SGN-41006 Signal Interpretation

Spring 2016

Heikki Huttunen
University Lecturer, Dr.Eng.
Heikki.Huttunen@tut.fi

Department of Signal Processing
Tampere University of Technology, Finland

Course Organization

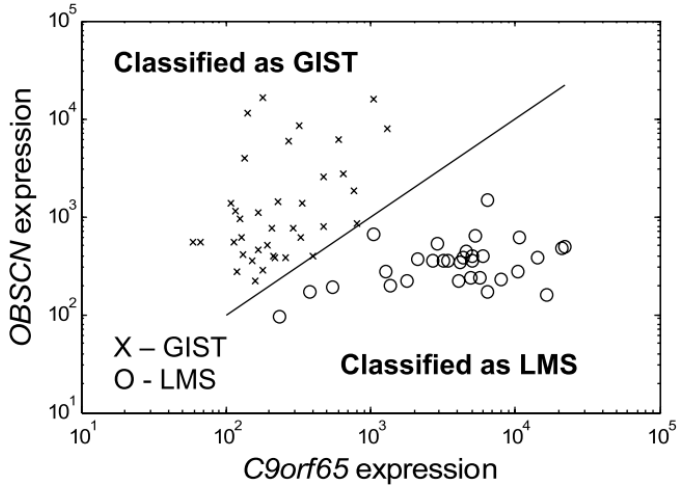
- Organized on 3rd period; January – February 2016.
- Lectures on Tuesdays 10-12 (TB111) and Fridays 10-12 (TB111).
- Exercises on Wed 10-12, Wed 12-14, Wed 14-16 and Thu 10-12 in TC407.
- More details:
  http://www.cs.tut.fi/courses/SGN-41006/

Course Contents

1. **Python:** Rapidly becoming the default platform for practical machine learning
2. **Estimation of Signal Parameters:** What are the phase, amplitude and frequency of this noisy sinusoid
3. **Detection Theory:** Detect whether there is a specific signal present or not
4. **Performance evaluation:** Cross-Validation, Bootstrapping, Receiver Operating Characteristics, other Error Metrics
5. **Machine Learning Models:** Logistic Regression, Support Vector Machine, Random Forests, Deep Learning
6. **Avoid Overlearning and Solve Ill-Posed Problems:** Regularization Techniques

Introduction

- Machine learning has become an important tool for multitude of scientific disciplines.
- Training based approaches are rapidly substituting traditional manually engineered pipelines.
- "Training based" = we show examples of what is interesting and hope the machine learns to do it for us
- A few modern research topics:
  - Image recognition (what is in this image and where?)
  - Speech recognition (what do I say?)
  - Medicine (data-driven diagnosis)



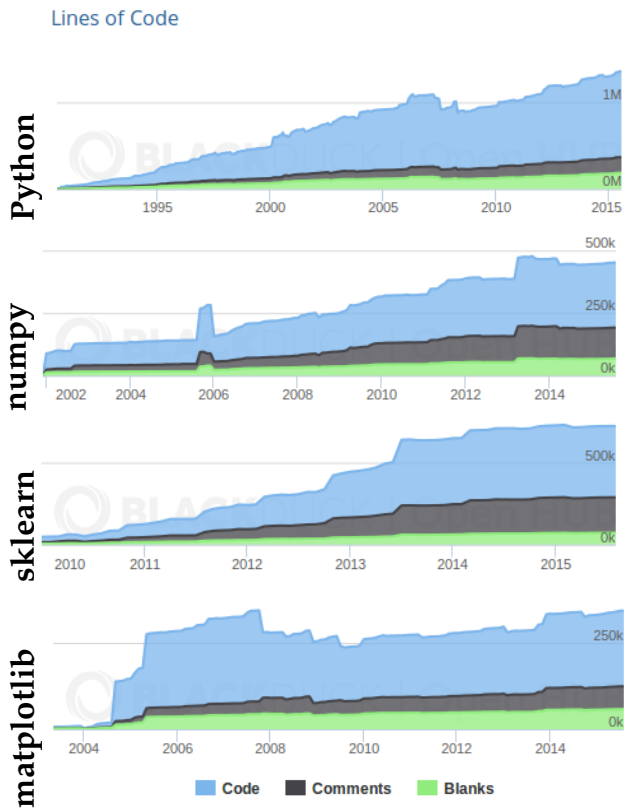
Price et al., "Highly accurate two-gene classifier for differentiating gastrointestinal stromal tumors and leiomyosarcomas," *PNAS 2007*.

Python in Scientific Computing

Why Python?

- Python is becoming increasingly central tool for data science.
- This was not always the case: 10 years ago everyone was using Matlab.
- However, due to licensing issues and heavy development of Python, scientific Python started to gain its user base.
- Python's strength is in its variability and huge community.
- There are 2 versions: Python 2.7 and 3.4. We'll use the former one as it's better supported by most packages interesting to us.

```
All Python releases are Open Source (see
http://www.opensource.org for
the Open Source Definition).
Historically, most, but not all, Python
releases have also been GPL-compatible;
```



Alternatives to Python in Science

Python vs. Matlab

- Matlab is #1 workhorse for linear algebra.
- Matlab is professionally maintained *product*.
- Some Matlab's toolboxes are great (Image Processing tb). Some are obsolete (Neural Network tb).
- New versions 2 twice a year. Amount of novelty varies.
- Matlab is expensive for non-educational users.

Python vs. R

- R is #1 workhorse for statistics and data analysis.
- Popularity: `http://www.kdnuggets.com/polls/2015/r-vs-python.html`
- R is great for specific data analysis and visualization needs.
- However, Python interfaces to other domains ranging from deep neural networks (Caffe, Theano) and image analysis (OpenCV) to even a fullblown webserver (Django)

Essential Modules

- **numpy**: The matrix / numerical analysis layer at the bottom
- **scipy**: Scientific computing utilities (linalg, FFT, signal/image processing...)
- **sklearn**: Machine learning (our focus here)
- **matplotlib**: Plotting and visualization
- **opencv**: Computer vision
- **pandas**: Data analysis
- **caffe, theano, minerva**: Deep neural networks
- **spyder**: The front end (Scientific PYthon Development EnviRonment)

Where to get Python?

- It is possible to construct your custom Python environment by installing individual modules.
- Alternatively, one may install a full distributions, such as
  - Anaconda http://www.continuum.io/ ← **my favorite**
  - Enthought Canopy http://www.enthought.com/
  - Pythonxy http://code.google.com/p/pythonxy/
- ...or in linux:

```
# apt-get install python
# apt-get install python-numpy
# apt-get install python-sklearn
# apt-get install python-matplotlib
# apt-get install spyder
```

The Language

- Python was designed to be a highly readable language.
- Python uses whitespace to delimit program blocks. First you hate it, later you love it.
- All used modules are imported using an `import` declaration.
- The members of a module are referred using the dot: `np.cos([1,2,3])`
- Interpreted language. Also interactive with IPython extensions.

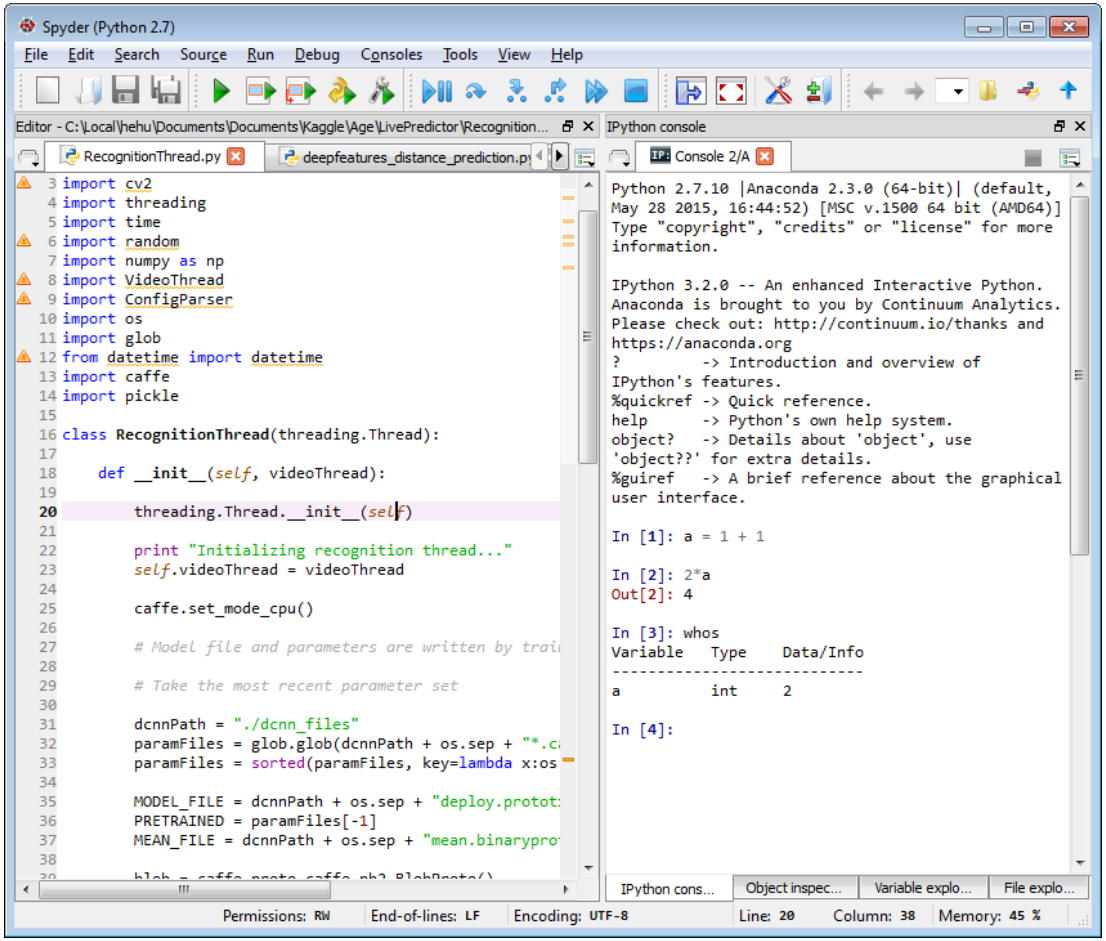
```
 8 import matplotlib.pyplot as plt
 9 import numpy as np
10 from sklearn.linear_model import Ridge
11 from sklearn.linear_model import Lasso
12
13 def get_obs_matrix(x, order = 5):
14     """
15     Return the observation matrix
16     constructed from powers of vector x.
17     """
18
19     H = []
20
21     for k in range(order):
22         H.append(x**k)
23
24     H = np.array(H).T
25
26     return H
27
28 if __name__ == "__main__":
29
30     # generate a noisy sinusoid
31
32     np.random.seed(2015)
33
34     x = np.arange(0,1.01,0.08)
35     y = np.cos(2 * 2*pi*x) \
36         + 0.35*np.random.randn(x.shape[0])
37     y2 = np.cos(1 * 2*pi*x) \
38         + 0.35*np.random.randn(x.shape[0])
39     x = np.arange(0,1.01,0.04)
```

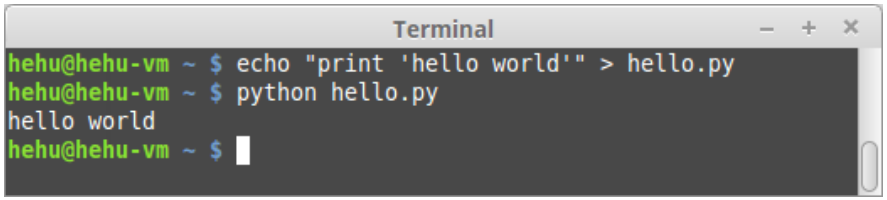
Things to Come

- Following slides will introduce the basic Python usage within scientific computing.
  - **The editor and the environment**
    - *Matlab slightly better than Python*
  - **Linear algebra**
    - *Matlab better than Python*
  - **Programming constructs** (loops, classes, etc.)
    - *Python better than Matlab*
  - **Machine learning**
    - *Python a lot better than Matlab*

