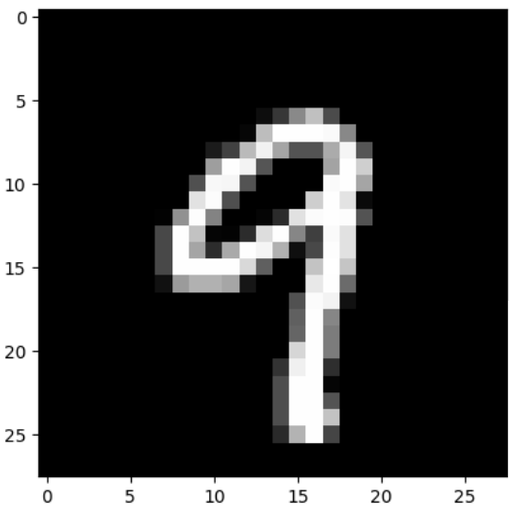
http://yann.lecun.com/exdb/mnist/

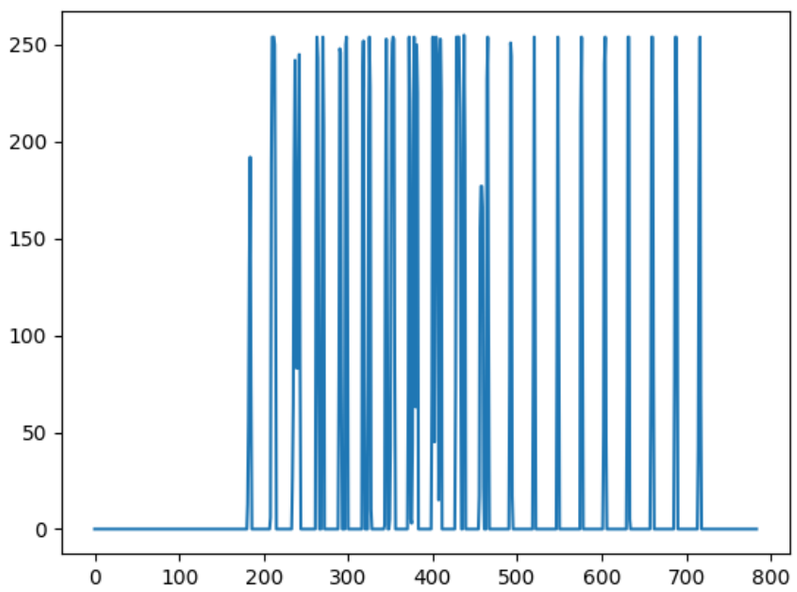
THE MNIST DATABASE

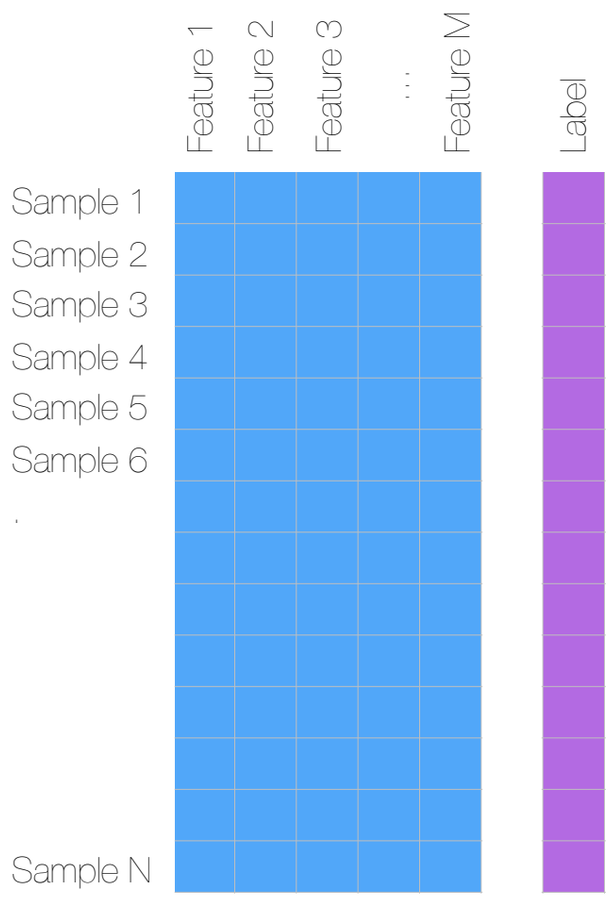
of handwritten digits

Yann LeCun, Courant Institute, NYU
Corinna Cortes, Google Labs, New York
Christopher J.C. Burges, Microsoft Research, Redmond









visualize_digits.py
convert_input.py

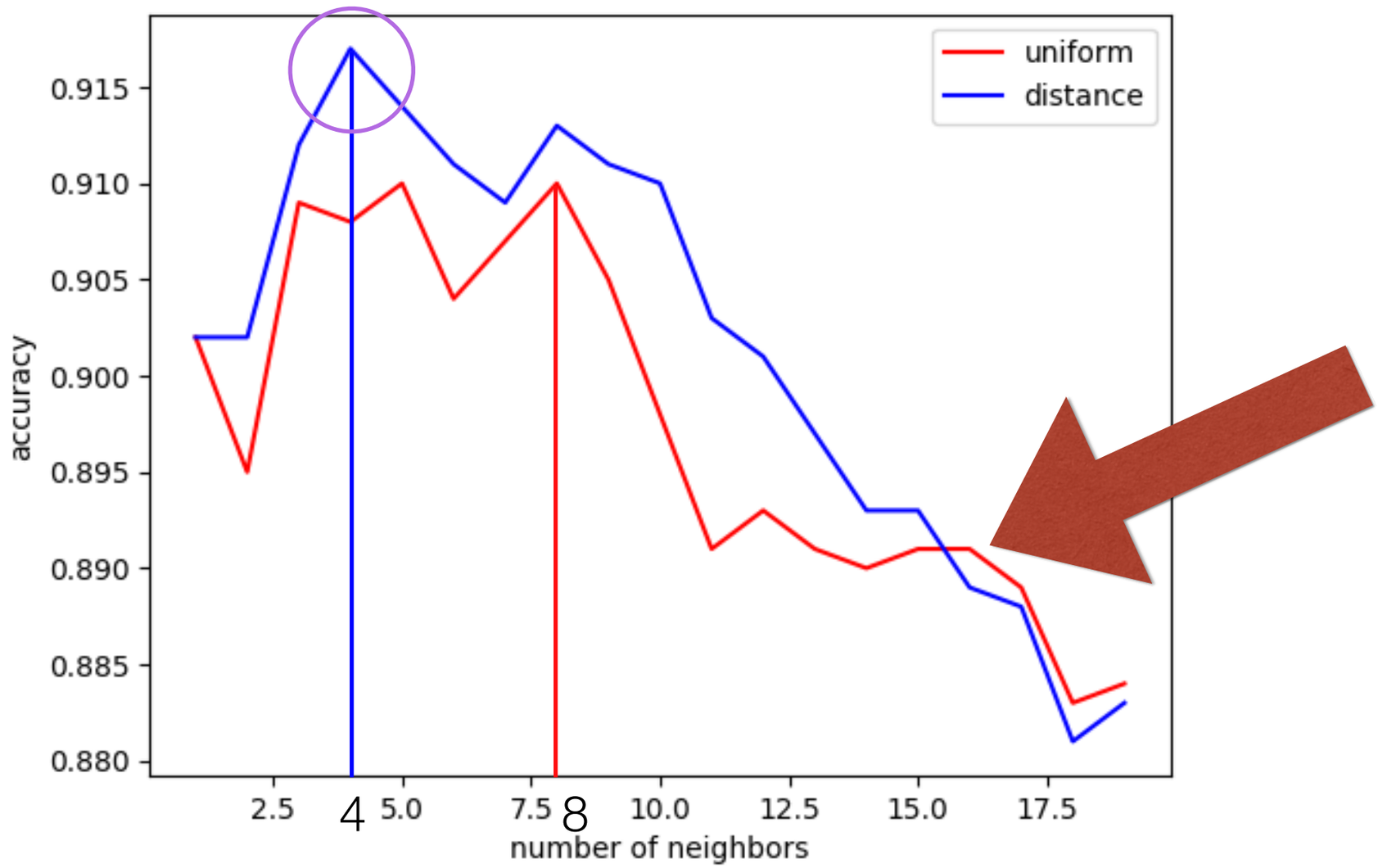
@bgoncalves

K-nearest neighbors

```python
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt

X_train = np.load('input/X_train.npy')
X_test = np.load('input/X_test.npy')
y_train = np.load('input/y_train.npy')
y_test = np.load('input/y_test.npy')

input_layer_size = X_train.shape[1]

X_train /= 255.
X_test /= 255.

def accuracy(y_test, y_pred):
    return np.sum(y_test==y_pred)/len(y_test)

results_distance = []
results_uniform = []

for k in range(1, 21):
    neigh = KNeighborsClassifier(n_neighbors=k, metric='euclidean', weights='uniform')
    neigh.fit(X_train, y_train)

    y_pred = neigh.predict(X_test)

    acc = accuracy(y_test, y_pred)
    results_distance.append([k, acc])

    print(k, acc)
```

knn.py

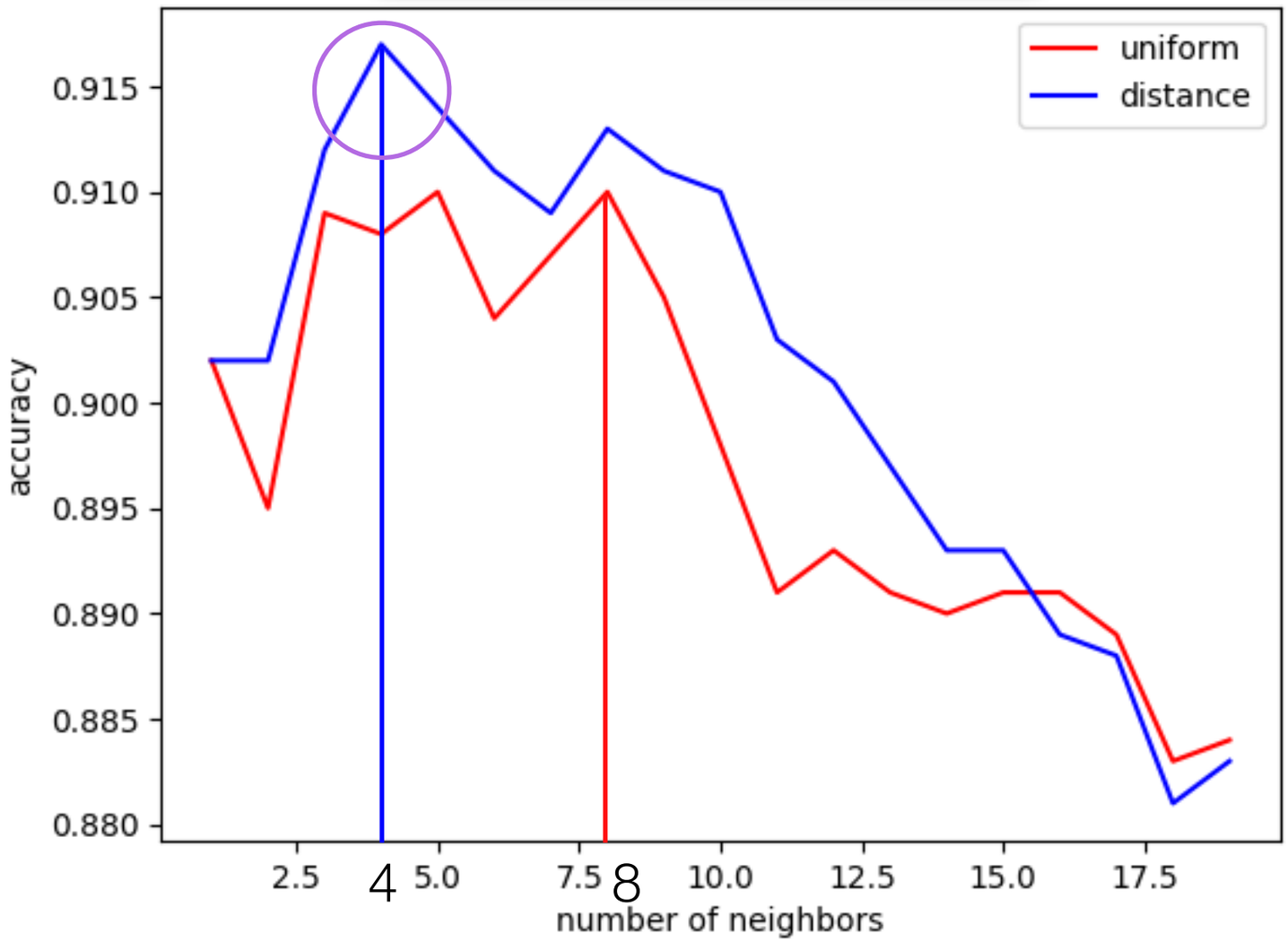
K-nearest neighbors

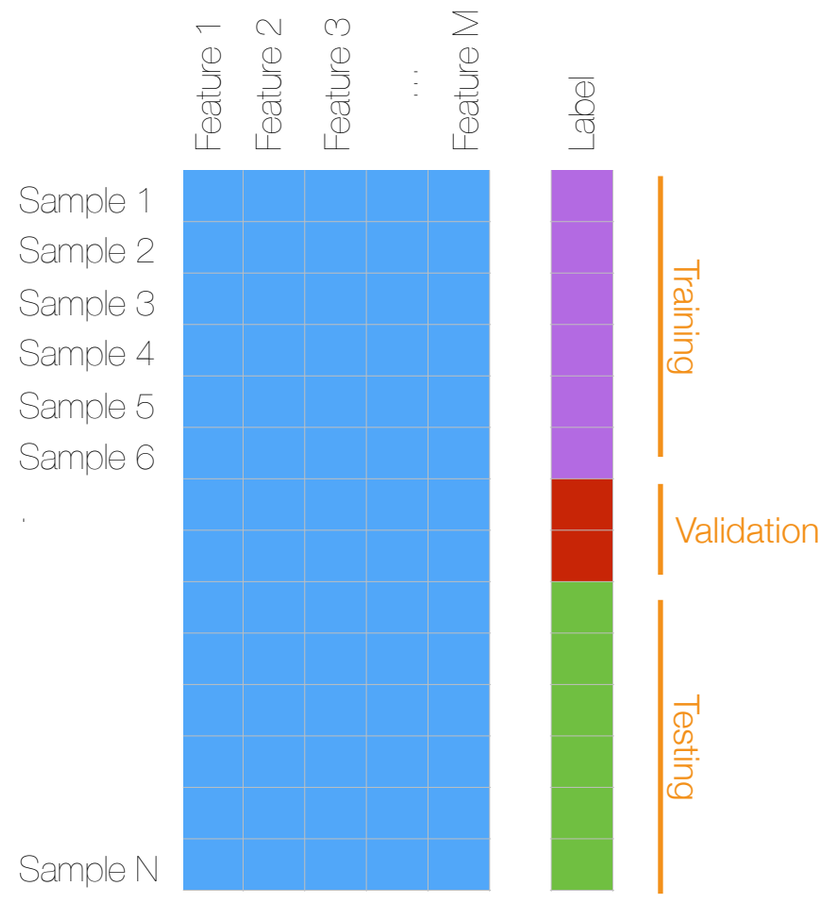


knn.py

K-nearest neighbors

Final Accuracy: 0.708!