Spyder Editor

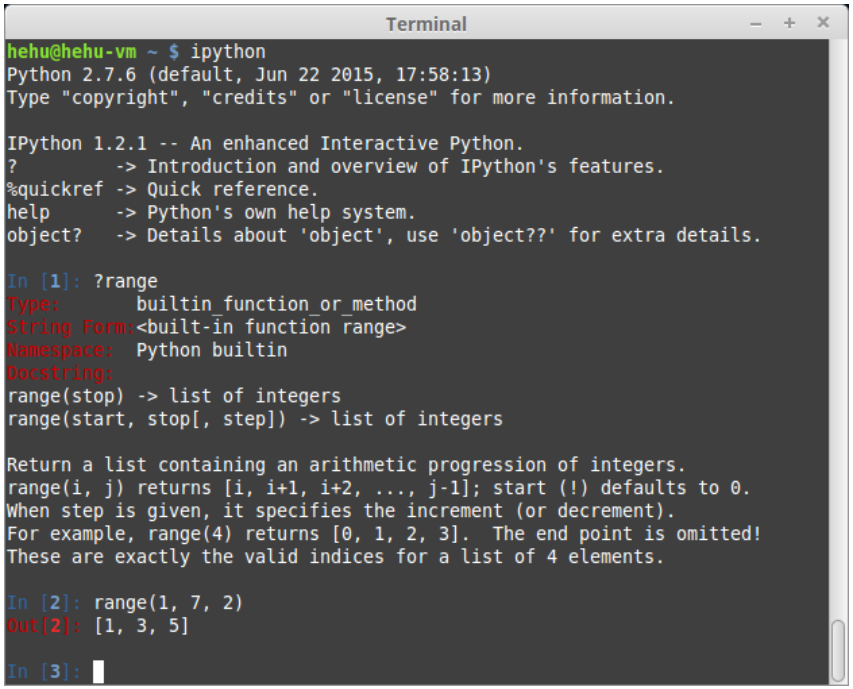
- In this course we use the *Spyder* editor.
- The editor window contains two panes: editor on the left and console on the right.
- F5: Run code; F9: Run selected region.
- Alternatively, you can use whatever editor you like, and run everything on the command line.





Python Basics

- Python code can be executed either from a script file (*.py) or in the interactive mode (just like Matlab).
- For the interactive mode; just execute *python* from the command line.
- Alternatively, *ipython* (if installed) starts Python in a more user-friendly mode:
  - Tab-completion works
  - Many utility functions (*e.g.*, `ls`, `pwd`, `cd`)
  - *Magic* functions (*e.g.*, `%run`, `%timeit`, `%edit`, `%pastebin`)



Command `range` *creates a list of integers. Compare to Matlab's syntax* `1:2:6`.

Help

- For each command, help is there to refresh your memory:

```
>>> help("".strip) # strip is a member of the string class
Help on built-in function strip:

strip(...)
    S.strip([chars]) -> string or unicode

    Return a copy of the string S with leading and trailing
    whitespace removed.
    If chars is given and not None, remove characters in chars instead.
    If chars is unicode, S will be converted to unicode before stripping
```

- In *ipython*, the shortcut ? is available, too (see previous slide).
- Many people prefer to Google for `python strip` instead; matter of taste.

NumPy

- Practically all scientific computing in Python is based on `numpy` and `scipy` modules.
- NumPy provides a numerical array as an alternative to Python list.
- The list type is very generic and accepts any mixture of data types.
- Although practical for generic manipulation, it is becomes inefficient in computing.
- Instead, the NumPy array is more limited and more focused on numerical computing.

```
# Python list accepts any data types
v = [1, 2, 3, "hello", None]
```

```
# We like to call numpy briefly "np"
>>> import numpy as np

# Define a numpy array (vector):
>>> v = np.array([1, 2, 3, 4])

# Note: the above actually casts a
# Python list into a numpy array.

# Resize into 2x2 matrix
>>> V = np.resize(v, (2, 2))

# Invert:
>>> np.linalg.inv(V)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

More on Vectors

- `np.arange` creates a range array (like `1:0.5:10` in Matlab)

```
>>> np.arange(1, 10, 0.5) # Arguments: (start, end, step)
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,  5.5,  6. ,
        6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])

# Note that the endpoint is not included (unlike Matlab).
```

- Most vector/matrix functions are similar to Matlab:

```
>>> np.linspace(1, 10, 5) # Arguments: (start, end, num_items)
array([  1.  ,   3.25,   5.5 ,   7.75,  10.  ])

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

>>> np.random.randn(2, 3)
array([[-2.23506417,  0.47311746,  0.05343861],
       [ 1.255074  , -0.03576461,  0.96121907]])
```

Matrices

- A matrix is defined similarly; either by specifying the values manually, or using special functions.

```
% A matrix is simply an array of arrays
% May seem complicated at first, but is in fact
% nice for N-D arrays.

>>> np.array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])

>>> from scipy.linalg import toeplitz, hilbert # You could also " ...import *"
>>> toeplitz([3, 1, -2])
array([[ 3,  1, -2],
       [ 1,  3,  1],
       [-2,  1,  3]])

>>> hilbert(3)
array([[ 1.        ,  0.5       ,  0.33333333],
       [ 0.5       ,  0.33333333,  0.25      ],
       [ 0.33333333,  0.25      ,  0.2       ]])
```

Matrices

- Matrix multiplication is different from Matlab:

```
>>> A = np.array([[1, 2], [3, 4]])
>>> B = np.array([[5, 6], [7, 8]])

>>> A * B # Elementwise product (Matlab: A .* B)
array([[ 5, 12],
       [21, 32]])

>>> np.dot(A, B) # Matrix product
array([[19, 22],
       [43, 50]])
```

- With the recently released NumPy version 1.10, matrix multiplication can be done with the new `@` operator: `A @ B`.

Indexing

- Indexing of vectors uses the colon notation, too.
- Below, we extract selected items from the vector `1...10`:

```
>>> x = np.arange(1, 11)
>>> x[0:8:2] # Unlike Matlab, indexing starts from 0
array([1, 3, 5, 7])

# Note: use square brackets for indexing
# Note2: colon operator has the order start:end:step;
# not start:step:end as in Matlab
```

- The start and end points can be omitted:

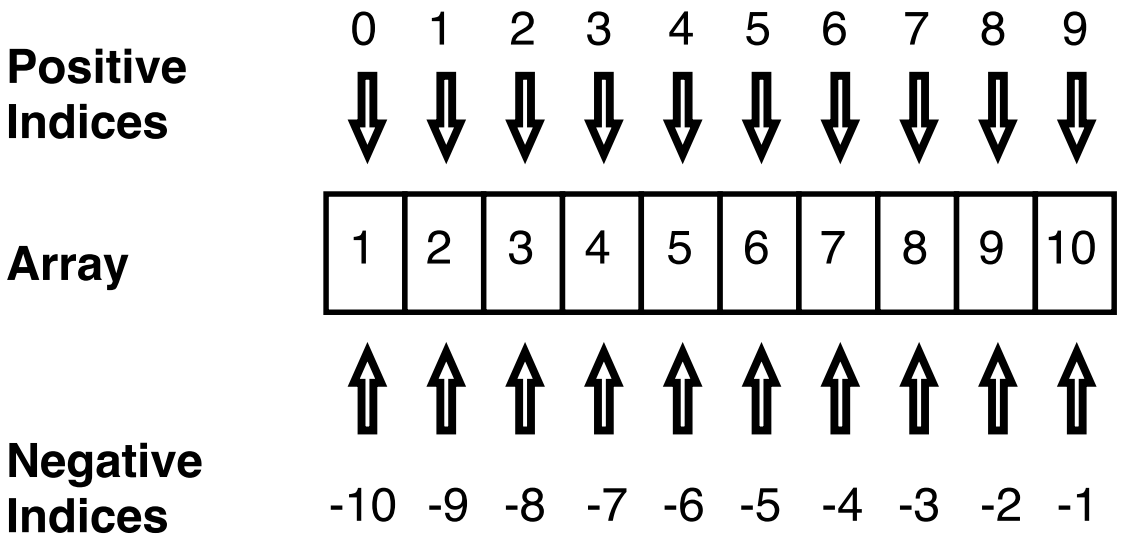
```
>>> x[5:] # All items from the 5'th
array([ 6,  7,  8,  9, 10])
>>> x[:5] # All items until the 5'th
array([1, 2, 3, 4, 5])
>>> x[::3] # All items with step 3
array([ 1,  4,  7, 10])
```

Indexing

- Negative indices are counted from the end (-1 = the last, -2 = second-to-last, etc.):

```
>>> x[-3:] # Three last items
array([ 8,  9, 10])
>>> x[::-1] # Items in inverse order
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
```

Indexing


Positive Indices
0 1 2 3 4 5 6 7 8 9
Array
1 2 3 4 5 6 7 8 9 10
Negative Indices
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1

Indexing

- Also matrices can be indexed similarly. This operation is called *slicing*, and the result is a *slice* of the matrix.

```
>>> M = np.reshape(np.arange(0, 36), (6, 6))
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])

>>> M[2:4, [1,2,4]]
array([[13, 14, 16],
       [19, 20, 22]])
```

- Here we request for items on the rows 2:4 = [2,3] and columns 1,2,4 (shown in red).
- Note, that with matrices, the first index is the row; not "x-coordinate".
- This order is called "Fortran style" or "column major" while the alternative is "C style" or "row major".

Indexing

- To specify only column or row indices, use ":" alone.

```
>>> M = np.reshape(np.arange(0, 36), (6, 6))
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])

>>> M[4:, :] # Alternatively: M[-2:, :]
array([[24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

- Now we wish to extract two bottom rows.
- `M[4:, :]` reads "give me all rows after the 4th and all columns".

N-Dimensional arrays

```
# Generate a random "image" array:
>>> A = np.random.rand(1000, 3, 96, 128)

# What size is it?
>>> A.shape
(1000L, 3L, 96L, 128L)

# Access the pixel (4, 3) of 2nd color channel
# of the 2nd image
>>> A[1, 2, 3, 4]
0.36569219631994954

# Request all color channels:
>>> A[1, :, 3, 4]
array([ 0.32306666,  0.60012626,  0.3656922 ])

# Request a complete 96x128 color channel:
>>> A[1, 2, :, :]
array([[ 0.19102217 ...
 0.88464718]])

# Equivalent shorter notation:
>>> A[1, 2, ...]
array([[ 0.19102217 ...
 0.88464718]])
```

- Higher-dimensional arrays are frequently encountered in machine learning.
- For example, a set of 1000 color images of size $w \times h = 128 \times 96$ is represented as a $1000 \times 3 \times 96 \times 128$ array.
- Here, dimensions are: image index, color channel, y-coordinate, x-coordinate.
- Sometimes, a shorter name is used: "(b, c, 0, 1) order".

Functions

```
# Define our first function
def hello(target):
    print "Hello " + target + "!"

>>> hello("world")
Hello world!

>>> hello("Finland")
Hello Finland!

# We can also define the default argument:
def hello(target = "world"):
    print "Hello " + target + "!"

>>> hello()
Hello world!

>>> hello("Finland")
Hello Finland!

# One can also assign using the name:

>>> hello(target = "Finland")
Hello Finland!
```

- Functions are defined using the `def` keyword.
- Function definition can appear anywhere in the code.
- Functions can be imported to other files using `import`.
- Function arguments can be *positional* or *named* (see code).
- Named arguments improve readability and are handy for setting the last argument in a long list.

Loops and Stuff

```
for lang in ['Assembler', 'Python', "Matlab", 'C++']:
   if lang in ["Assembler", "C++"]:
      print "I am ok with %s." % (lang)
   else:
      print "I love %s." % (lang)

I am ok with Assembler.
I love Python.
I love Matlab.
I am ok with C++.
```

```
# Read all lines of a file until the end

fp = open("myfile.txt", "r")
lines = []

while True:

    try:
        line = fp.readline()
        lines.append(line)
    except:
        # File ended
        break

fp.close()
```

- Loops and other usual programming constructs are easy to remember.
- `for` can loop over anything *iterable*, such as a list or a file.
- In Matlab, appending values to a vector in a loop is not recommended. Python lists are actual lists, so appending is fine.

Example: Reading in a Data File

- Suppose we need to read the following csv file into Python.

```
1 Sample_ID,M1;M2;M3;M4;M5;M6;M7;M8;M9;M10;M11;M12;M1
2 S0;0.063915;0.033242;0.018484;0.0086177;0.035629;0.
3 S1;0.025409;0.051085;0.056305;0.021738;0.02741;0.01
4 S2;0.025536;0.036123;0.054195;0.009735;0.027521;0.0
5 S3;0.012817;0.029652;0.07929;0.050677;0.039737;0.05
6 S4;0.019846;-0.010577;-0.0075045;0.019042;0.068786;
```

- The file consists of 216 rows (samples) with 4000 measurements each.
- We will write file reading code from scratch.
- Alternatively, many modules contain csv-reading functions
  - `numpy.loadtxt` or `numpy.genfromtxt`
  - `csv.reader`
  - `pandas.read_csv`

Example: Reading in a Data File

```
import numpy as np

if __name__ == "__main__":

    X = [] # Rows of the file go here

    # We use Python's with statement.
    # Then we do not have to worry
    # about closing it.

    with open("ovarian.csv", "r") as fp:

        # File is iterable, so we can
        # read it directly (instead of
        # using readline).

        for line in fp:

            # Skip the first line:
            if "Sample_ID" in line:
                continue
            # Otherwise, split the line
            # to numbers:
            values = line.split(";")

            # Omit the first item
            # ("S1" or similar):
            values = values[1:]

            # Cast each item from
            # string to float:
            values = [float(v) for v in
                  values]

            # Append to X
            X.append(values)

    # Now, X is a list of lists. Cast to
    # Numpy array:
    X = np.array(X)

    print "All data read."
    print "Result size is %s" % (str(X.
          shape))
```

Classification

- Many machine learning problems can be posed as **classification** tasks.
- Most classification tasks can be posed as a problem of partitioning a vector space into disjoint regions.
- These problems consist of following components:

$$\text{Samples: } \mathbf{x}[0], \mathbf{x}[1], \ldots, \mathbf{x}[N-1] \in \mathbb{R}^P$$
$$\text{Class labels: } y[0], y[1], \ldots, y[N-1] \in \{1, 2, \ldots, C\}$$
$$\text{Classifier: } F(\mathbf{x}) : \mathbb{R}^P \mapsto \{1, 2, \ldots, C\}$$

- Now, the task is to find the function F that maps the samples most accurately to their corresponding labels.
- For example: Find the function F that minimizes the number of erroneous predictions, *i.e.*, the cases $F(\mathbf{x}[k]) \neq y[k]$.

Regression

- The second large class of machine learning problems consists of **regression** tasks.
- For regression, the output is real-valued instead of categorical.
- These problems consist of following components:

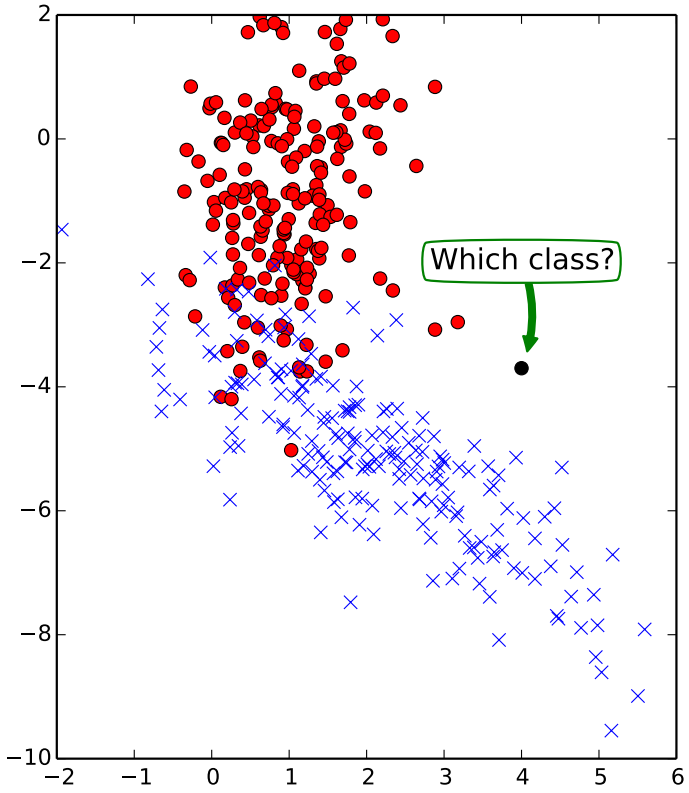
$$\begin{aligned} \text{Inputs: } & \mathbf{x}[0], \mathbf{x}[1], \ldots, \mathbf{x}[N-1] \in \mathbb{R}^P \\ \text{Targets: } & y[0], y[1], \ldots, y[N-1] \in \mathbb{R} \\ \text{Predictor: } & F(\mathbf{x}) : \mathbb{R}^P \mapsto \mathbb{R} \end{aligned}$$

- This time the task is to find the function F that maps the samples most accurately to their corresponding targets.
- For example: Find the function F that minimizes the squared sum of distances between predictions and targets:

$$\mathcal{E} = \sum_{k=0}^{N-1} \left(y[k] - F(\mathbf{x}[k]) \right)^2 .$$

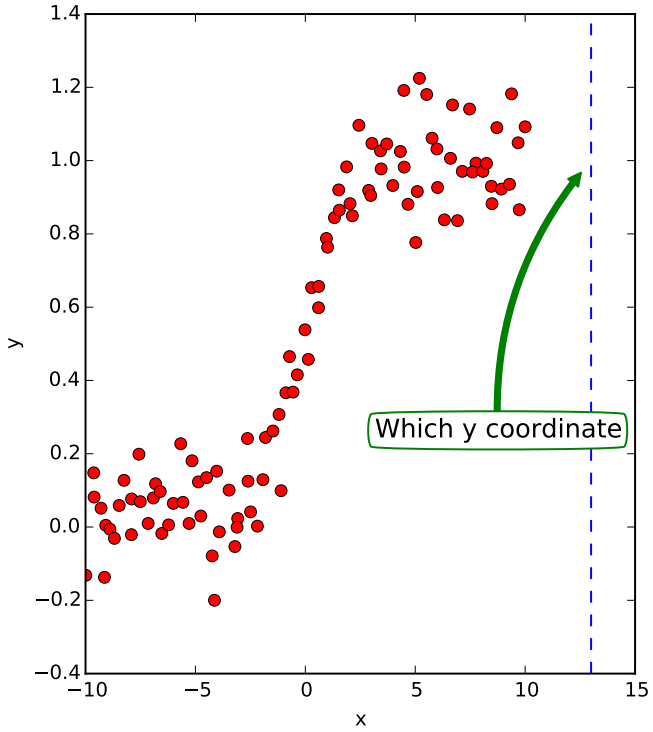
Classification Example

- For example, consider the 2-dimensional dataset on the right.
- The data consists of 200 blue crosses and 200 red circles.
- Based on these data, what would be a good partition of the 2D space into "red" and "blue" regions?
- What kind of boundaries are allowed between the regions?
  - Straight lines?
  - Continuous curved boundaries?
  - Boundary without any restriction?
- Note: In 2-D this can be solved manually, but not in 4000-D.



Regression Example

- For example, consider the 1-dimensional data on the right.
- The data consists of 100 data points, where y coordinate is a function of x.
- Based on these data, what would be a good prediction of the target value at $x = 1.3$ (the dashed line)?
- An obvious solution is to fit a curve into the data points. What kind of forms may the curve have?
  - Straight lines?
  - Continuous curves?



Estimation and Detection Theory

Classical Estimation and Detection Theory

- Before the machine learning part, we will take a look at classical estimation theory.
- Estimation theory has many connections to the foundations of modern machine learning.
- Outline of the next few hours:
  - Estimation theory: Fundamentals
  - Estimation theory: Maximum likelihood
  - Estimation theory: Examples
  - Detection theory: Fundamentals
  - Detection theory: Error metrics
  - Estimation theory: Examples

Introduction - estimation

- Our goal is to estimate the values of a group of parameters from data.
- Examples: radar, sonar, speech, image analysis, biomedicine, communications, control, seismology, etc.
- *Parameter estimation*: Given an N-point data set $\mathbf{x} = \{x[0], x[1], \ldots, x[N-1]\}$ which depends on the unknown parameter $\theta \in \mathbb{R}$, we wish to design an *estimator* for θ

$$\hat{\theta} = g(x[0], x[1], \ldots, x[N-1]).$$

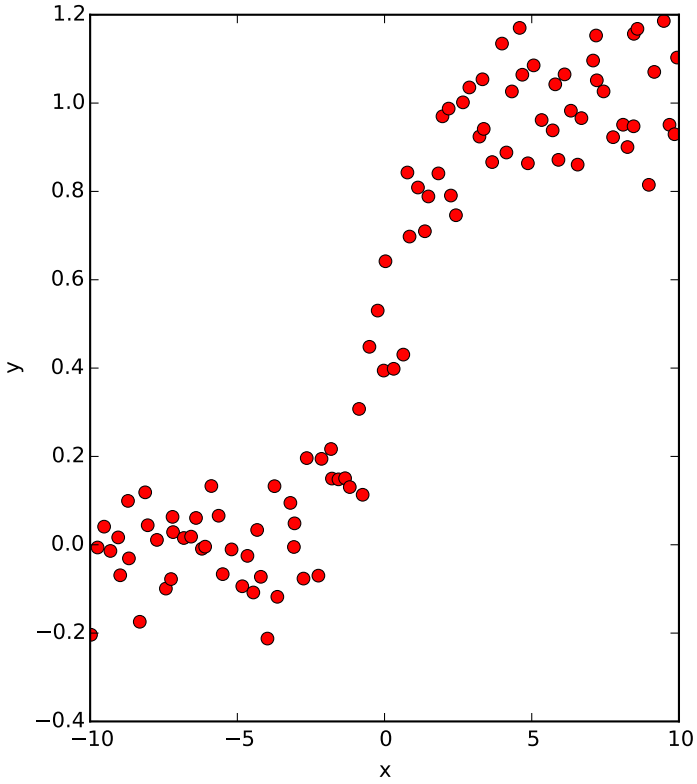
- The question is how to determine a good model and its parameters.

Introductory Example – Straight line

- Suppose we have the illustrated time series and would like to approximate the relationship of the two coordinates.
- The relationship looks linear, so we could assume the following model:

$$y[n] = ax[n] + b + w[n],$$

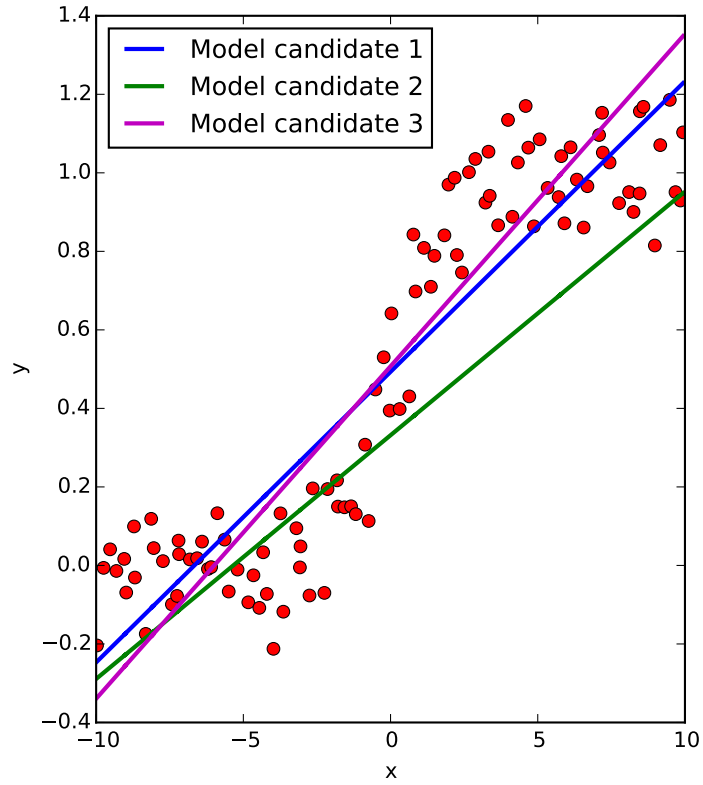
with $a \in \mathbb{R}$ and $b \in \mathbb{R}$ unknown and $w[n] \sim \mathcal{N}(0, \sigma^2)$[a]



[a] The normal distribution with mean 0 and variance σ^2.