knn.py

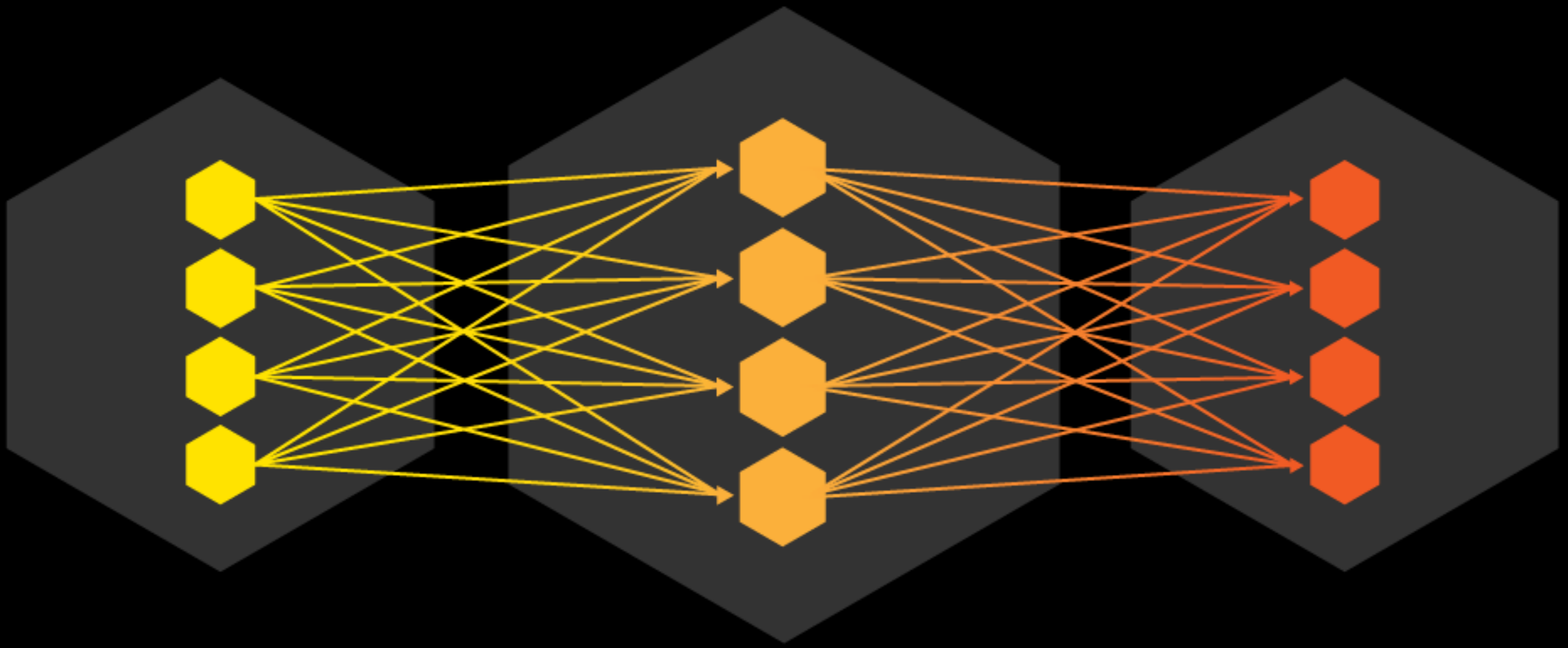
@bgoncalves

K-nea

```python
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt

X_train = np.load('input/X_train.npy')
y_train = np.load('input/y_train.npy')

order = list(range(X_train.shape[0]))
np.random.shuffle(order)

X_train = X_train[order]
y_train = y_train[order]

X_test = np.load('input/X_test.npy')
y_test = np.load('input/y_test.npy')

order = list(range(X_test.shape[0]))
np.random.shuffle(order)

X_test = X_test[order]
y_test = y_test[order]

# Use 1000 points from the training set for the validation set
X_validate = X_train[:1000]
X_train = X_train[1000:]

y_validate = y_train[:1000]
y_train = y_train[1000:]

(…)

neigh = KNeighborsClassifier(n_neighbors=4, metric='euclidean', weights='distance')
neigh.fit(X_train, y_train)

y_pred = neigh.predict(X_validate)
acc = accuracy(y_validate, y_pred)

print(acc)
```

Using a Validation Set
prevents
Information Leakage

knn.py



INPUT TERMS

FEATURES
PREDICTIONS
ATTRIBUTES
PREDICTABLE VARIABLES

MACHINE

ALGORITHMS
TECHNIQUES
MODELS

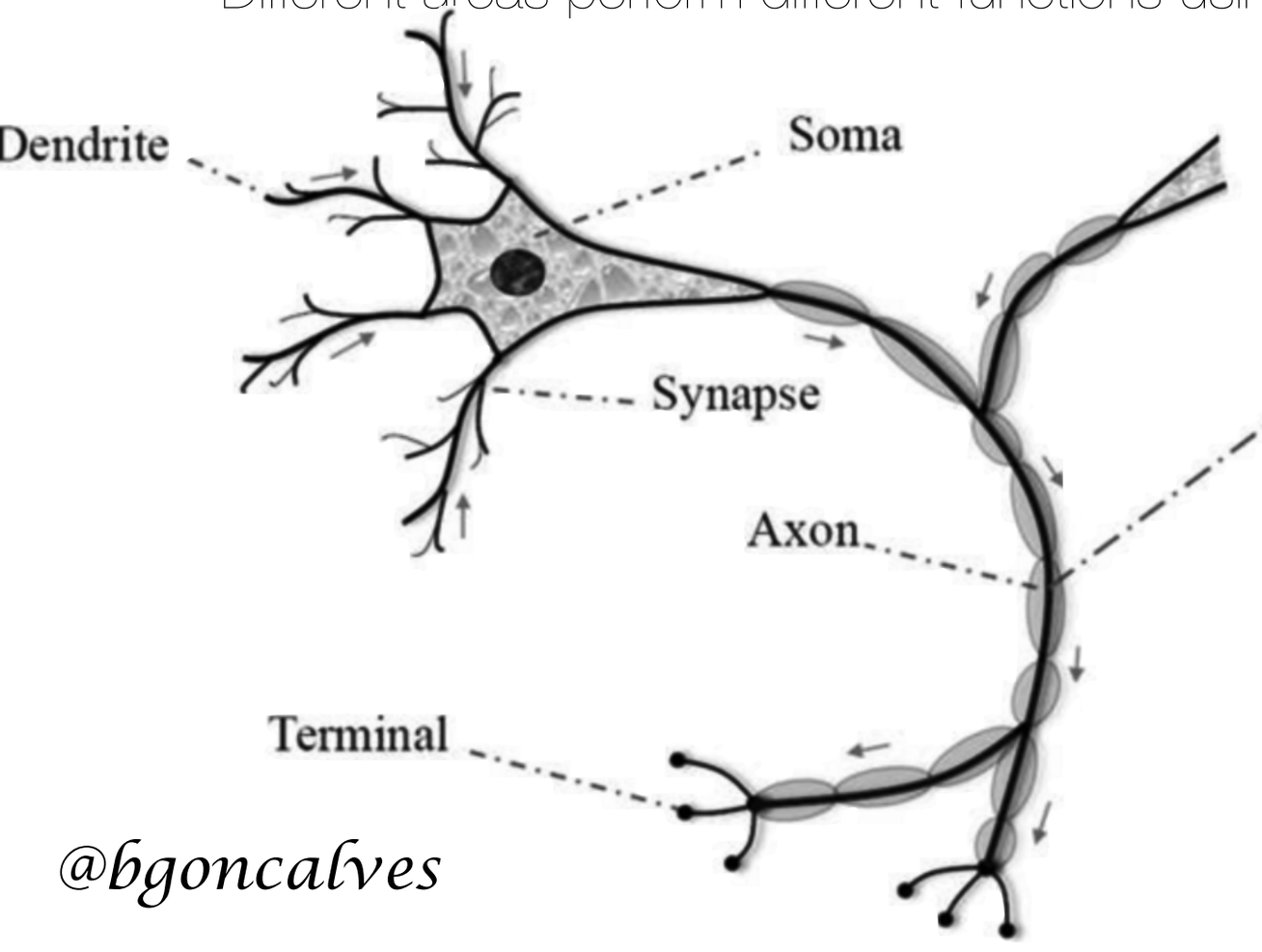
OUTPUT TERMS

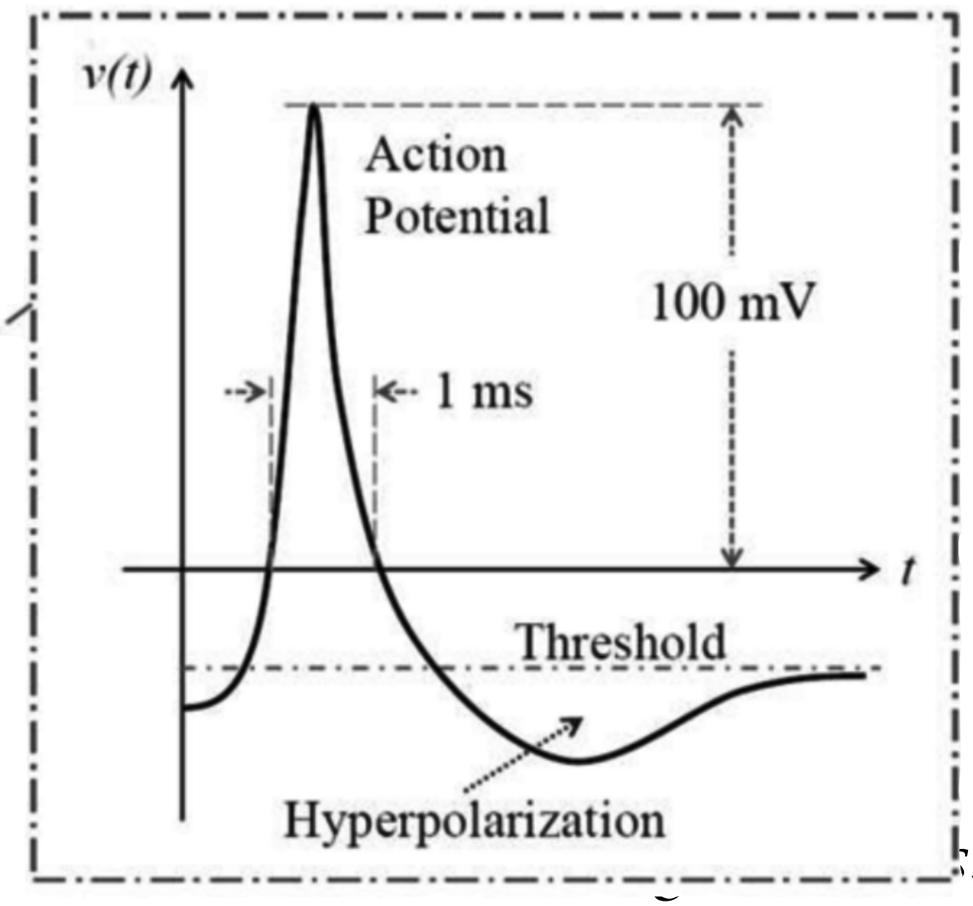
CLASSES
RESPONSES
TARGETS
DEPENDANT VARIABLES

How the Brain "Works" (Cartoon version)

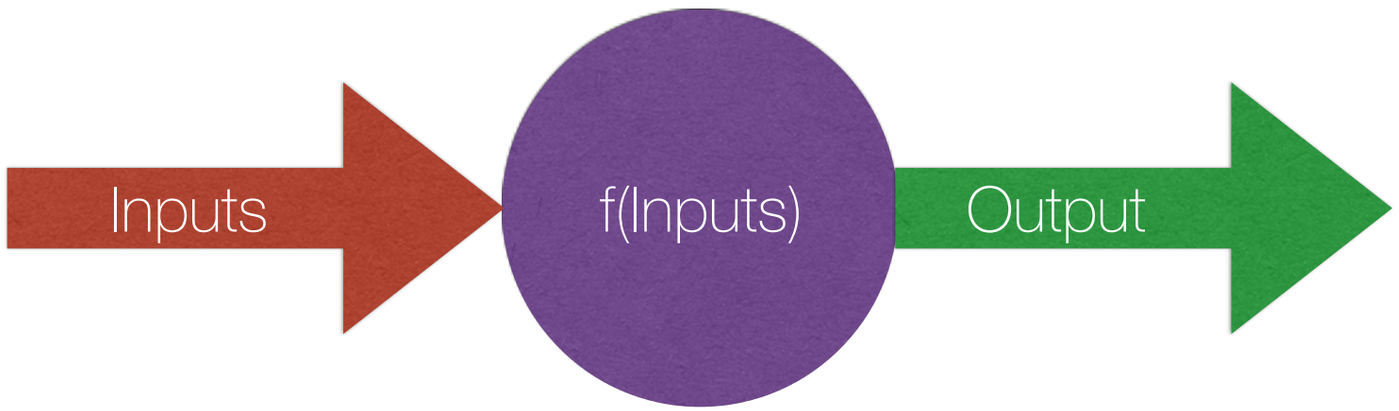
How the Brain "Works" (Cartoon version)

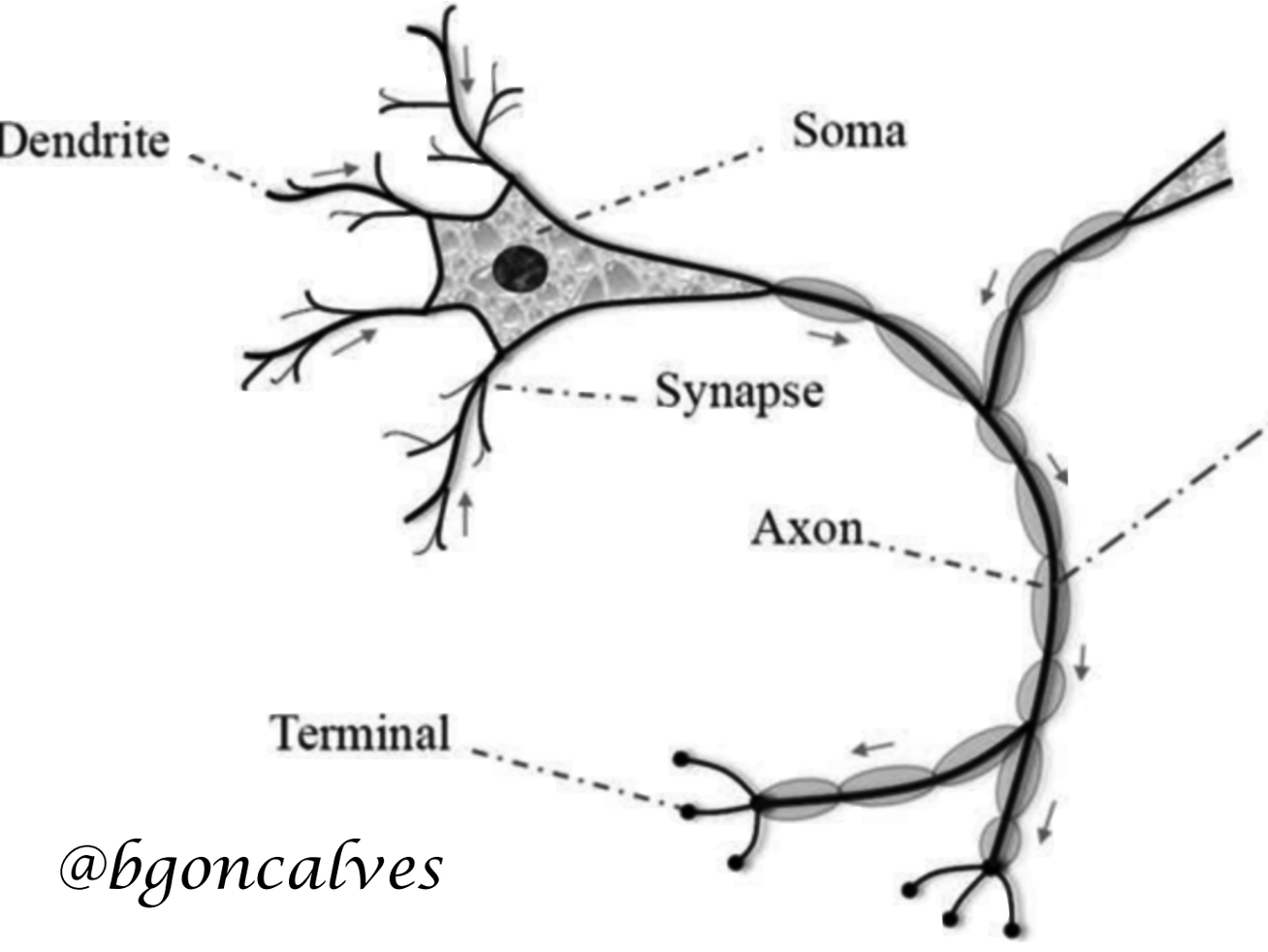
- Each neuron receives input from other neurons
- 10^{11} neurons, each with with 10^{4} weights
- Weights can be positive or negative
- Weights adapt during the learning process
- "neurons that fire together wire together" (Hebb)
- Different areas perform different functions using same structure (Modularity)