Introductory Example – Straight line

- Each pair of a and b represent one line.
- Which line of the three would best describe the data set? Or some other line?



Introductory Example – Straight line

- It can be shown that the best solution (in the *maximum likelihood* sense; to be defined later) is given by

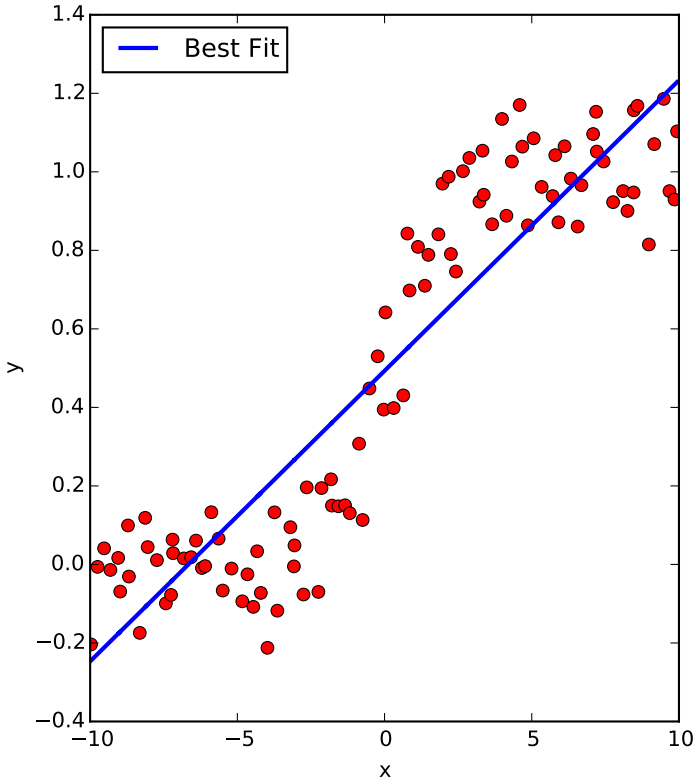
$$\begin{aligned}\hat{a} &= -\frac{6}{N(N+1)}\sum_{n=0}^{N-1} y(n) + \frac{12}{N(N^2-1)}\sum_{n=0}^{N-1} x(n)y(n)\\ \hat{b} &= \frac{2(2N-1)}{N(N+1)}\sum_{n=0}^{N-1} y(n) - \frac{6}{N(N+1)}\sum_{n=0}^{N-1} x(n)y(n).\end{aligned}$$

- Or, as we will later learn, in an easy matrix form:

$$\hat{\boldsymbol{\beta}} = \begin{pmatrix}\hat{a}\\ \hat{b}\end{pmatrix} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

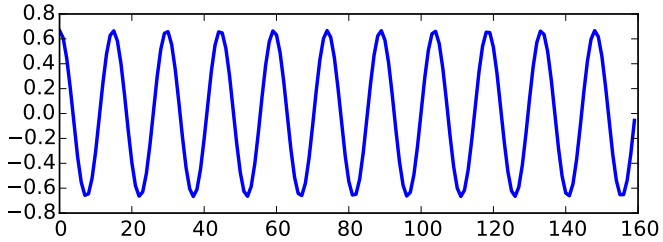
Introductory Example – Straight line

- In this case, $\hat{a} = 0.07401$ and $\hat{b} = 0.49319$, which produces the line shown on the right.

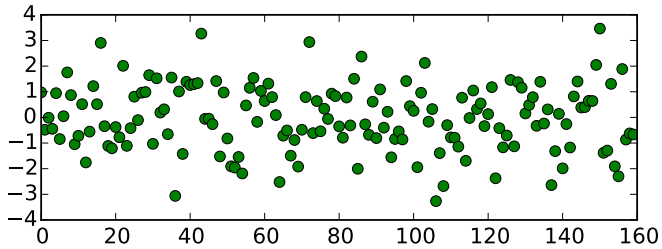


Introductory Example 2 – Sinusoid

- Consider transmitting the sinusoid below.



- When the data is received, it is corrupted by noise and the received samples look like below.



- Can we recover the parameters of the sinusoid?

Introductory Example 2 – Sinusoid

- In this case, the problem is to find good values for A, f_0 and ϕ in the following model:

$$x[n] = A\cos(2\pi f_0 n + \phi) + w[n],$$

with $w[n] \sim \mathcal{N}(0, \sigma^2)$.

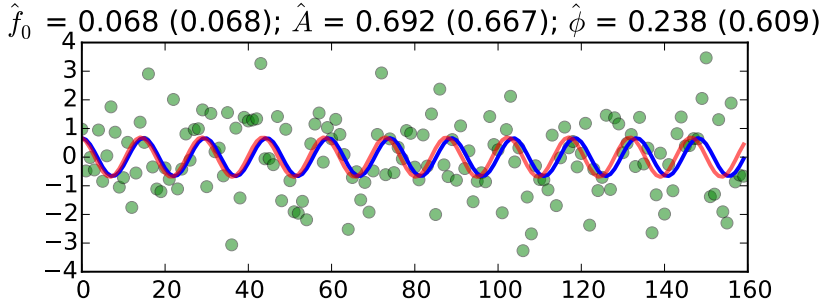
Introductory Example 2 – Sinusoid

- We will learn that the *maximum likelihood estimator; MLE* for parameters A, f_0 and ϕ are given by

$$\begin{aligned}
\hat{f}_0 &= \text{value of } f \text{ that maximizes } \left| \sum_{n=0}^{N-1} x(n) e^{-2\pi i f n} \right|, \\
\hat{A} &= \frac{2}{N} \left| \sum_{n=0}^{N-1} x(n) e^{-2\pi i \hat{f}_0 n} \right| \\
\hat{\phi} &= \arctan \frac{-\sum_{n=0}^{N-1} x(n) \sin(2\pi \hat{f}_0 n)}{\sum_{n=0}^{N-1} x(n) \cos(2\pi \hat{f}_0 n)}.
\end{aligned}$$

Introductory Example 2 – Sinusoid

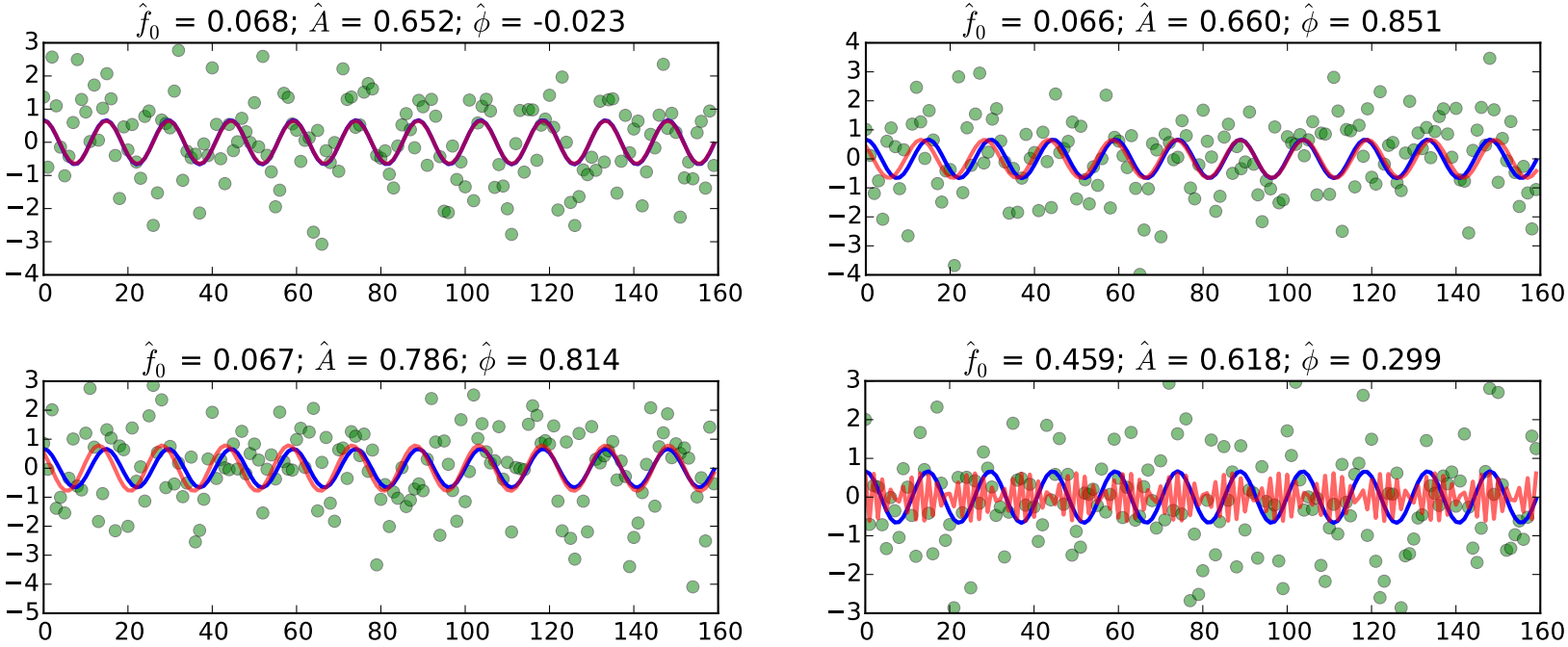
- It turns out that the sinusoidal parameter estimation is very successful:



- The blue curve is the original sinusoid, and the red curve is the one estimated from the green circles.
- The estimates are shown in the figure (true values in parentheses).

Introductory Example 2 – Sinusoid

- However, the results are different for each *realization* of the noise $w[n]$.

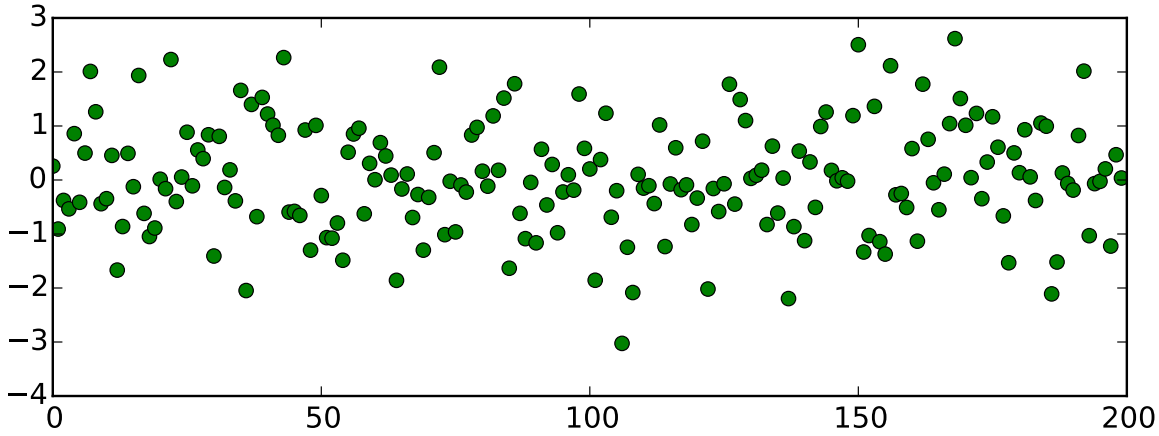


Introductory Example 2 – Sinusoid

- Thus, we're not very interested in an individual case, but rather on the distributions of estimates
  - What are the expectations: $E[\hat{f}_0]$, $E[\hat{\phi}]$ and $E[\hat{A}]$?
  - What are their respective variances?
  - Could there be a better formula that would yield smaller variance?
  - If yes, how to discover the better estimators?

Example of the Variance of an Estimator

- Consider the estimation of the mean of the following measurement data:



Example of the Variance of an Estimator

- Now we're searching for the estimator $\hat{A}$ in the model

$$x[n] = A + w[n],$$

  with $w[n] \sim \mathcal{N}(0, \sigma^2)$ where σ^2 is also unknown.
- A natural estimator of A is the sample mean:

$$\hat{A} = \frac{1}{N} \sum_{n=0}^{N-1} x[n].$$

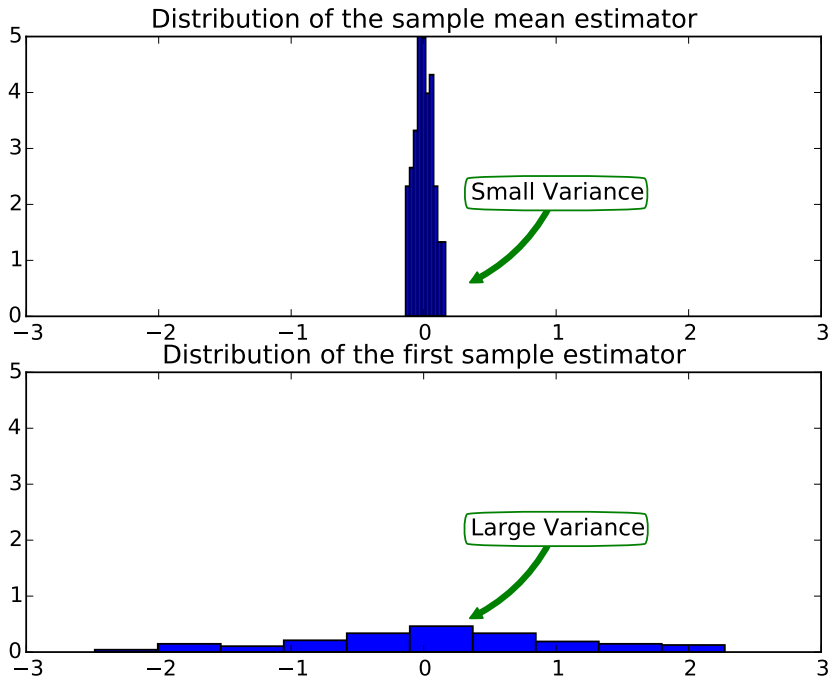
- Alternatively, one might propose to use only the first sample as such:

$$\check{A} = x[0].$$

- How to justify that the first one is better?

Example of the Variance of an Estimator

- Method 1: estimate variances **empirically**.
- Histograms of the estimates over 100 data realizations are shown on the right.
- In other words, we synthesized 100 versions of the data with the same statistics.
- Each synthetic sample produces one estimate of the mean for both estimators.



Example of the Variance of an Estimator

```
mean_estimates_1 = []
mean_estimates_2 = []

for iteration in range(100):

    # Create a random sample of data from N(0,1)
    x = np.random.randn(numSamples)

    # Compute the estimate of the mean using
    # the sample mean
    aHat_1 = x.mean() # or np.mean(x)

    # Compute the "first sample estimator"
    aHat_2 = x[0]

    # Append the computed values to our lists:
    mean_estimates_1.append(aHat_1)
    mean_estimates_2.append(aHat_2)

# Plot the empirical distributions
plt.figure()
subfig_1 = plt.subplot(211)
subfig_1.hist(mean_estimates_1, normed = True)

subfig_2 = plt.subplot(212)
subfig_2.hist(mean_estimates_2, normed = True)

plt.show()
```

- Attached is the code for the experiment of previous slide.
- Requires the following imports:

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

Example of the Variance of an Estimator

- Method 2: estimate variances **analytically**.
- Namely, it is easy to compute variances in a closed form:

$$\begin{aligned}
\textbf{Estimator 1: } \mathrm{var}(\hat{A}) &= \mathrm{var}\left(\frac{1}{N}\sum_{n=0}^{N-1} x[n]\right) \\
&= \frac{1}{N^2}\sum_{n=0}^{N-1} \mathrm{var}(x[n]) \\
&= \frac{1}{N^2} N\sigma^2 = \frac{\sigma^2}{N}.
\end{aligned}$$

$$\textbf{Estimator 2: } \mathrm{var}(\check{A}) = \mathrm{var}(x[0]) = \sigma^2.$$

Example of the Variance of an Estimator

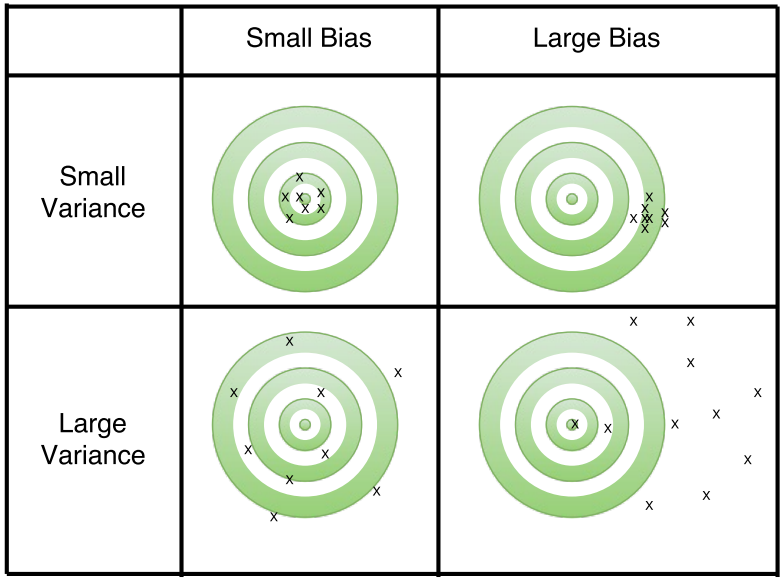
- Compared to the "First sample estimator" $\check{A} = x[0]$, the estimator variance of $\hat{A}$ is one N'th.
- The analytical approach is clearly the desired one whenever possible:
  - Faster, more elegant and less prone to random effects.
  - Often also provides proof that there exists no estimator that would be more efficient.
- Usually can be done for easy cases.
- More complicated scenarios can only be studied empirically.

Estimator Design

- There are a few well established approaches for estimator design:
  - **Minimum Variance Unbiased Estimator (MVU):** Analytically discover the estimator that minimizes the output variance among all *unbiased* estimators.
  - **Maximum Likelihood Estimator (ML):** Analytically discover the estimator that maximizes the likelihood of observing the measured data.
  - Others: **Method of Moments (MoM)** and **Least Squares (LS)**.
- Our emphasis will be on Maximum Likelihood, as it appears in the machine learning part as well.

Minimum Variance Unbiased Estimator

- Commonly the MVU estimator is considered optimal.
- However, finding the MVU estimator may be difficult. The MVUE may not even exist.
- We will not concentrate on this estimator design approach. Interested reader may consult, *e.g.*, S. Kay: *Fundamentals of Statistical Signal Processing: Volume 1* (1993).
- For an overview, read Wikipedia articles on *Minimum-variance unbiased estimator* and *Lehmann–Scheffé theorem*.



Maximum Likelihood Estimation

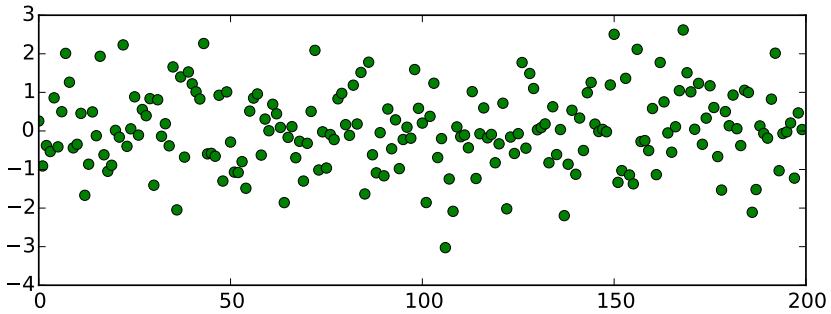
- Maximum likelihood (ML) is the most popular estimation approach due to its applicability in complicated estimation problems.
- The method was proposed by Fisher in 1922, though he published the basic principle already in 1912 as a third year undergraduate.
- The basic principle is simple: find the parameter θ that is the most probable to have generated the data $\mathbf{x}$.
- The ML estimator is in general not optimal in the minimum variance sense. Neither is it unbiased.

The Likelihood Function

- Consider again the problem of estimating the mean level A of noisy data.
- Assume that the data originates from the following model:

$$x[n] = A + w[n],$$

where $w[n] \sim \mathcal{N}(0, \sigma^2)$: Constant plus Gaussian random noise with zero mean and unit variance.



The Likelihood Function

- For simplicity, consider the first sample estimator for estimating A.
- We assume normally distributed $w[n]$, *i.e.*, the following probability density function (PDF):

$$p(w[n]) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2\sigma^2}(w[n])^2\right]$$

- Since $x[n] = A + w[n]$, we can substitute $w[n] = x[n] - A$ above to describe the PDF of $x[n]$[1]:

$$p(x[n]; A) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2\sigma^2}(x[n] - A)^2\right]$$

The Likelihood Function

- Thus, our first sample estimator has the PDF

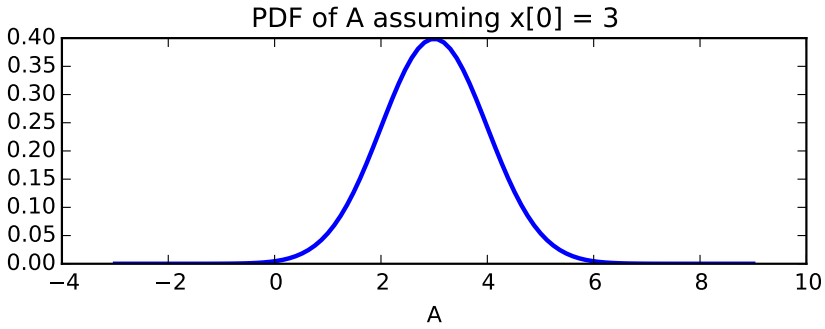
$$p(x[0];A) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2\sigma^2}(x[0]-A)^2\right]$$

- Now, suppose we have observed $x[0]$, say $x[0] = 3$.
- Then some values of A are more likely than others and we can derive the complete PDF of A easily.

The Likelihood Function

- Actually, the PDF of A has the same form as the PDF of $x[0]$:

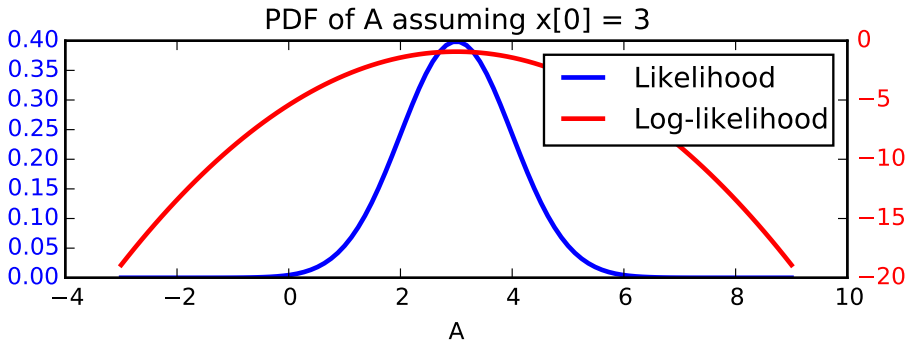
$$\text{pdf of } A = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2\sigma^2}(3-A)^2\right]$$



- This function is called *the likelihood function* of A, and its maximum the *maximum likelihood estimate*.