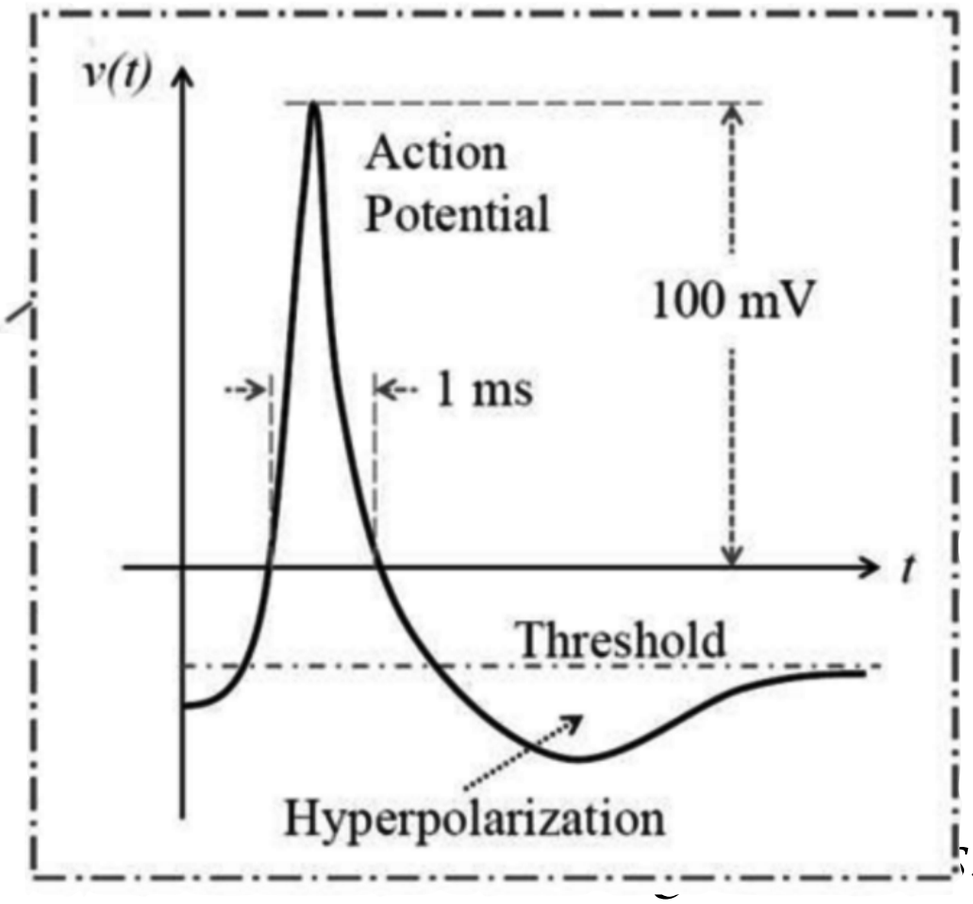




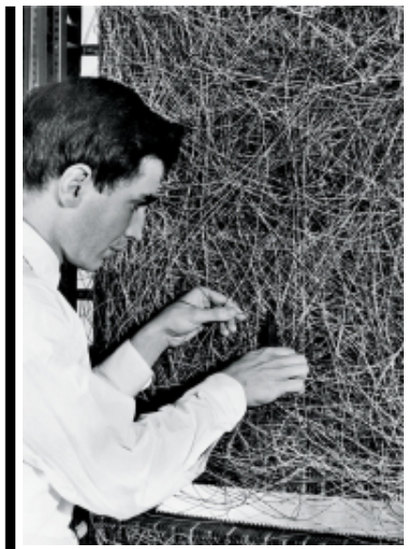
How the Brain "Works" (Cartoon version)







Historical Perspective

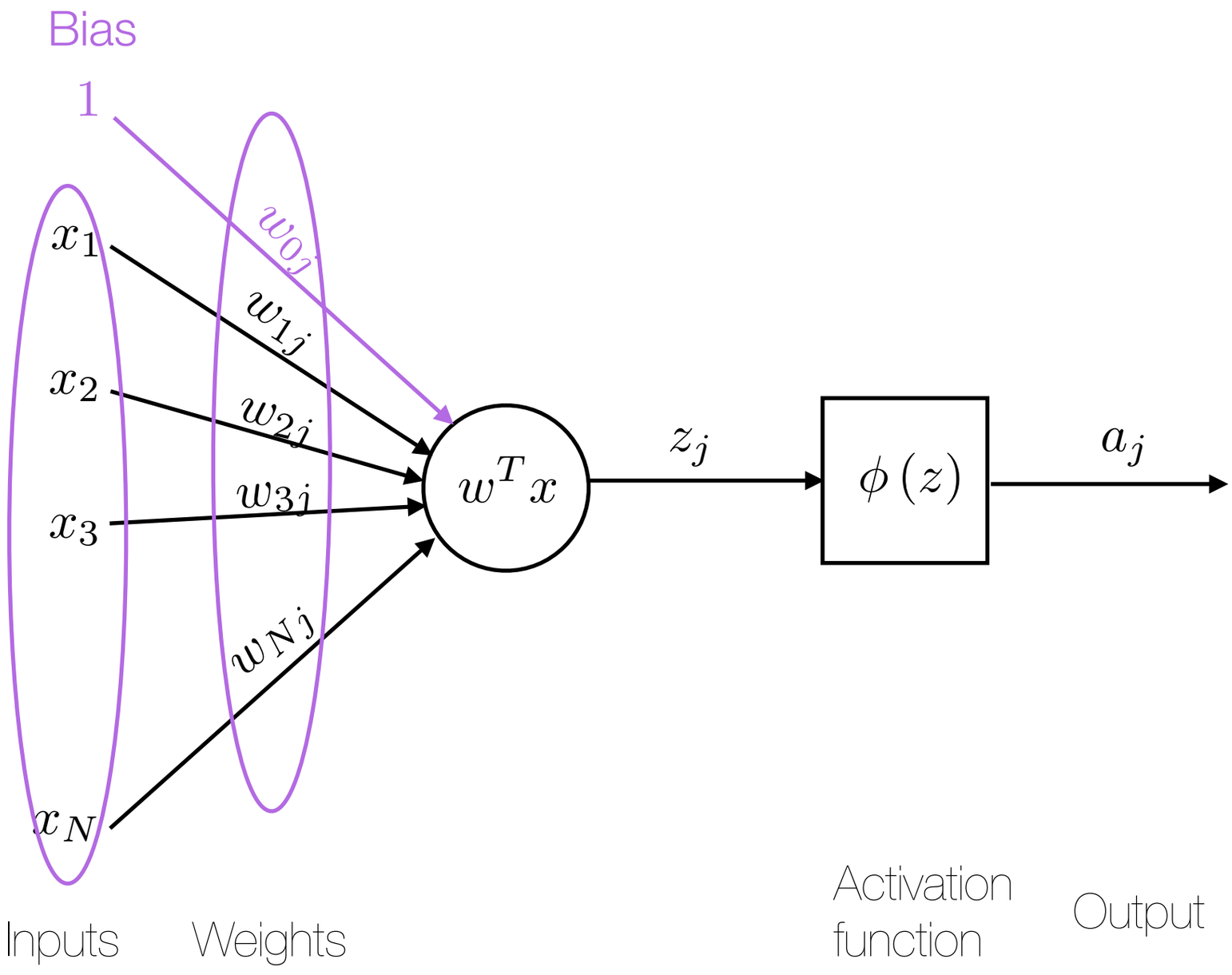


1958

Perceptron

- Popularized by F. Rosenblatt who wrote "Principles of Neurodynamics"
- Still used today
- Simple but limited training procedure
- Single Layer

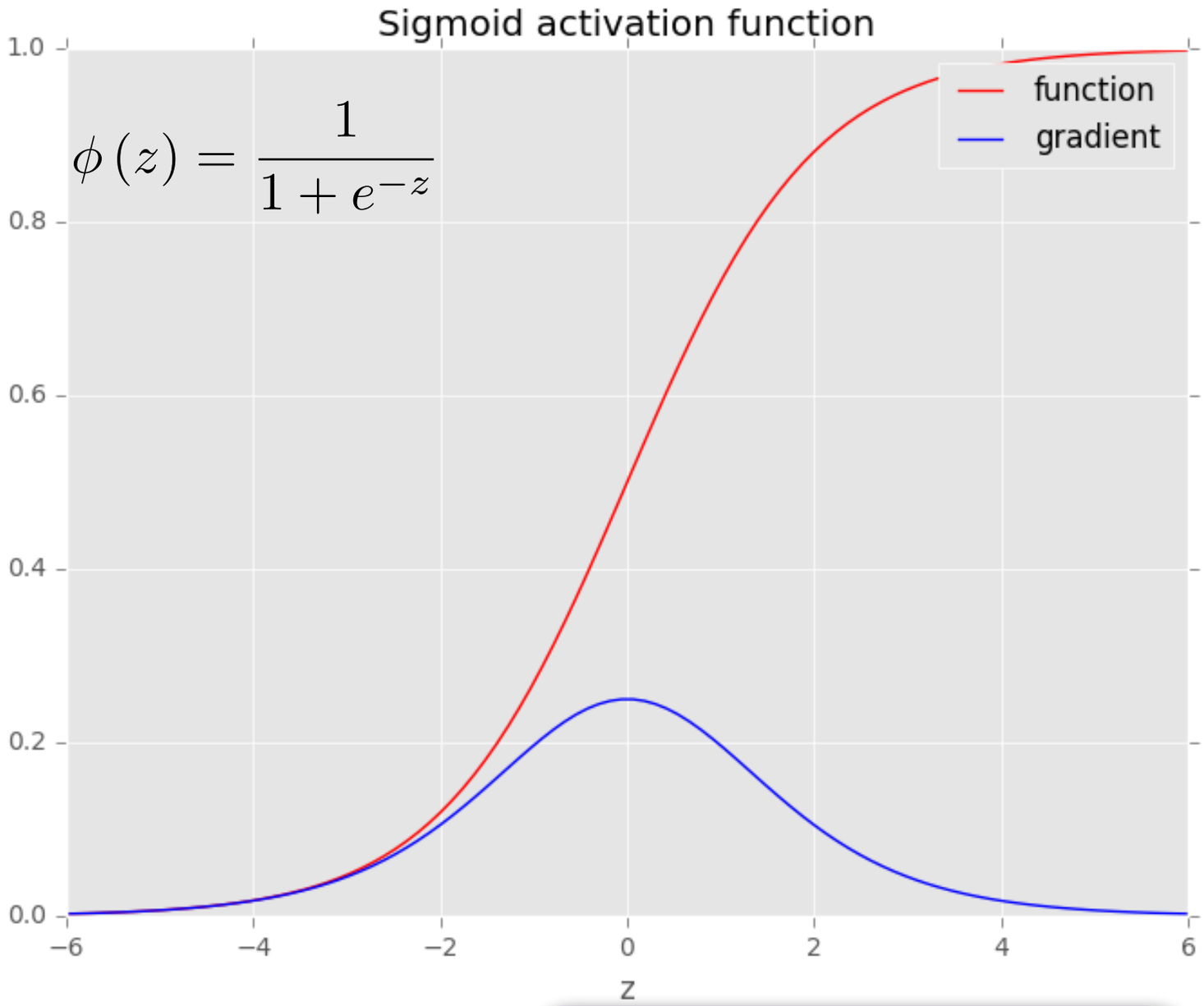
Perceptron


Bias
1
w_{0j}
x_1
w_{1j}
x_2
w_{2j}
x_3
w_{3j}
w_{Nj}
x_N
$w^T x$
z_j
$\phi(z)$
a_j
Inputs
Weights
Activation function
Output

Activation Function - Sigmoid

http://github.com/bmtgoncalves/Neural-Networks

- Non-Linear function
- Differentiable
- non-decreasing
- Compute new sets of features
- Each layer builds up a more abstract representation of the data
- Perhaps the most common

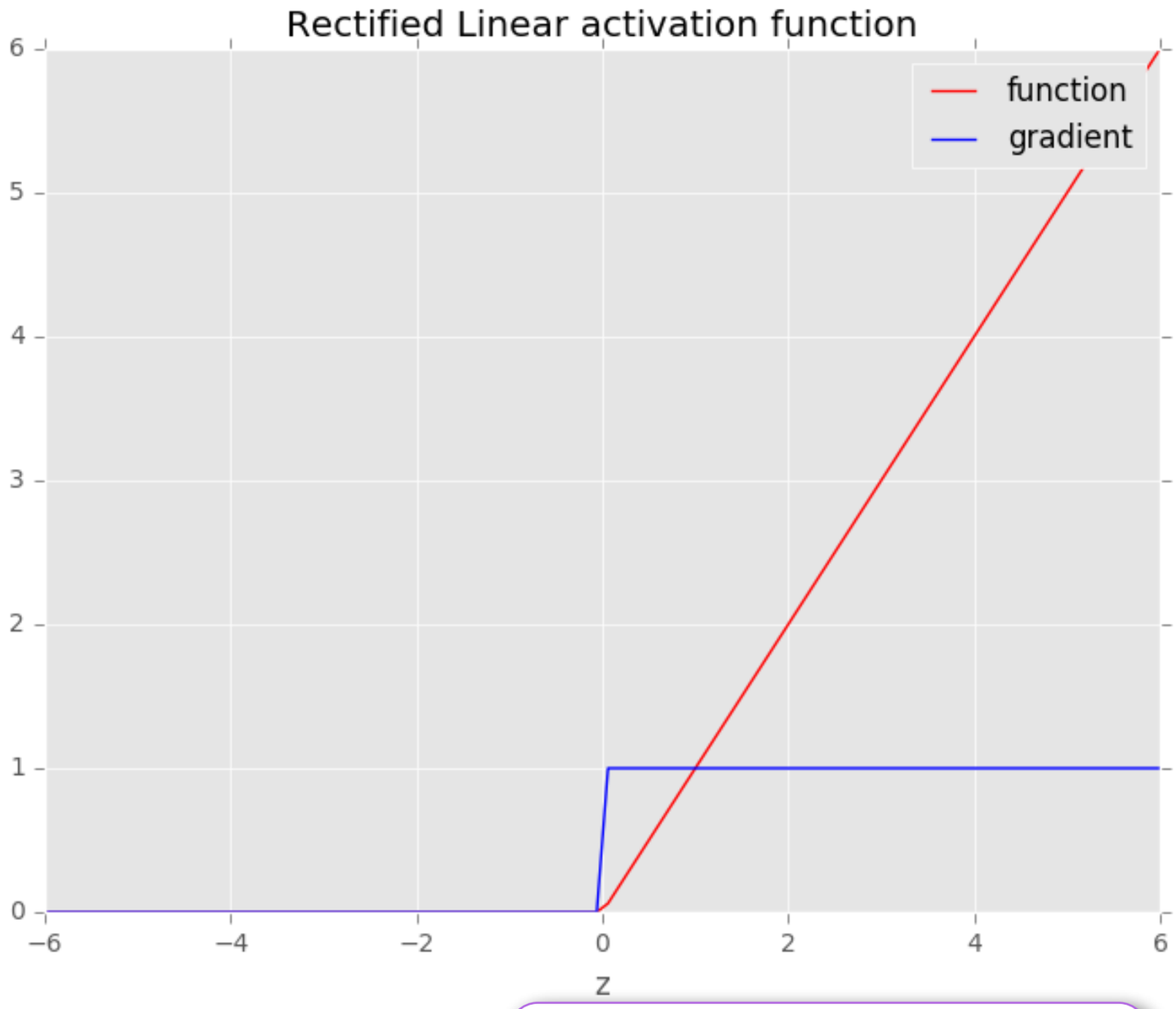


activation.py

Activation Function - ReLu

http://github.com/bmtgoncalves/Neural-Networks

- Non-Linear function
- Differentiable
- non-decreasing
- Compute new sets of features
- Each layer builds up a more abstract representation of the data
- Results in faster learning than with sigmoid