The Likelihood Function

- In summary: If the PDF of the data is viewed as a function of the unknown parameter (with fixed data), it is called the *likelihood function*.
- Often the likelihood function has an exponential form. Then it's usual to take the natural logarithm to get rid of the exponential. Note that the maximum of the new *log-likelihood* function does not change.



ML Example

- Consider the familiar example of estimating the mean of a signal:

$$x[n] = A + w[n], \qquad n = 0, 1, \ldots, N-1,$$

  with $w[n] \sim \mathcal{N}(0, \sigma^2)$.
- The noise samples $w[n]$ are assumed independent, so the distribution of the whole batch of samples $\mathbf{x} = (x[0], \ldots, x[N-1])$ is obtained by multiplication:

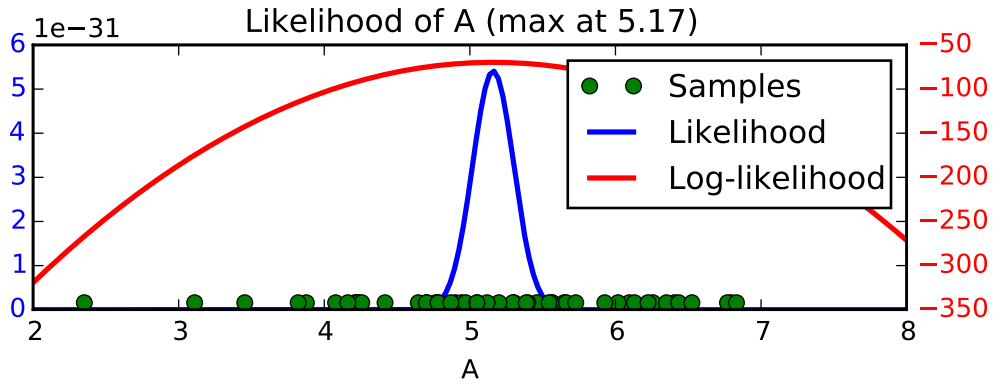
$$p(\mathbf{x}; A) = \prod_{n=0}^{N-1} p(x[n]; A) = \frac{1}{(2\pi\sigma^2)^{\frac{N}{2}}} \exp\left[-\frac{1}{2\sigma^2} \sum_{n=0}^{N-1} (x[n] - A)^2\right]$$

ML Example

- When we have observed the data $\mathbf{x}$, we can turn the problem around and consider what is the most likely parameter A that generated the data.
- Some authors emphasize this by turning the order around: $p(A; \mathbf{x})$ or give the function a different name such as $L(A; \mathbf{x})$ or $\ell(A; \mathbf{x})$.
- So, consider $p(\mathbf{x}; A)$ as a function of A and try to maximize it.

ML Example

- The picture below shows the likelihood function and the log-likelihood function for one possible realization of data.
- The data consists of 50 points, with true $A = 5$.
- The likelihood function gives the probability of observing these particular points with different values of A.



ML Example

- Instead of finding the maximum from the plot, we wish to have a closed form solution.
- Closed form is faster, more elegant, accurate and numerically more stable.
- Just for the sake of an example, below is the code for the stupid version.

```
# The samples are in array called x0

x = np.linspace(2, 8, 200)
likelihood = []
log_likelihood = []

for A in x:
    likelihood.append(gaussian(x0, A, 1).prod())
    log_likelihood.append(gaussian_log(x0, A, 1).sum())

print "Max likelihood is at %.2f" % (x[np.argmax(log_likelihood)])
```

ML Example

- Maximization of $p(\mathbf{x}; A)$ directly is nontrivial. Therefore, we take the logarithm, and maximize it instead:

$$p(\mathbf{x}; A) = \frac{1}{(2\pi\sigma^2)^{\frac{N}{2}}} \exp\left[-\frac{1}{2\sigma^2}\sum_{n=0}^{N-1}(x[n]-A)^2\right]$$

$$\ln p(\mathbf{x}; A) = -\frac{N}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{n=0}^{N-1}(x[n]-A)^2$$

- The maximum is found via differentiation:

$$\frac{\partial \ln p(\mathbf{x}; A)}{\partial A} = \frac{1}{\sigma^2}\sum_{n=0}^{N-1}(x[n]-A)$$

ML Example

- Setting this equal to zero gives

$$
\begin{aligned}
\frac{1}{\sigma^2}\sum_{n=0}^{N-1}(x[n]-A) &= 0 \\
\sum_{n=0}^{N-1}(x[n]-A) &= 0 \\
\sum_{n=0}^{N-1}x[n]-\sum_{n=0}^{N-1}A &= 0 \\
\sum_{n=0}^{N-1}x[n]-NA &= 0 \\
\sum_{n=0}^{N-1}x[n] &= NA \\
A &= \frac{1}{N}\sum_{n=0}^{N-1}x[n]
\end{aligned}
$$

Conclusion

- *What did we actually do?*
  - We proved that the **sample mean** is the maximum likelihood estimator for the **distribution mean**.
- *But I could have guessed this result from the beginning. What's the point?*
  - We can do the same thing for cases where you can not guess.

Example: Sinusoidal Parameter Estimation

- Consider the model

$$x[n] = A\cos(2\pi f_0 n + \phi) + w[n]$$

with $w[n] \sim \mathcal{N}(0, \sigma^2)$. It is possible to find the MLE for all three parameters: $\boldsymbol{\beta} = [A, f_0, \phi]^T$.

- The PDF is given as

$$p(\mathbf{x}; \boldsymbol{\beta}) = \frac{1}{(2\pi\sigma^2)^{\frac{N}{2}}} \exp\left[-\frac{1}{2\sigma^2}\sum_{n=0}^{N-1}\underbrace{(x[n] - A\cos(2\pi f_0 n + \phi)}_{w[n]})^2\right]$$

Example: Sinusoidal Parameter Estimation

- Instead of proceeding directly through the log-likelihood function, we note that the above function is maximized when

$$J(A, f_0, \phi) = \sum_{n=0}^{N-1} (x[n] - A\cos(2\pi f_0 n + \phi))^2$$

  is minimized.
- The minimum of this function can be found although is a nontrivial task (about 10 slides).
- We skip the derivation for now.

Sinusoidal Parameter Estimation

- The MLE of frequency f_0 is obtained by maximizing the *periodogram* over f_0:

$$\hat{f}_0 = \arg\max_f \left| \sum_{n=0}^{N-1} x[n] \exp(-j2\pi f n) \right|$$

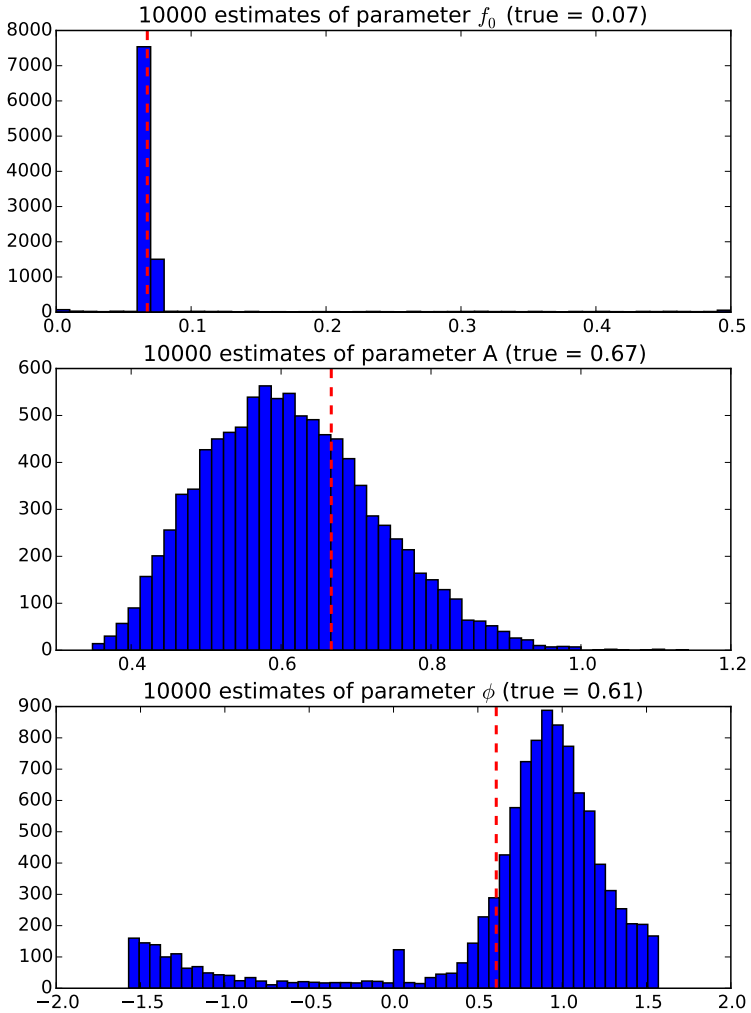
- Once $\hat{f}_0$ is available, proceed by calculating the other parameters:

$$\hat{A} = \frac{2}{N} \left| \sum_{n=0}^{N-1} x[n] \exp(-j2\pi \hat{f}_0 n) \right|$$

$$\hat{\phi} = \arctan \left(-\frac{\sum_{n=0}^{N-1} x[n] \sin 2\pi \hat{f}_0 n}{\sum_{n=0}^{N-1} x[n] \cos 2\pi \hat{f}_0 n} \right)$$

Sinusoidal Parameter Estimation—Experiments

- Four example runs of the estimation algorithm are illustrated in the figures.
- The algorithm was also tested for 10000 realizations of a sinusoid with fixed β and $N = 160$, $\sigma^2 = 1.2$.
- Note that the estimator is not unbiased.



Detection Theory

Detection theory

- A the last topic of the course, we will briefly consider detection theory.
- The methods are based on estimation theory and attempt to answer questions such as
  - Is a signal of specific model present in our time series? E.g., detection of noisy sinusoid; beep or no beep?
  - Is the transmitted pulse present at radar signal at time t?
  - Does the mean level of a signal change at time t?
  - After calculating the mean change in pixel values of subsequent frames in video, is there something moving in the scene?

Detection theory

- The area is closely related to *hypothesis testing*, which is widely used e.g., in medicine: Is the response in patients due to the new drug or due to random fluctuations?
- In our case, the hypotheses could be

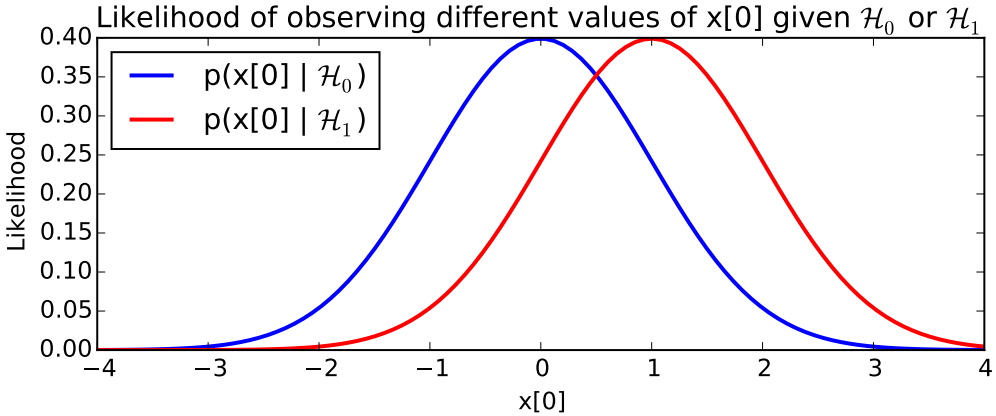
$$\mathcal{H}_1 : x[n] = A\cos(2\pi f_0 n + \phi) + w[n]$$
$$\mathcal{H}_0 : x[n] = w[n]$$

- This example corresponds to detection of noisy sinusoid.
- The hypothesis $\mathcal{H}_1$ corresponds to the case that the sinusoid is present and is called *alternative hypothesis*.
- The hypothesis $\mathcal{H}_0$ corresponds to the case that the measurements consists of noise only and is called *null hypothesis*.