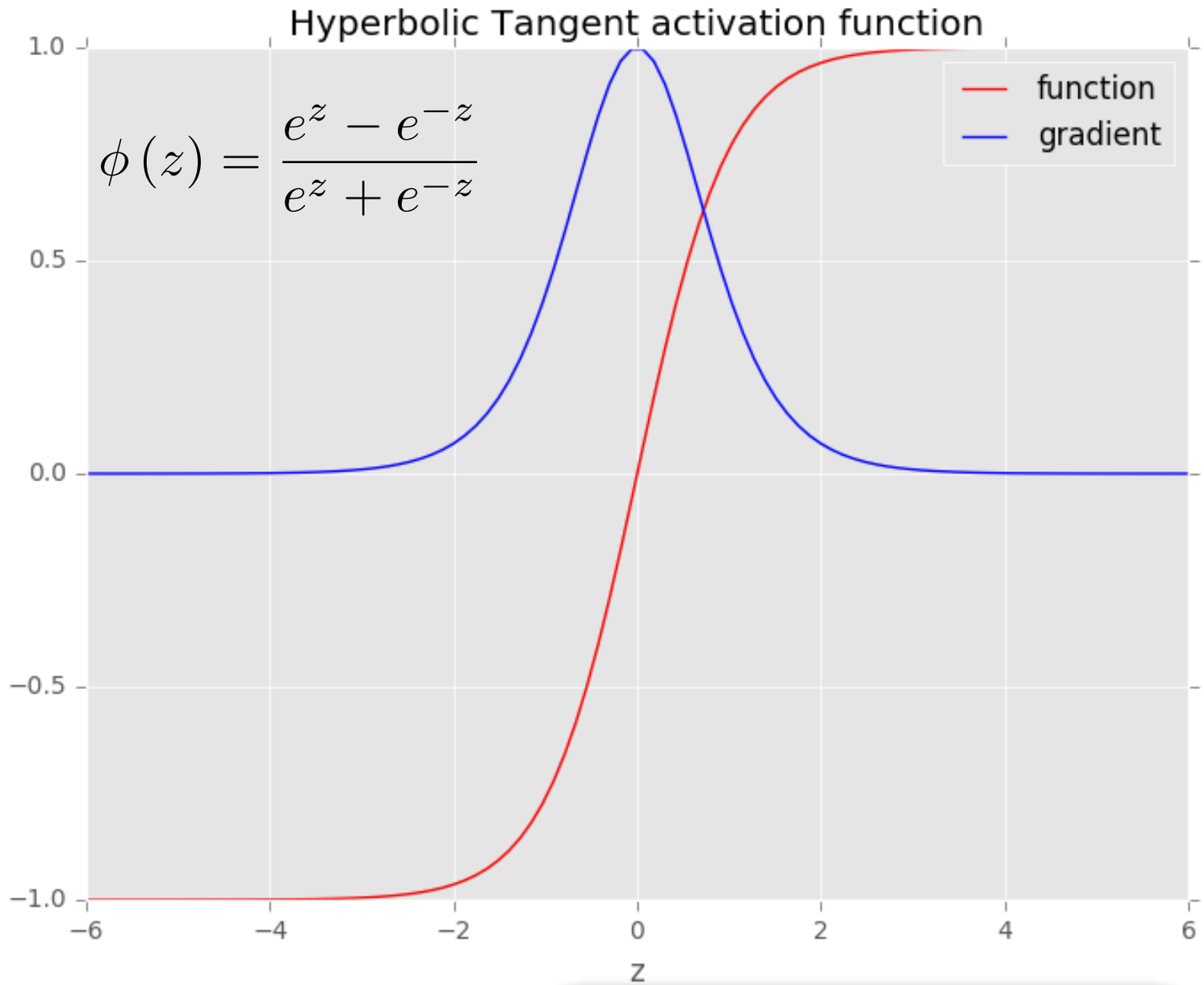


activation.py

Activation Function - tanh

http://github.com/bmtgoncalves/Neural-Networks

- Non-Linear function
- Differentiable
- non-decreasing
- Compute new sets of features
- Each layer builds up a more abstract representation of the data



activation.py

Activation Function

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


def linear(z):
    return z


def binary(z):
    return np.where(z > 0, 1, 0)


def relu(z):
    return np.where(z > 0, z, 0)


def sigmoid(z):
    return 1./(1+np.exp(-z))


def tanh(z):
    return np.tanh(z)

z = np.linspace(-6, 6, 100)

plt.style.use('ggplot')

plt.plot(z, linear(z), 'r-')
plt.xlabel('z')
plt.title('Linear activation function')
plt.savefig('linear.png')
plt.close()
```

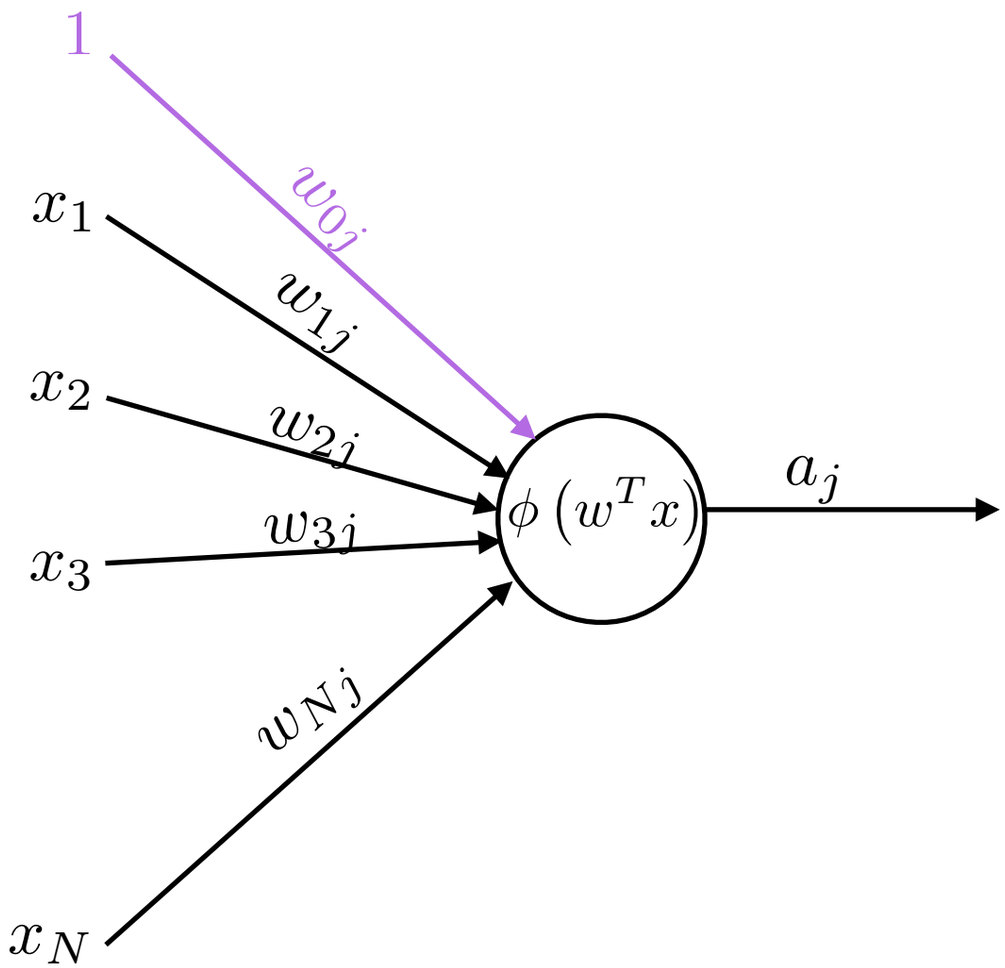
activation.py

Perceptron - Forward Propagation

```python
import numpy as np

def forward(Theta, X, active):
    N = X.shape[0]

    # Add the bias column
    X_ = np.concatenate((np.ones((N, 1)), X), 1)

    # Multiply by the weights
    z = np.dot(X_, Theta.T)

    # Apply the activation function
    a = active(z)

    return a

if __name__ == "__main__":
    Theta1 = np.load('input/Theta1.npy')
    X = np.load('input/X_train.npy')[:10]