Introductory Example

- *Neyman-Pearson approach* is the classical way of solving detection problems in an optimal manner.
- It relies on so called *Neyman-Pearson theorem*.
- Before stating the theorem, consider a simplistic detection problem, where we observe one sample $x[n]$ from one of two densities: $\mathcal{N}(0,1)$ or $\mathcal{N}(1,1)$.
- The task is to choose the correct density in an optimal manner.

Introductory Example



- Our hypotheses are now

$$\mathcal{H}_1 : \mu = 1,$$
$$\mathcal{H}_0 : \mu = 0.$$

Introductory Example

- An obvious approach for deciding the density would choose the one, which is higher for a particular $x[0]$.
- More specifically, study the likelihoods and choose the more likely one.
- The likelihoods are

$$\mathcal{H}_1 : p(x[0] \mid \mu = 1) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x[n]-1)^2}{2}\right).$$

$$\mathcal{H}_0 : p(x[0] \mid \mu = 0) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x[n])^2}{2}\right).$$

- Now, one should select $\mathcal{H}_1$ if $p(x[0] \mid \mu = 1) > p(x[0] \mid \mu = 0)$.

Introductory Example

- Let's state this in terms of $x[0]$:

$$
\begin{aligned}
& p(x[0] \mid \mu = 1) > p(x[0] \mid \mu = 0) \\
\Leftrightarrow\ & \frac{p(x[0] \mid \mu = 1)}{p(x[0] \mid \mu = 0)} > 1 \\
\Leftrightarrow\ & \frac{\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x[n]-1)^2}{2}\right)}{\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x[n])^2}{2}\right)} > 1 \\
\Leftrightarrow\ & \exp\left(-\frac{(x[n]-1)^2) - x[n]^2}{2}\right) > 1
\end{aligned}
$$

Introductory Example

$$\begin{aligned}
&\Leftrightarrow (x[n]^2 - (x[n]-1)^2) > 0 \\
&\Leftrightarrow 2x[n] - 1 > 0 \\
&\Leftrightarrow x[n] > \frac{1}{2}.
\end{aligned}$$

- In other words, choose $\mathcal{H}_1$ if $x[0] > 0.5$ and $\mathcal{H}_0$ if $x[0] < 0.5$.
- Studying the ratio of likelihoods on the second row of the derivation is the key.
- This ratio is called *likelihood ratio*, and comparison to a threshold γ (here $\gamma = 1$) is called *likelihood ratio test* (LRT).

Introductory Example

- Note, that it is also possible to study posterior probability ratios $p(\mathcal{H}_1 \mid \mathbf{x})/p(\mathcal{H}_0 \mid \mathbf{x})$ instead of the above likelihood ratio $p(\mathbf{x} \mid \mathcal{H}_1)/p(\mathbf{x} \mid \mathcal{H}_0)$.
- However, using Bayes rule, this *MAP test* turns out to be

$$\frac{p(\mathbf{x} \mid \mathcal{H}_1)}{p(\mathbf{x} \mid \mathcal{H}_0)} > \frac{p(\mathcal{H}_1)}{p(H_2)},$$

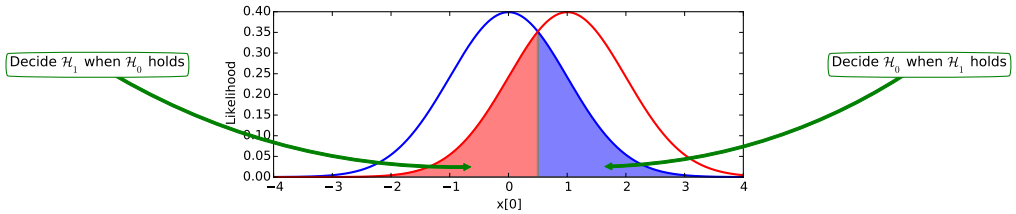
  i.e., the only effect of using posterior probability is on the threshold for the LRT.

Error Types

- It might be that the detection problem is not symmetric and some errors are more costly than others.
- For example, when detecting a disease, a missed detection is more costly than a false alarm.
- The tradeoff between misses and false alarms can be adjusted using the threshold of the LRT.

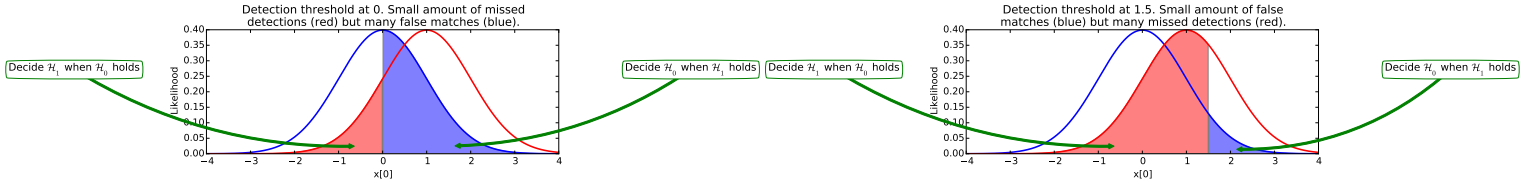
Error Types

- The below figure illustrates the probabilities of the two kinds of errors. The red area on the left corresponds to the probability of choosing $\mathcal{H}_1$ while $\mathcal{H}_0$ would hold (false match). The blue area is the probability of choosing $\mathcal{H}_0$ while $\mathcal{H}_1$ would hold (missed detection).



Error Types

- It can be seen that we can decrease either probability arbitrarily small by adjusting the detection threshold.



- Left: large threshold; small probability of false match (red), but a lot of misses (blue).
- Right: small threshold; only a few missed detections (blue), but a huge number of false matches (red).

Error Types

- Probability of false alarm for the threshold $\gamma = 1.5$ is

$$P_{FA} = P(x[0] > \gamma \mid \mu = 0) = \int_{1.5}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x[n])^2}{2}\right) dx[n] \approx 0.0668.$$

- Probability of missed detection is

$$P_M = P(x[0] > \gamma \mid \mu = 1) = \int_{-\infty}^{1.5} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x[n]-1)^2}{2}\right) dx[n] \approx 0.6915.$$

- An equivalent, but more useful term is the complement of P_M: probability of detection:

$$P_D = 1 - P_M = \int_{1.5}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x[n]-1)^2}{2}\right) dx[n] \approx 0.3085.$$

Neyman-Pearson Theorem

- Since P_{FA} and P_D depend on each other, we would like to maximize P_D subject to given maximum allowed P_{FA}. Luckily the following theorem makes this easy.
- **Neyman-Pearson Theorem:** For a fixed P_{FA}, the likelihood ratio test maximizes P_D with the decision rule

$$L(\mathbf{x}) = \frac{p(\mathbf{x}; \mathcal{H}_1)}{p(\mathbf{x}; \mathcal{H}_0)} > \gamma,$$

with threshold γ is the value for which

$$\int_{\mathbf{x}:L(\mathbf{x})>\gamma} p(\mathbf{x}; \mathcal{H}_0)\, d\mathbf{x} = P_{FA}.$$

Neyman-Pearson Theorem

- As an example, suppose we want to find the best detector for our introductory example, and we can tolerate 10% false alarms ($P_{FA} = 0.1$).
- According to the theorem, the detection rule is:

$$\text{Select } \mathcal{H}_1 \text{ if } \frac{p(x \mid \mu = 1)}{p(x \mid \mu = 0)} > \gamma$$

The only thing to find out now is the threshold γ such that

$$\int_{\gamma}^{\infty} p(x \mid \mu = 0)\, dx = 0.1.$$

This can be done with Matlab function `icdf`, which solves the inverse cumulative distribution function.

Neyman-Pearson Theorem

- Unfortunately `icdf` solves the γ for which

$$\int_{-\infty}^{\gamma} p(x \mid \mu = 0)\, dx = 0.1 \text{ instead of } \int_{\gamma}^{\infty} p(x \mid \mu = 0)\, dx = 0.1.$$

  Thus, we have to use the function like this:
  `icdf('norm', 1-0.1, 0, 1)`, which gives $\gamma \approx 1.2816$.
- Similarly, we can also calculate the P_D with this threshold:

$$P_D = \int_{1.2816}^{\infty} p(x \mid \mu = 1)\, dx \approx 0.3891.$$

Detector for a known waveform

- The NP approach applies to all cases where likelihoods are available.
- An important special case is that of a known waveform $s[n]$ embedded in WGN sequence $w[n]$:

$$\mathcal{H}_1 : x[n] = s[n] + w[n]$$
$$\mathcal{H}_0 : x[n] = w[n].$$

- An example of a case where the waveform is known could be detection of radar signals, where a pulse $s[n]$ transmitted by us is reflected back after some propagation time.

Detector for a known waveform

- For this case the likelihoods are

$$p(\mathbf{x} \mid \mathcal{H}_1) = \prod_{n=0}^{N-1} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x[n]-s[n])^2}{2\sigma^2}\right),$$

$$p(\mathbf{x} \mid \mathcal{H}_0) = \prod_{n=0}^{N-1} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x[n])^2}{2\sigma^2}\right).$$

- The likelihood ratio test is easily obtained as

$$\frac{p(\mathbf{x} \mid \mathcal{H}_1)}{p(\mathbf{x} \mid \mathcal{H}_0)} = \exp\left[-\frac{1}{2\sigma^2}\left(\sum_{n=0}^{N-1}(x[n]-s[n])^2 - \sum_{n=0}^{N-1}(x[n])^2\right)\right] > \gamma.$$

Detector for a known waveform

- This simplifies by taking the logarithm from both sides:

$$-\frac{1}{2\sigma^2}\left(\sum_{n=0}^{N-1}(x[n]-s[n])^2-\sum_{n=0}^{N-1}(x[n])^2\right)>\ln\gamma.$$

- This further simplifies into

$$\frac{1}{\sigma^2}\sum_{n=0}^{N-1}x[n]s[n]-\frac{1}{2\sigma^2}\sum_{n=0}^{N-1}(s[n])^2>\ln\gamma.$$

Detector for a known waveform

- Since $s[n]$ is a known waveform (= constant), we can simplify the procedure by moving it to the right hand side and combining it with the threshold:

$$\sum_{n=0}^{N-1} x[n]s[n] > \sigma^2 \ln \gamma + \frac{1}{2} \sum_{n=0}^{N-1} (s[n])^2.$$

We can equivalently call the right hand side as our threshold (say γ') to get the final decision rule

$$\sum_{n=0}^{N-1} x[n]s[n] > \gamma'.$$

Examples

- This leads into some rather obvious results.
- The detector for a known DC level in WGN is

$$\sum_{n=0}^{N-1} x[n]A > \gamma \Rightarrow A \sum_{n=0}^{N-1} x[n] > \gamma$$

Equally well we can set a new threshold and call it $\gamma' = \gamma/(AN)$. This way the detection rule becomes: $\bar{x} > \gamma'$. Note that a negative A would invert the inequality.