    from activation import sigmoid

    active_value = forward(Theta1, X, sigmoid)
```

forward.py

Perceptron

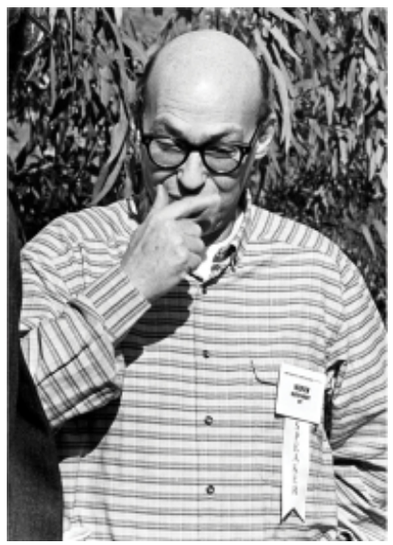


Training Procedure:

- If correct, do nothing
- If output incorrectly outputs 0, add input to weight vector
- if output incorrectly outputs 1, subtract input to weight vector

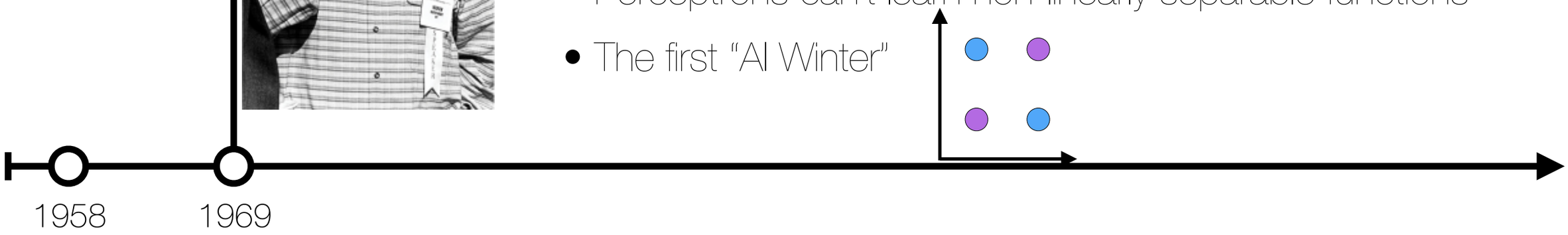
- Guaranteed to converge, if a correct set of weights exists
- Given enough features, perceptrons can learn almost anything
- Specific Features used limit what is possible to learn

Historical Perspective



Marvin Minsky

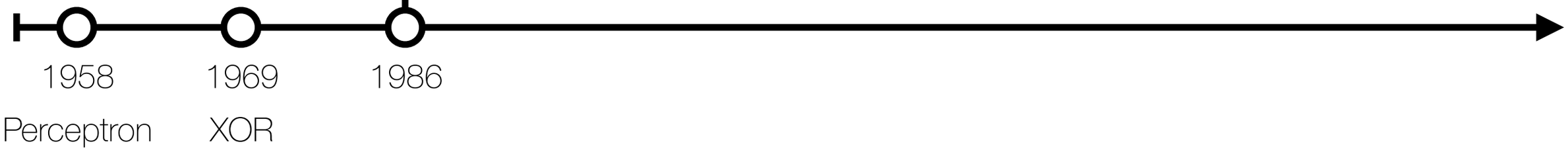
- Co-authors "Perceptrons" with Seymour Papert
- XOR Problem
- Perceptrons can't learn non-linearly separable functions
- The first "AI Winter"



Historical Perspective

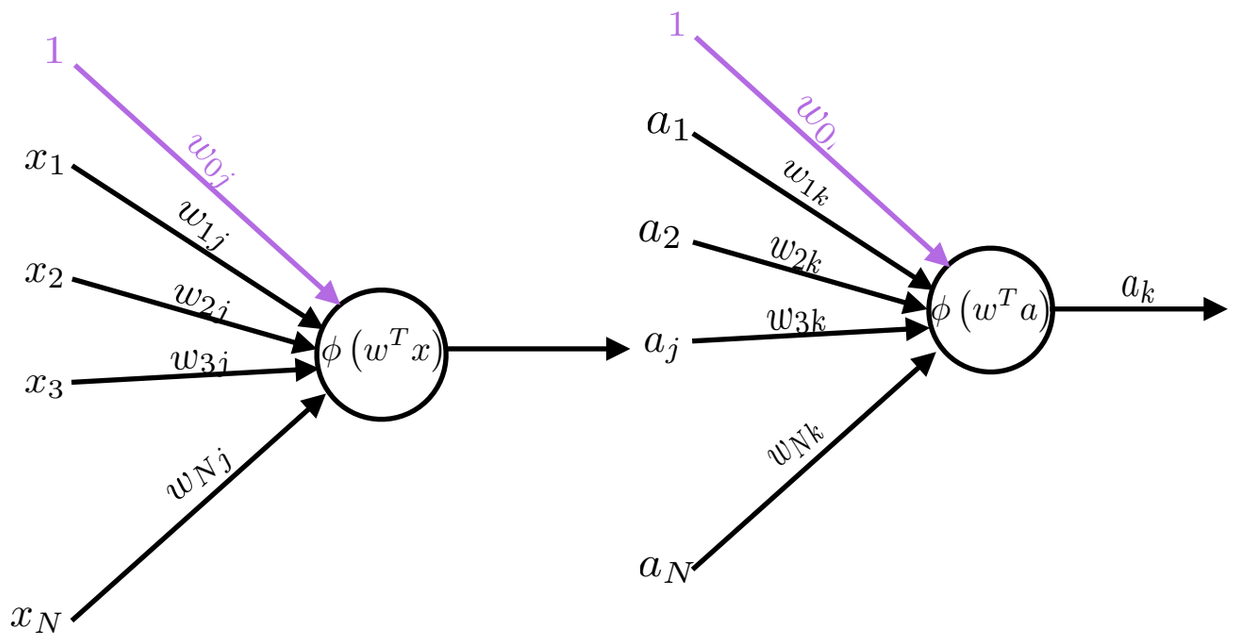
Geoff Hinton

- Discovers "Backpropagation"
- "Multi-layer Perceptron"
- Expensive computation requiring lots of data
- Impractical



Forward Propagation

- The output of a perceptron is determined by a sequence of steps:
    - obtain the inputs
    - multiply the inputs by the respective weights
    - calculate output using the activation function
- To create a multi-layer perceptron, you can simply use the output of one layer as the input to the next one.



- But how can we propagate back the errors and update the weights?

Backward Propagation of Errors (BackProp)

- BackProp operates in two phases:
    - Forward propagate the inputs and calculate the deltas
    - Update the weights

- The error at the output is the squared difference between predicted output and the observed one:

$$E = (t - y)^2$$

- Where t is the real output and y is the predicted one.

- For inner layers there is no "real output"!

Chain-rule

- From the forward propagation described above, we know that the output y_j of a neuron is:

$$y_j = \phi\left(w^T x\right)$$

- But how can we calculate how to modify the weights w_{ij}?

- We take the derivative of the error with respect to the weights!

$$\frac{\partial E}{\partial w_{ij}}$$

- Using the chain rule:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial w_{ij}}$$

- And finally we can update each weight in the previous layer:

$$w_{ij} \longleftarrow w_{ij} - \alpha\frac{\partial E}{\partial w_i j}$$

- where α is the learning rate

Loss Functions

- For learning to occur, we must quantify how far off we are from the desired output. There are two common ways of doing this:

  - Quadratic error function:

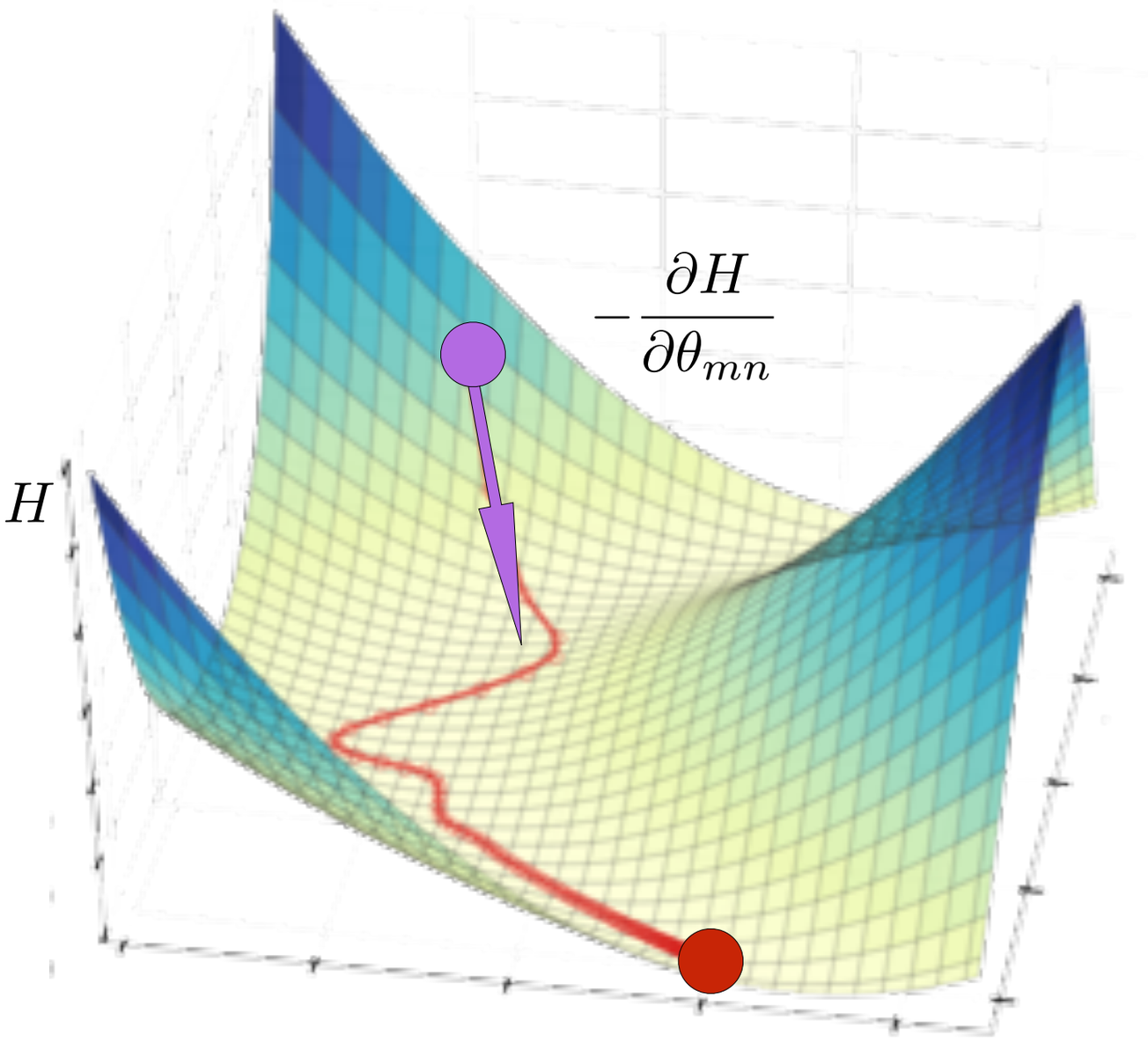
$$E = \frac{1}{N} \sum_n |y_n - a_n|^2$$

  - Cross Entropy

$$J = -\frac{1}{N} \sum_n \left[y_n^T \log a_n + (1 - y_n)^T \log (1 - a_n) \right]$$

The Cross Entropy is complementary to sigmoid activation in the output layer and improves its stability.

Gradient Descent



- Find the gradient for each training batch
- Take a step **downhill** along the direction of the gradient

$$\theta_{mn} \leftarrow \theta_{mn} - \alpha \frac{\partial H}{\partial \theta_{mn}}$$

- where α is the step size.
- Repeat until "convergence".

Historical Perspective

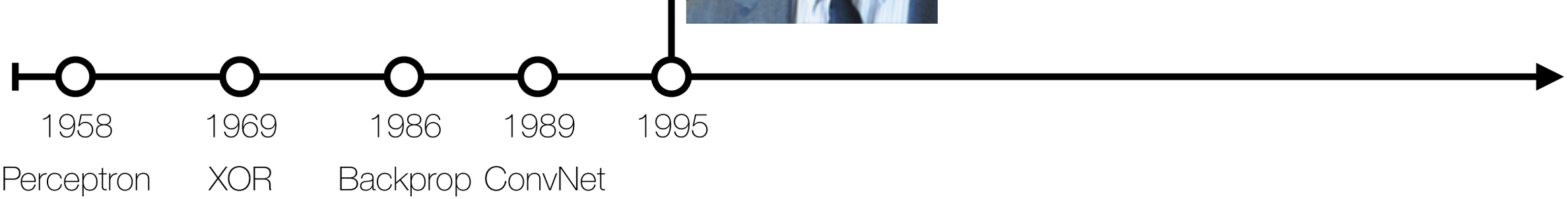
Yann LeCun

- Starts working on Convolutional Neural Networks
- (Eventually) develops the first practical application of image recognition

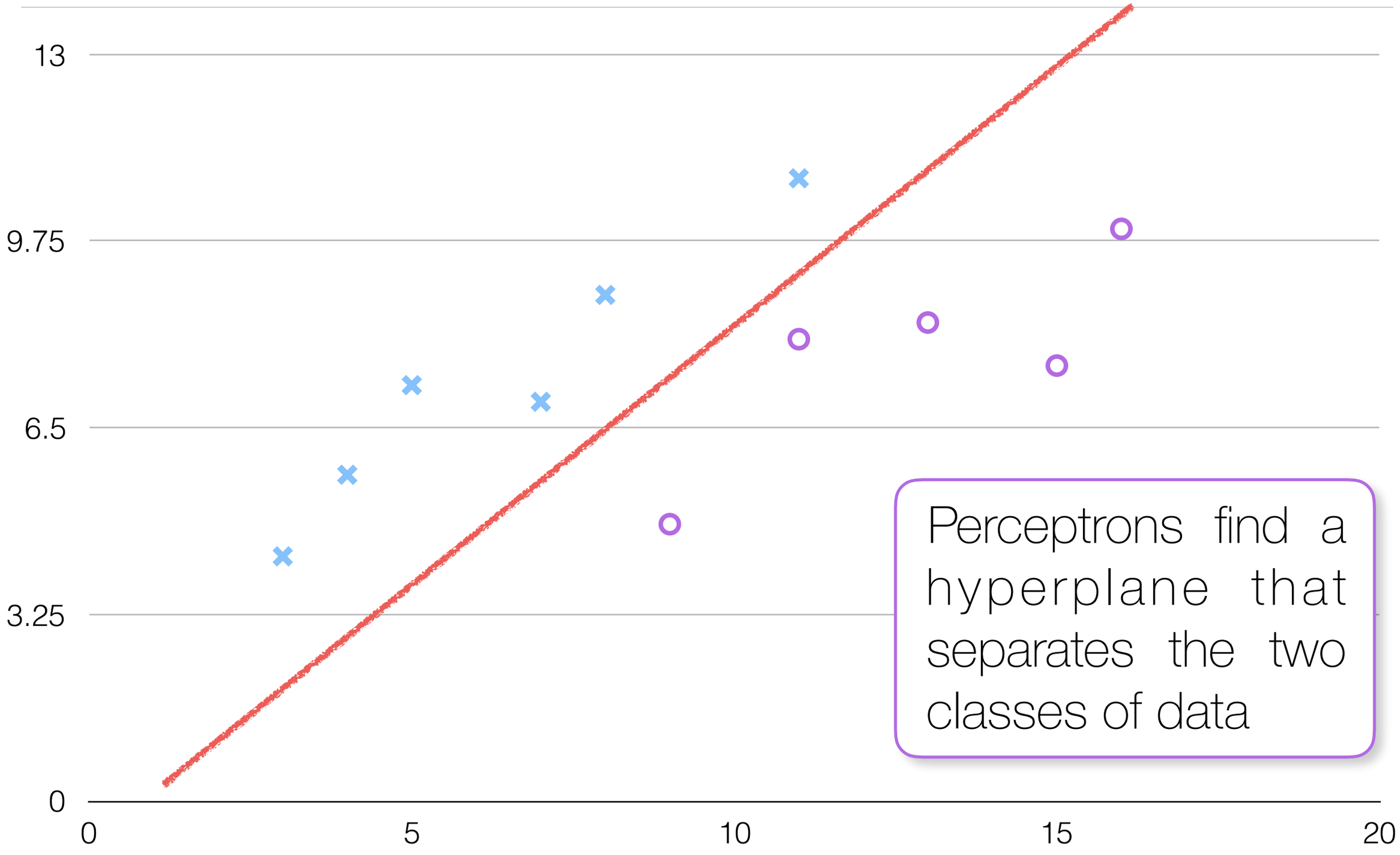
Historical Perspective

Vladimir Vapnik

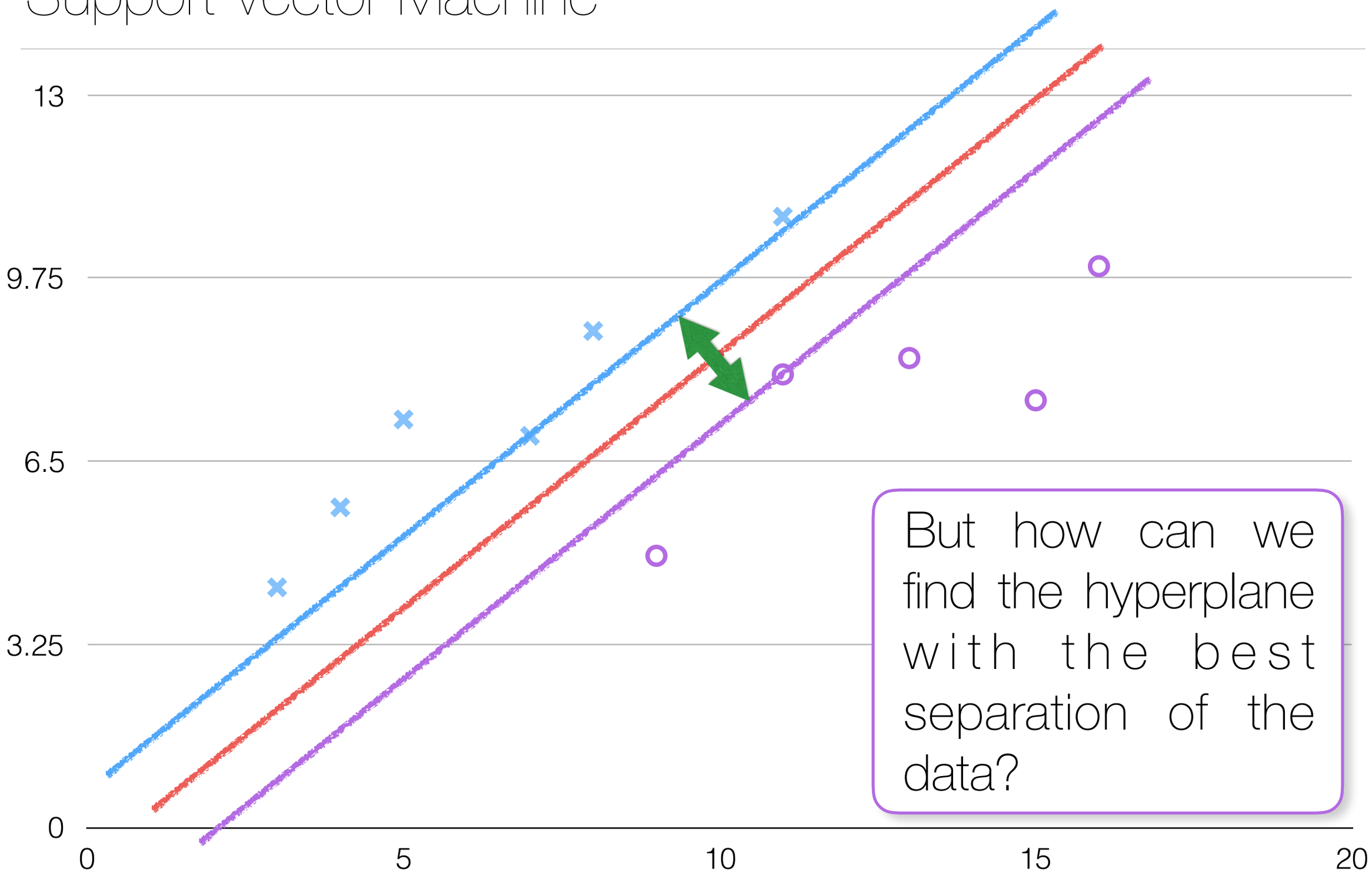
- Support Vector Machines
- A clever variation on Perceptrons
- Kernel Trick



Support Vector Machine



Support Vector Machine



But how can we find the hyperplane with the best separation of the data?

Support Vector Machines

- Decision plane has the form:

$$w^T x = 0$$

- We want $w^T x \geq b$ for points in the "positive" class and $w^T x \leq -b$ for points in the negative class. Where the "margin" $2b$ is as large as possible.

- Normalize such that $b = 1$ and solve the optimization problem:

$$\min_w ||w||^2$$

subject to:

$$y_i \left(w^T x \right) > 1$$

- The margin is:

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

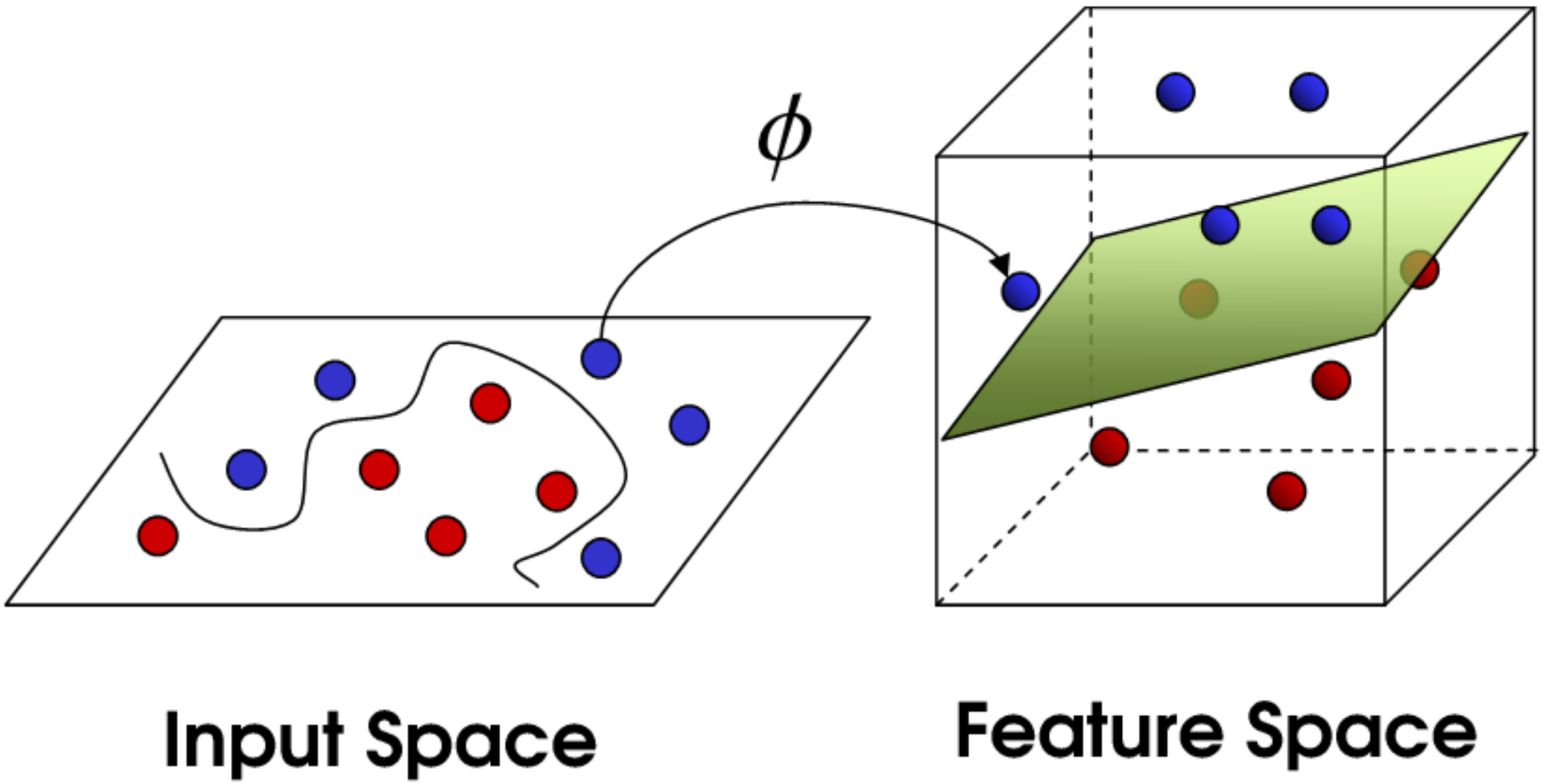
Support Vector Machines

```python
import numpy as np
from sklearn import svm
import matplotlib.pyplot as plt

X_train = np.load('input/X_train.npy')
y_train = np.load('input/y_train.npy')

order = list(range(X_train.shape[0]))
np.random.shuffle(order)
X_train = X_train[order]
y_train = y_train[order]

X_test = np.load('input/X_test.npy')
y_test = np.load('input/y_test.npy')

order = list(range(X_test.shape[0]))
np.random.shuffle(order)
X_test = X_test[order]
y_test = y_test[order]

input_layer_size = X_train.shape[1]
X_train /= 255.
X_test /= 255.

def accuracy(y_test, y_pred):
    return np.sum(y_test==y_pred)/len(y_test)

clf = svm.SVC()
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

print(accuracy(y_test, y_pred))
```

SVM.py

Kernel "trick"

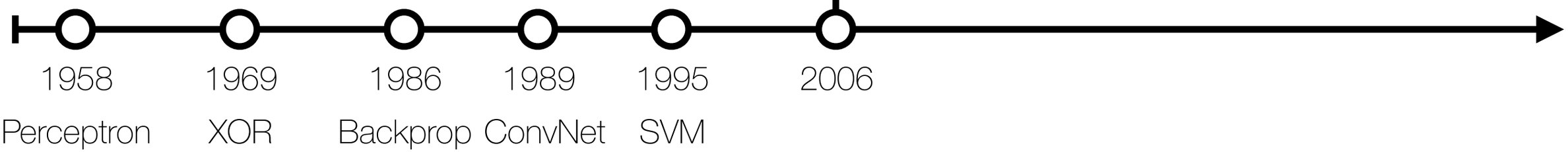
- SVM procedure uses only the dot products of vectors and never the vectors themselves.
- We can redefine the dot product in any way we wish.
- In effect we are mapping from a non-linear input space to a linear feature space



Historical Perspective

Geoff Hinton

- Deep Neural Networks
- Combines backprop with faster machines and larger datasets



A practical example - MNIST

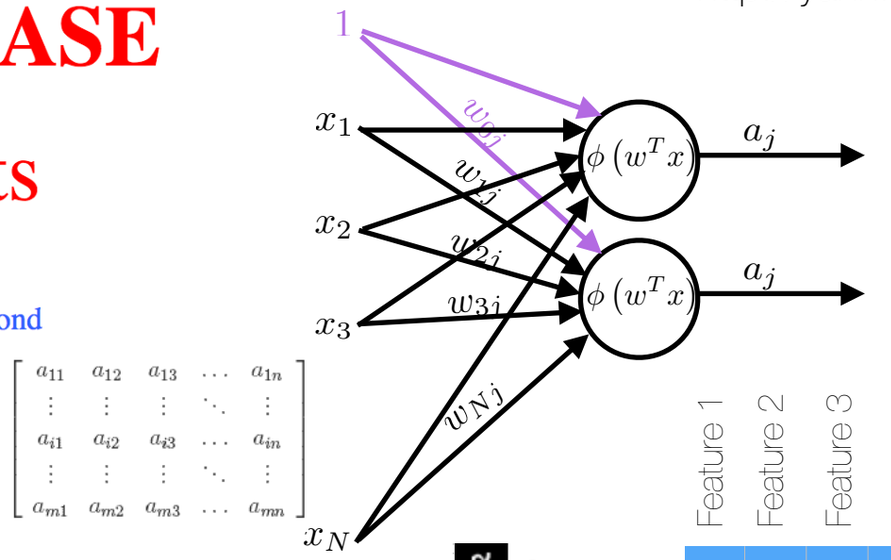
http://github.com/bmtgoncalves/Neural-Networks

http://yann.lecun.com/exdb/mnist/

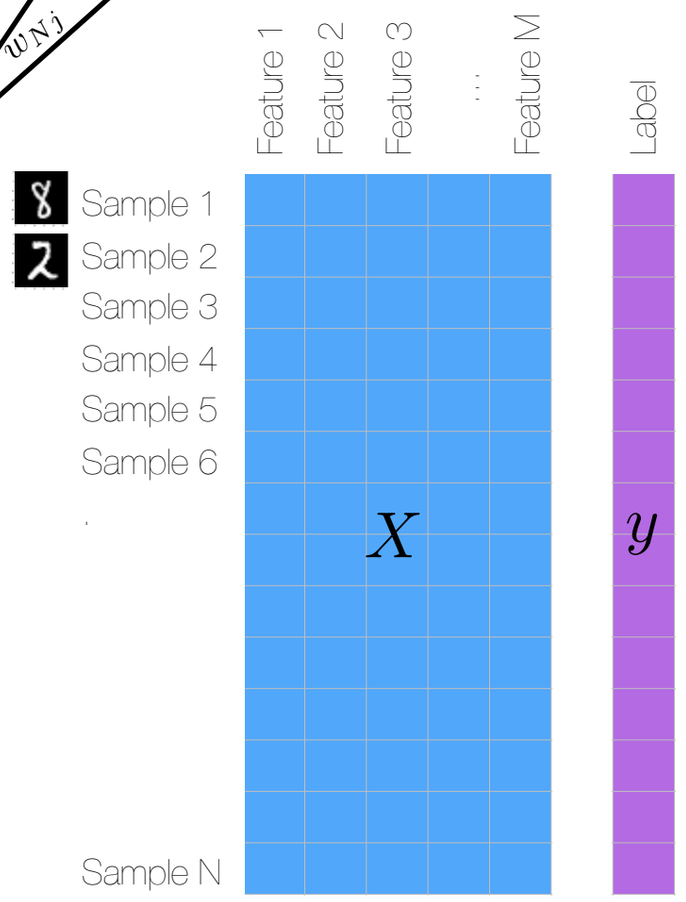
THE MNIST DATABASE

of handwritten digits

Yann LeCun, Courant Institute, NYU
Corinna Cortes, Google Labs, New York
Christopher J.C. Burges, Microsoft Research, Redmond

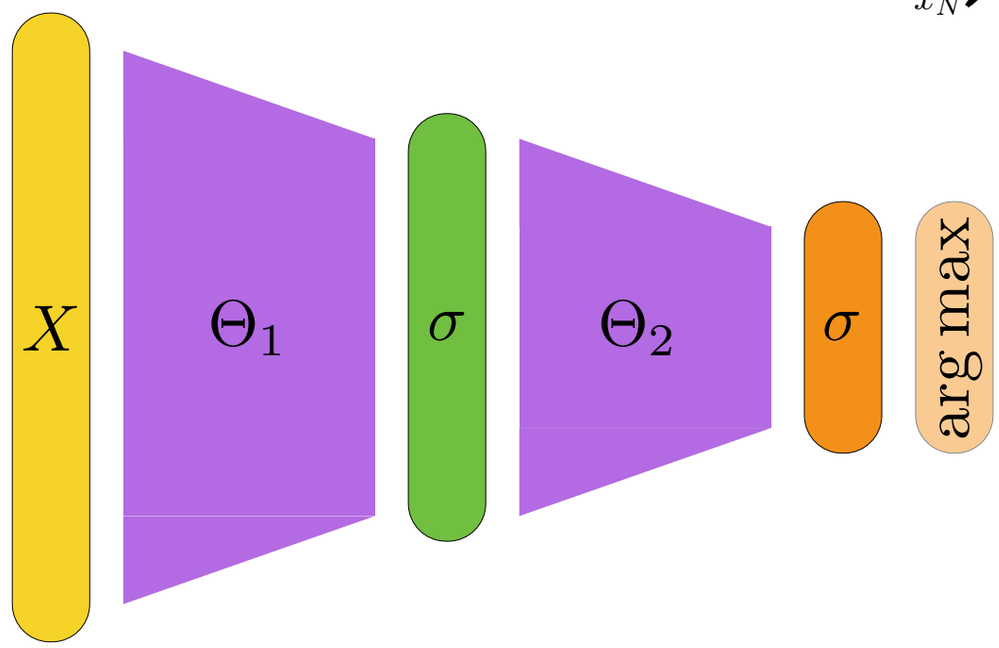


$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & a_{i3} & \dots & a_{in} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{bmatrix}$$



3 layers:

- 1 input layer
- 1 hidden layer
- 1 output layer

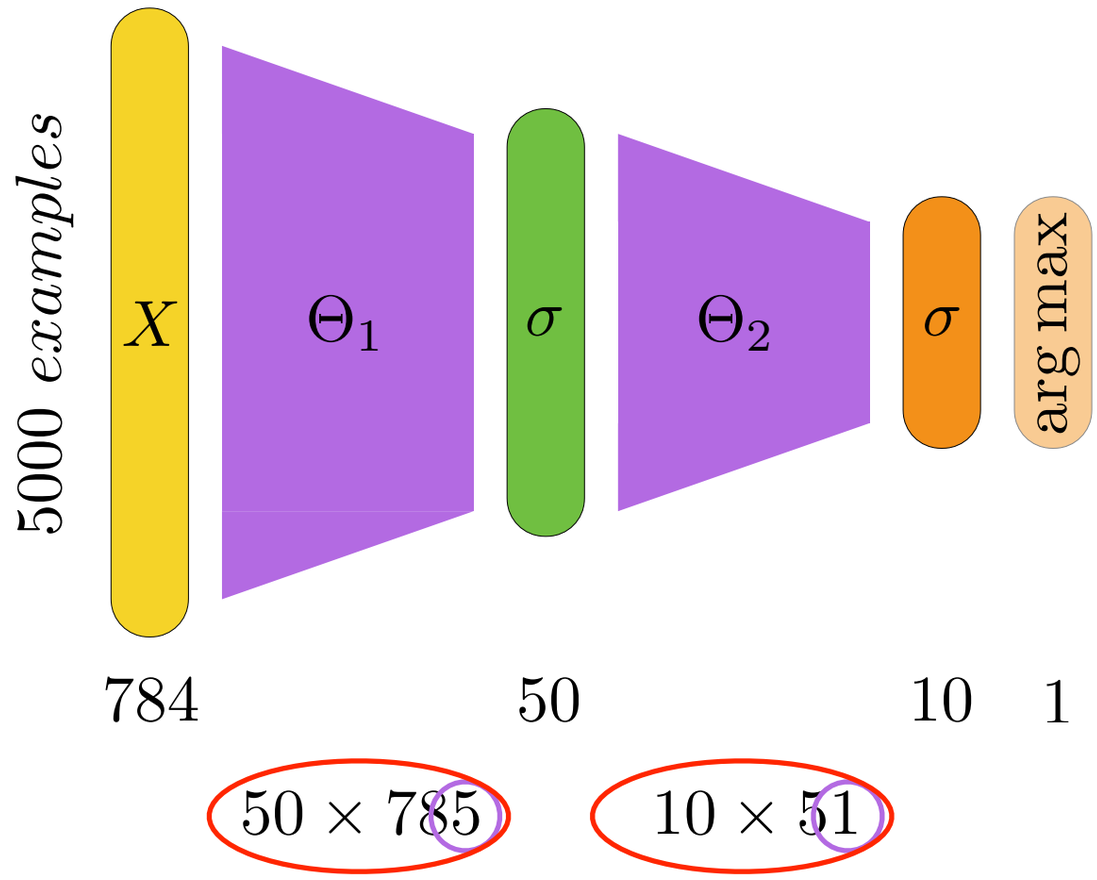


visualize_digits.py
convert_input.py

@bgoncalves

A practical example - MNIST

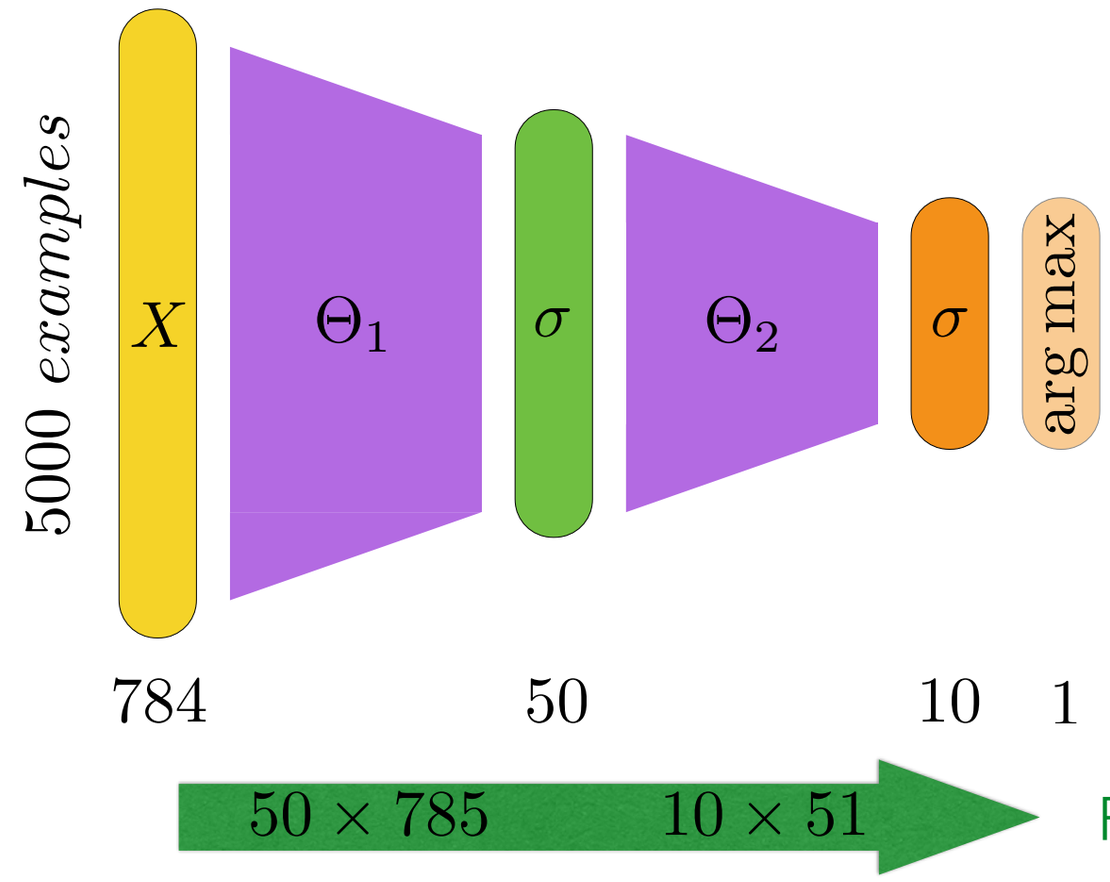
http://github.com/bmtgoncalves/Neural-Networks



nn.py
forward.py

A practical example - MNIST

http://github.com/bmtgoncalves/Neural-Networks



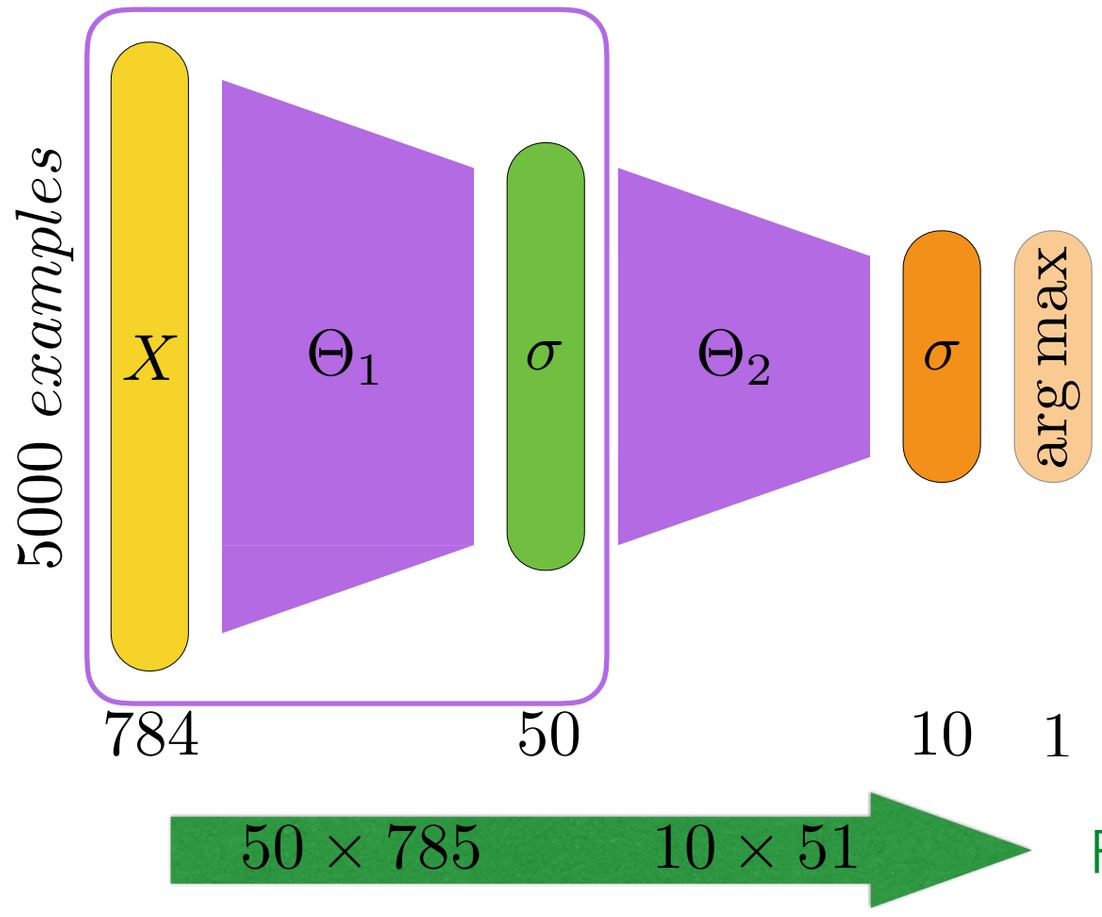

```python
def forward(Theta, X, active):
    N = X.shape[0]

    # Add the bias column
    X_ = np.concatenate((np.ones((N, 1)), X), 1)

    # Multiply by the weights
    z = np.dot(X_, Theta.T)

    # Apply the activation function
    a = active(z)

    return a
```

```python
def predict(Theta1, Theta2, X):
    h1 = forward(Theta1, X, sigmoid)
    h2 = forward(Theta2, h1, sigmoid)

    return np.argmax(h2, 1)
```

nn.py
forward.py

A practical example - MNIST

http://github.com/bmtgoncalves/Neural-Networks



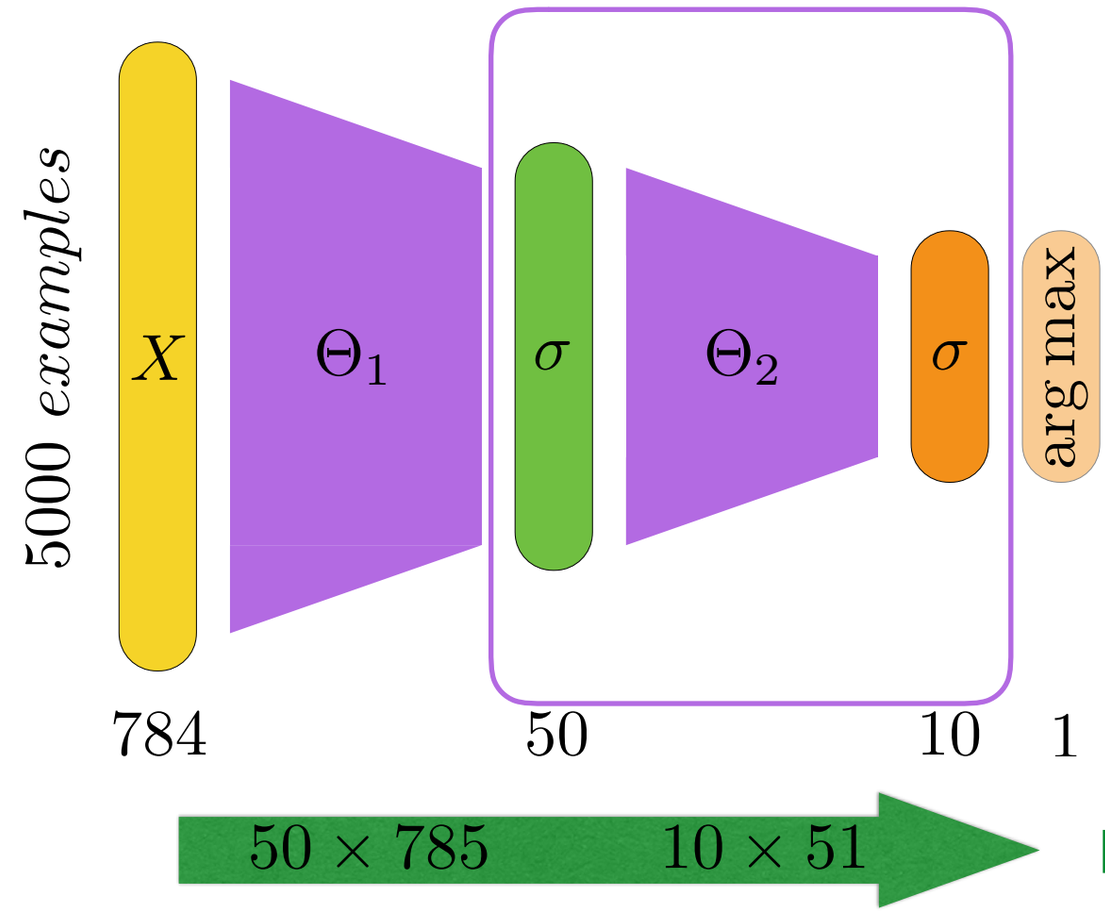

```python
def forward(Theta, X, active):
    N = X.shape[0]

    # Add the bias column
    X_ = np.concatenate((np.ones((N, 1)), X), 1)

    # Multiply by the weights
    z = np.dot(X_, Theta.T)

    # Apply the activation function
    a = active(z)

    return a
```

```python
def predict(Theta1, Theta2, X):
    h1 = forward(Theta1, X, sigmoid)
    h2 = forward(Theta2, h1, sigmoid)

    return np.argmax(h2, 1)
```

nn.py
forward.py

A practical example - MNIST

http://github.com/bmtgoncalves/Neural-Networks



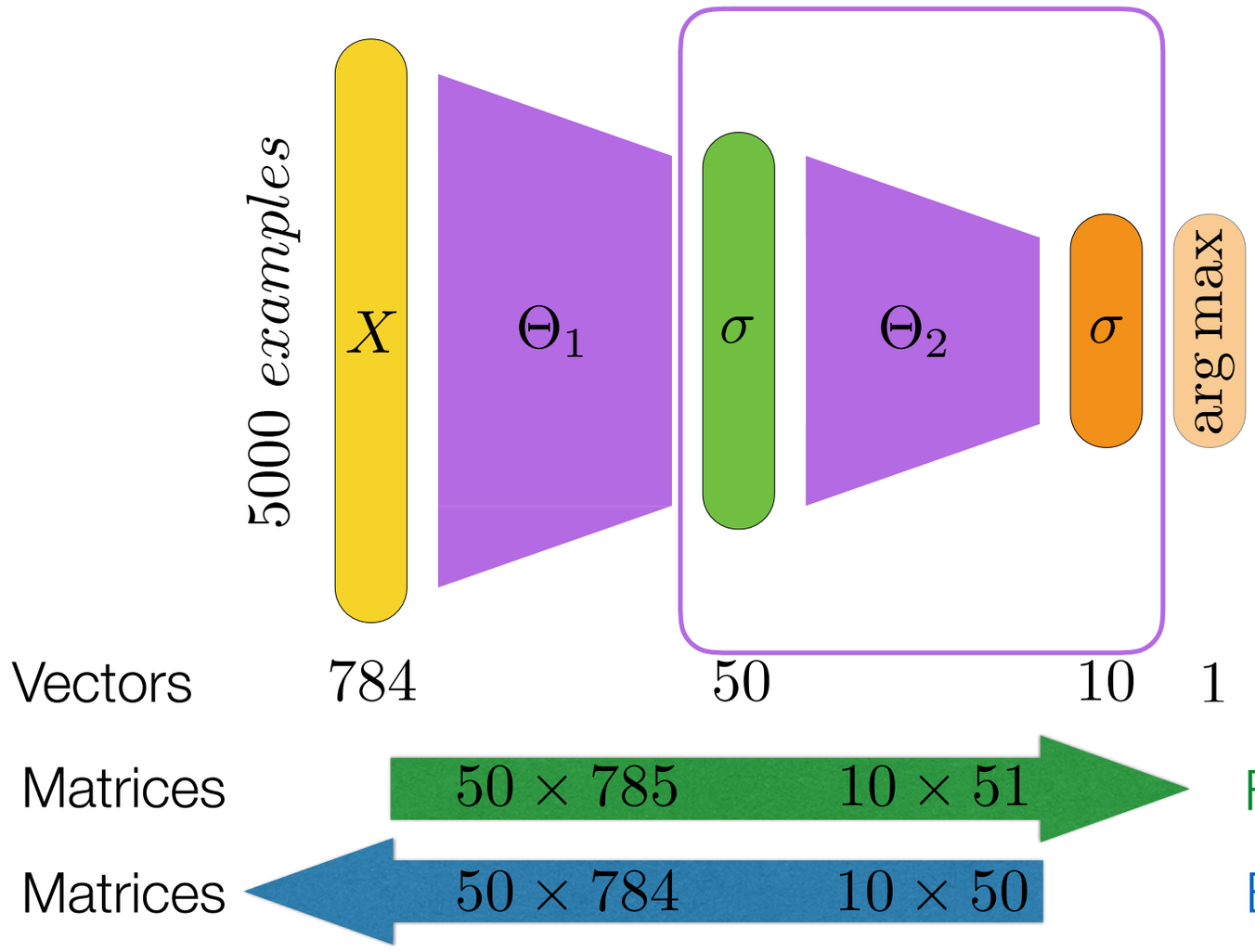

```python
def forward(Theta, X, active):
    N = X.shape[0]

    # Add the bias column
    X_ = np.concatenate((np.ones((N, 1)), X), 1)

    # Multiply by the weights
    z = np.dot(X_, Theta.T)

    # Apply the activation function
    a = active(z)

    return a
```

```python
def predict(Theta1, Theta2, X):
    h1 = forward(Theta1, X, sigmoid)
    h2 = forward(Theta2, h1, sigmoid)

    return np.argmax(h2, 1)
```

nn.py
forward.py

A practical example - MNIST

http://github.com/bmtgoncalves/Neural-Networks



nn.py
forward.py

Backprop

http://github.com/bmtgoncalves/Neural-Networks

```python
def backprop(Theta1, Theta2, X, y):
    N = X.shape[0]
    K = Theta2.shape[0]
    J = 0

    Delta2 = np.zeros(Theta2.shape)
    Delta1 = np.zeros(Theta1.shape)

    for i in range(N): # Forward propagation, saving intermediate results
        a1 = np.concatenate(([1], X[i]))  # Input layer
        z2 = np.dot(Theta1, a1)
        a2 = np.concatenate(([1], sigmoid(z2)))  # Hidden Layer
        z3 = np.dot(Theta2, a2)
        a3 = sigmoid(z3)  # Output layer
        y0 = one_hot(K, y[i])

        # Cross entropy
        J -= np.dot(y0.T, np.log(a3))+np.dot((1-y0).T, np.log(1-a3))

        # Calculate the weight deltas
        delta_3 = a3-y0
        delta_2 = np.dot(Theta2.T, delta_3)[1:]*sigmoidGradient(z2)
        Delta2 += np.outer(delta_3, a2)
        Delta1 += np.outer(delta_2, a1)

J /= N

Theta1_grad = Delta1/N
Theta2_grad = Delta2/N

return [J, Theta1_grad, Theta2_grad]
```

nn.py

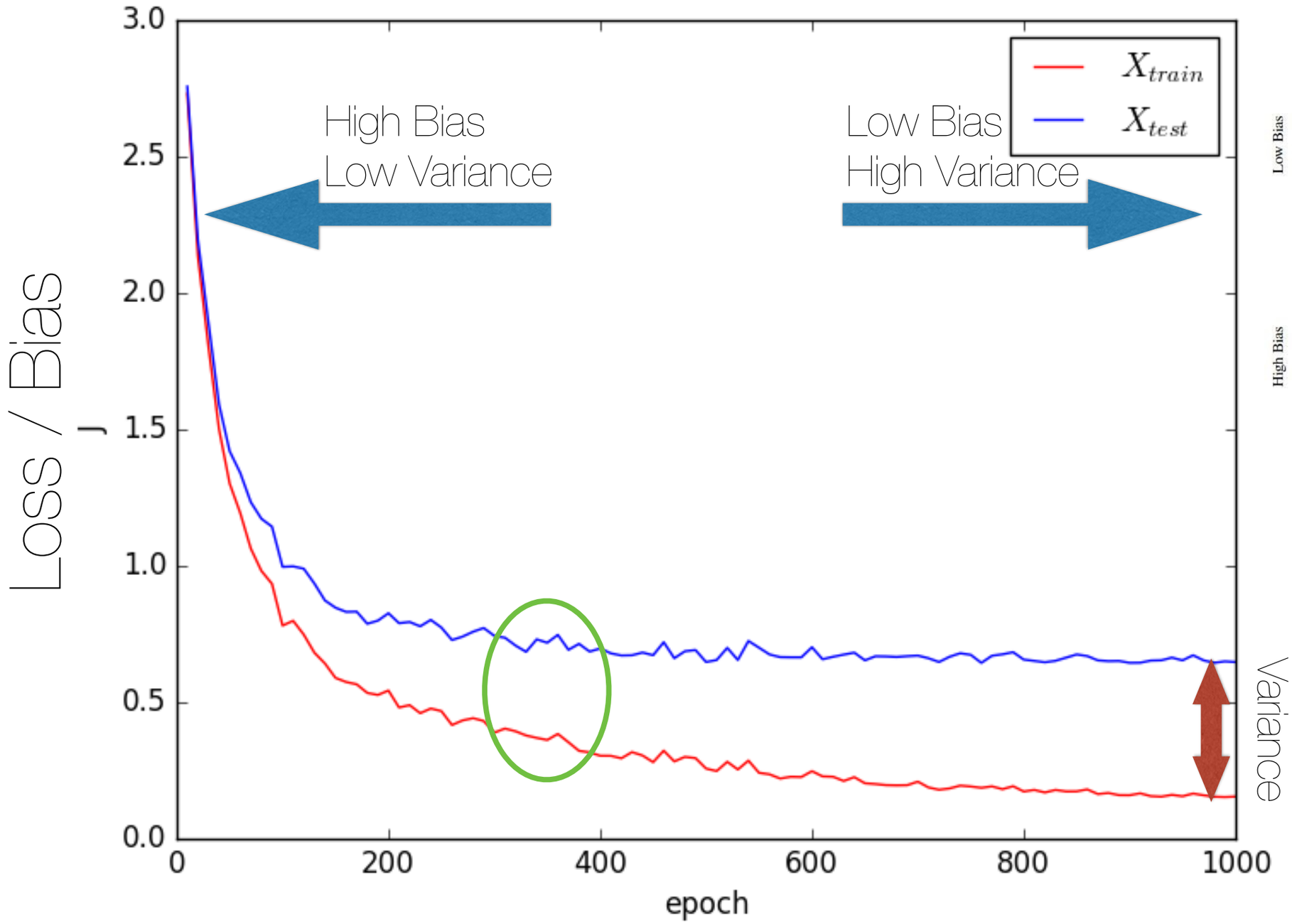
Training

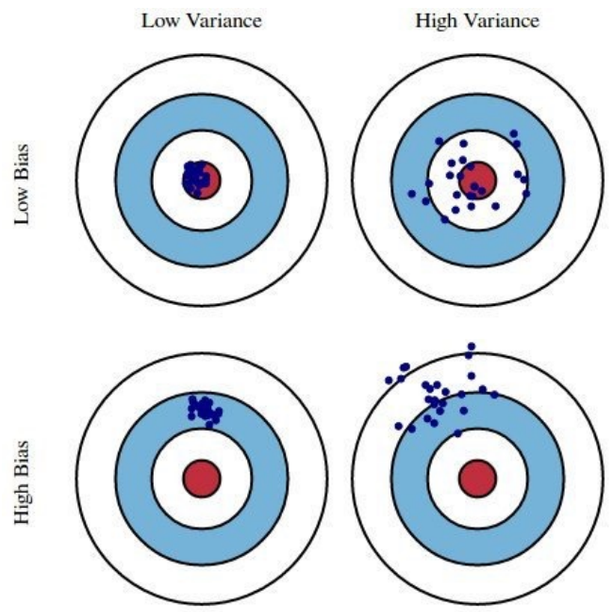
http://github.com/bmtgoncalves/Neural-Networks

```
Theta1 = init_weights(input_layer_size, hidden_layer_size)
Theta2 = init_weights(hidden_layer_size, num_labels)

iter = 0
tol = 1e-3
J_old = 1/tol
diff = 1

while diff > tol:
    J_train, Theta1_grad, Theta2_grad = backprop(Theta1, Theta2, X_train, y_train)

    diff = abs(J_old-J_train)
    J_old = J_train

    Theta1 -= .5*Theta1_grad
    Theta2 -= .5*Theta2_grad
```


nn.py

Bias-Variance Tradeoff

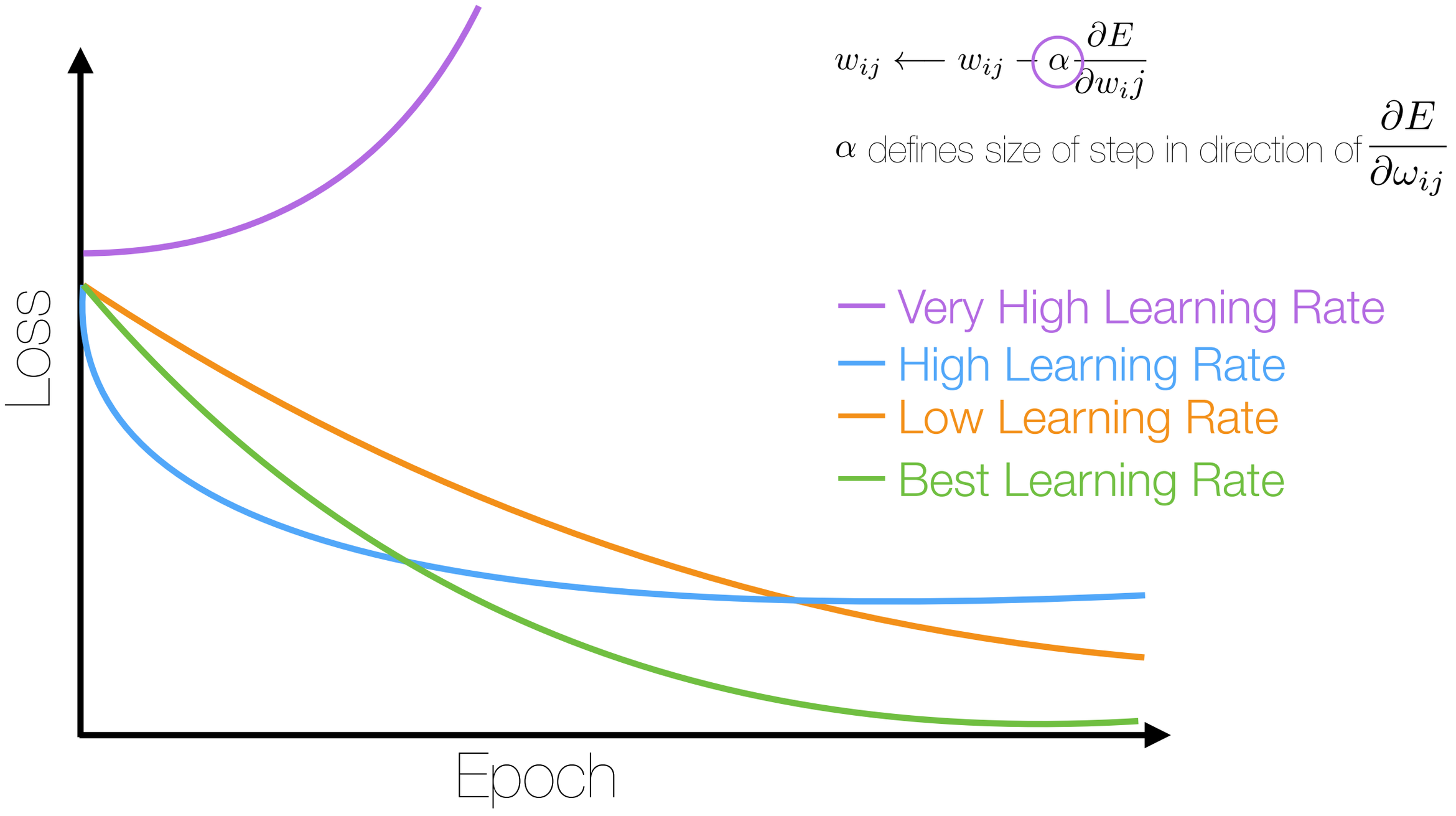




Learning Rate

$$w_{ij} \longleftarrow w_{ij} - \alpha \frac{\partial E}{\partial w_i j}$$

α defines size of step in direction of $\frac{\partial E}{\partial \omega_{ij}}$



Tips

- online learning - update weights after each case
  - might be useful to update model as new data is obtained
  - subject to fluctuations

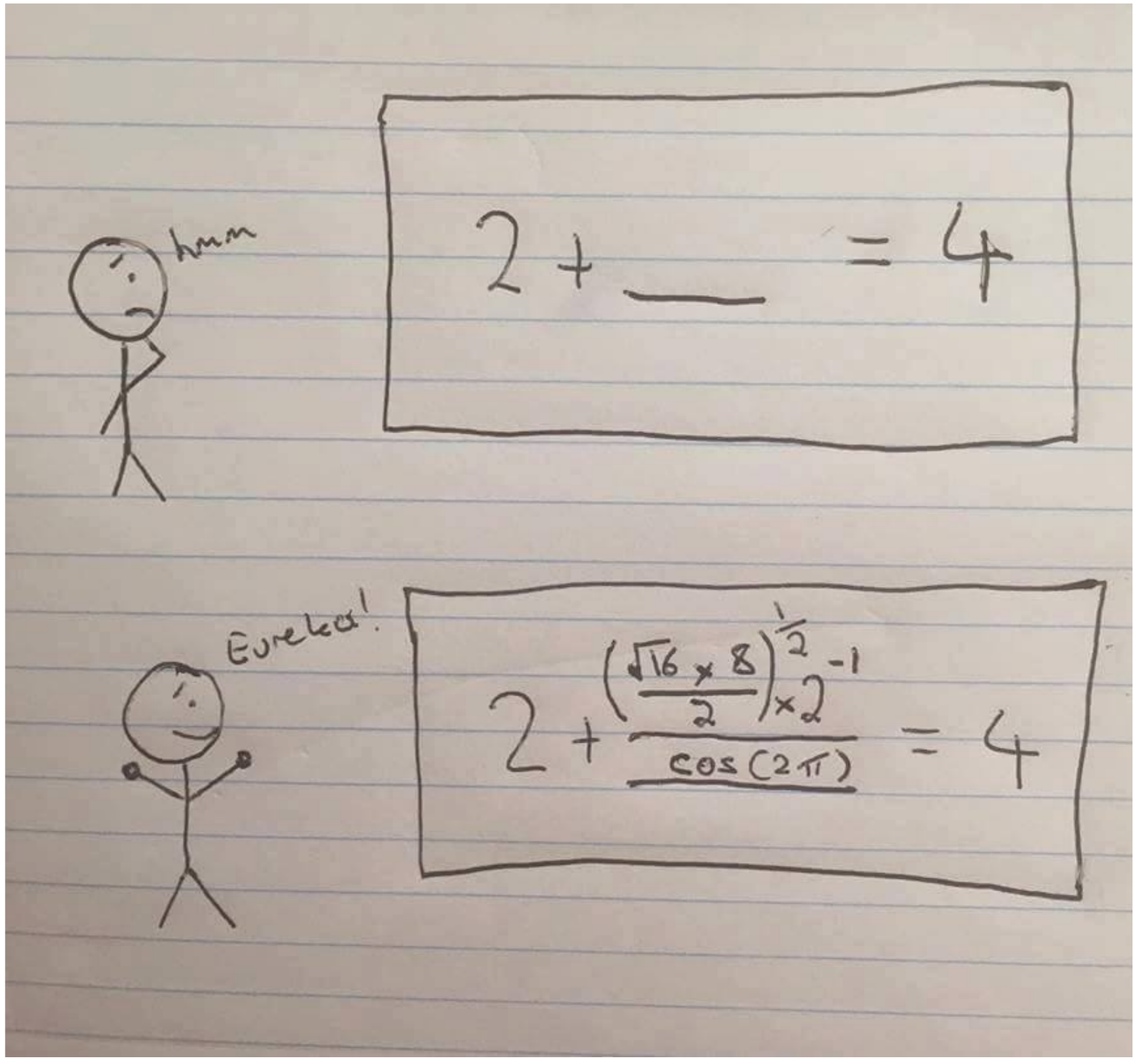
- mini-batch - update weights after a "small" number of cases
  - batches should be balanced
  - if dataset is redundant, the gradient estimated using only a fraction of the data is a good approximation to the full gradient.

- momentum - let gradient change the velocity of weight change instead of the value directly

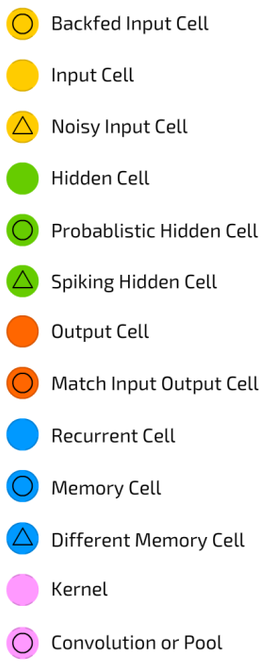
- rmsprop - divide learning rate for each weight by a running average of "recent" gradients

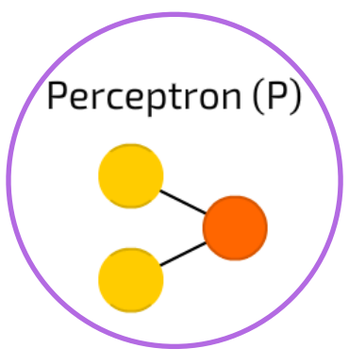
- learning rate - vary over the course of the training procedure and use different learning rates for each weight

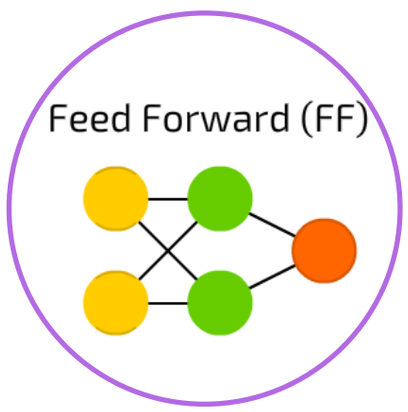
Interpretability

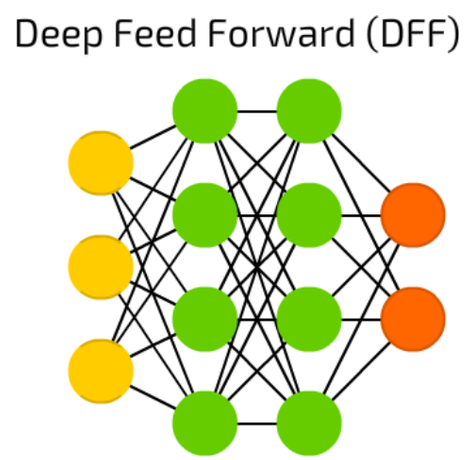


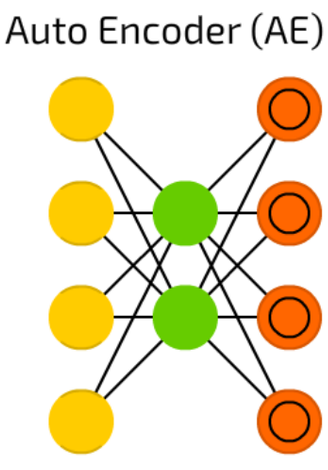
Neural Network Architectures

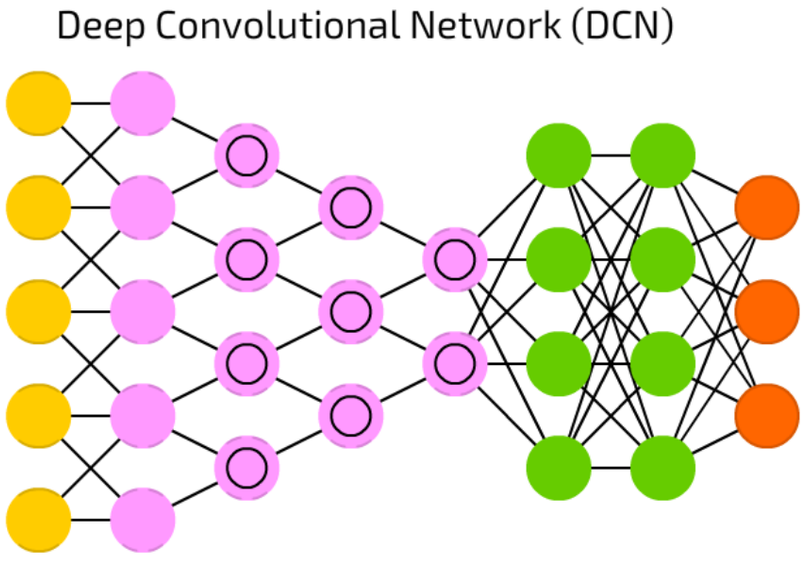












Deep Learning...