Examples

- The detector for a sinusoid in WGN is

$$\sum_{n=0}^{N-1} x[n] A\cos(2\pi f_0 n + \phi) > \gamma \Rightarrow A\sum_{n=0}^{N-1} x[n]\cos(2\pi f_0 n + \phi) > \gamma.$$

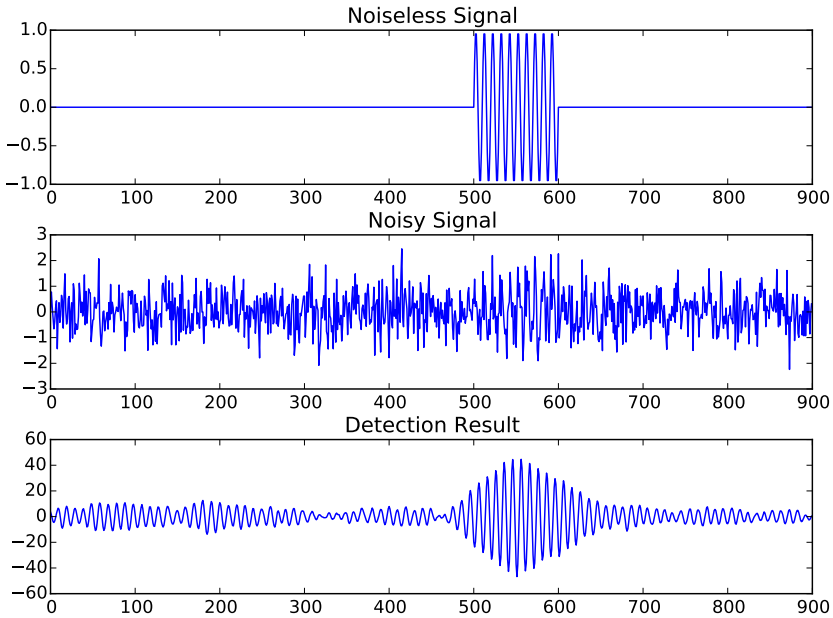
- Again we can divide by A to get

$$\sum_{n=0}^{N-1} x[n]\cos(2\pi f_0 n + \phi) > \gamma'.$$

- In other words, we check the correlation with the sinusoid. Note that the amplitude A does not affect our statistic, only the threshold which is anyway selected according to the fixed P_{FA} rate.

Examples

- As an example, the below picture shows the detection process with $\sigma = 0.5$.



Detection of random signals

- The problem with the previous approach was that the model was too restrictive; the results depend on how well the phases match.
- The model can be relaxed by considering *random signals*, whose exact form is unknown, but the correlation structure is known. Since the correlation captures the frequency (but not the phase), this is exactly what we want.
- In general, the detection of a random signal can be formulated as follows.

Detection of random signals

- Suppose $\mathbf{s} \sim \mathcal{N}(0, \mathbf{C}_s)$ and $\mathbf{w} \sim \mathcal{N}(0, \sigma^2\mathbf{I})$. Then the detection problem is a hypothesis test

$$\mathcal{H}_0 : \mathbf{x} \sim \mathcal{N}(0, \sigma^2\mathbf{I})$$
$$\mathcal{H}_1 : \mathbf{x} \sim \mathcal{N}(0, \mathbf{C}_s + \sigma^2\mathbf{I})$$

- It can be shown (see Kay-2, p. 145), that the decision rule becomes

$$\text{Decide } \mathcal{H}_1, \text{ if } \mathbf{x}^T\hat{\mathbf{s}} > \gamma,$$

where

$$\hat{\mathbf{s}} = \mathbf{C}_s(\mathbf{C}_s + \sigma^2\mathbf{I})^{-1}\mathbf{x}.$$

Detection of random signals

- The term $\hat{\mathbf{s}}$ is in fact the estimate of the signal; more specifically, the linear Bayesian MMSE estimator, which assumes linearity for the estimator (similar to BLUE).
- A particular special case of a random signal is the Bayesian linear model.
- The Bayesian linear model assumes linearity $\mathbf{x} = \mathbf{H}\boldsymbol{\beta} + \mathbf{w}$ together with a prior for the parameters, such as $\boldsymbol{\beta} \sim \mathcal{N}(0, \sigma^2\mathbf{I})$
- Consider the following detection problem:

$$\mathcal{H}_0 : \mathbf{x} = \mathbf{w}$$
$$\mathcal{H}_1 : \mathbf{x} = \mathbf{H}\boldsymbol{\beta} + \mathbf{w}$$

Detection of random signals

- Within the earlier random signal framework, this is written as

$$\mathcal{H}_0 : \mathbf{x} \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$$
$$\mathcal{H}_1 : \mathbf{x} \sim \mathcal{N}(0, \mathbf{C}_s + \sigma^2 \mathbf{I})$$

  with $\mathbf{C}_s = \mathbf{H}\mathbf{C}_{\boldsymbol{\beta}}\mathbf{H}^T$.
- The assumption $\mathbf{s} \sim \mathcal{N}(0, \mathbf{H}\mathbf{C}_{\boldsymbol{\beta}}\mathbf{H}^T)$ states that the exact form of the signal is unknown, and we only know its covariance structure.

Detection of random signals

- This is helpful in the sinusoidal detection problem: we are not interested in the phase (included by the exact formulation $x[n] = A\cos(2\pi f_0 n + \phi)$), but only in the frequency (as described by the covariance matrix $\mathbf{H}\mathbf{C}_\beta\mathbf{H}^T$).
- Thus, the decision rule becomes:

$$\text{Decide } \mathcal{H}_1, \text{ if } \mathbf{x}^T\hat{\mathbf{s}} > \gamma,$$

where

$$\begin{aligned}\hat{\mathbf{s}} &= \mathbf{C}_s(\mathbf{C}_s + \sigma^2\mathbf{I})^{-1}\mathbf{x} \\ &= \mathbf{H}\mathbf{C}_\beta\mathbf{H}^T(\mathbf{H}\mathbf{C}_\beta\mathbf{H}^T + \sigma^2\mathbf{I})^{-1}\mathbf{x}\end{aligned}$$

Detection of random signals

- Luckily the decision rule simplifies quite a lot by noticing that the last part is the MMSE estimate of $\boldsymbol{\beta}$:

$$\begin{aligned}\mathbf{x}^T\hat{\mathbf{s}} &= \mathbf{x}^T\mathbf{H}\mathbf{C}_{\boldsymbol{\beta}}\mathbf{H}^T(\mathbf{H}\mathbf{C}_{\boldsymbol{\beta}}\mathbf{H}^T + \sigma^2\mathbf{I})^{-1}\mathbf{x} \\ &= \mathbf{x}^T\mathbf{H}\hat{\boldsymbol{\beta}}.\end{aligned}$$

- An example of applying the linear model is in Kay: Statistical Signal Processing, vol. 2; Detection Theory, pages 155-158.
- In the example, a Rayleigh fading sinusoid is studied, which has an unknown amplitude A and phase term ϕ. Only the frequency f_0 is assumed to be known.

Detection of random signals

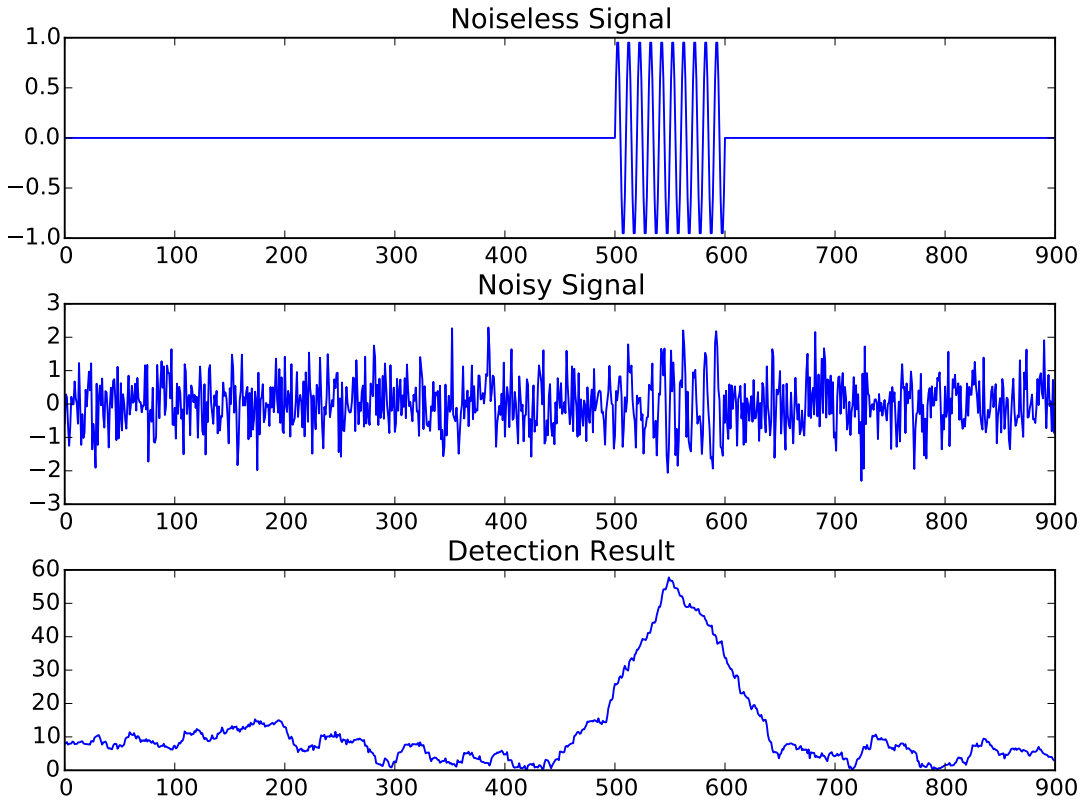
- This can be manipulated into a linear model form with two unknowns corresponding to A and ϕ.
- The final result is the decision rule:

$$\left| \sum_{n=0}^{N-1} x[n] \exp(-2\pi i f_0 n) \right| > \gamma.$$

- As an example, the below picture shows the detection process with $\sigma = 0.5$.
- Note the simplicity of Matlab implementation:

```
h = exp(-2*pi*sqrt(-1)*f0*n);
y = abs(conv(h,x));
```

Detection of random signals



Receiver Operating Characteristics

- A usual way of illustrating the detector performance is the *Receiver Operating Characteristics* curve (ROC curve).
- This describes the relationship between P_{FA} and P_D for all possible values of the threshold γ.
- The functional relationship between P_{FA} and P_D depends on the problem (and the selected detector, although we have proven that LRT is optimal).

Receiver Operating Characteristics

- For example, in the DC level example,

$$P_D(\gamma) = \int_{\gamma}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x-1)^2}{2}\right) dx$$

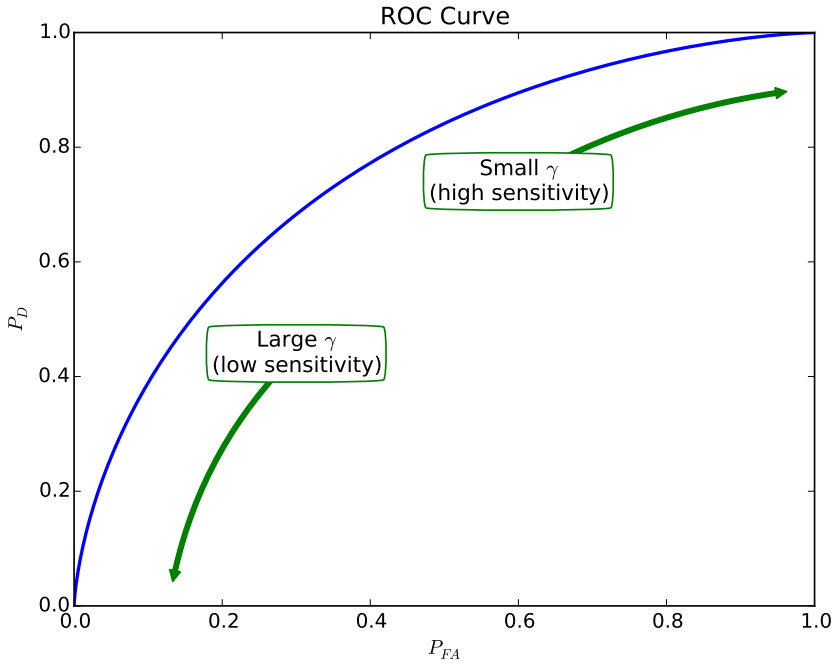
$$P_{FA}(\gamma) = \int_{\gamma}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx$$

- It is easy to see the relationship:

$$P_D(\gamma) = \int_{\gamma-1}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx = P_{FA}(\gamma - 1).$$

Receiver Operating Characteristics

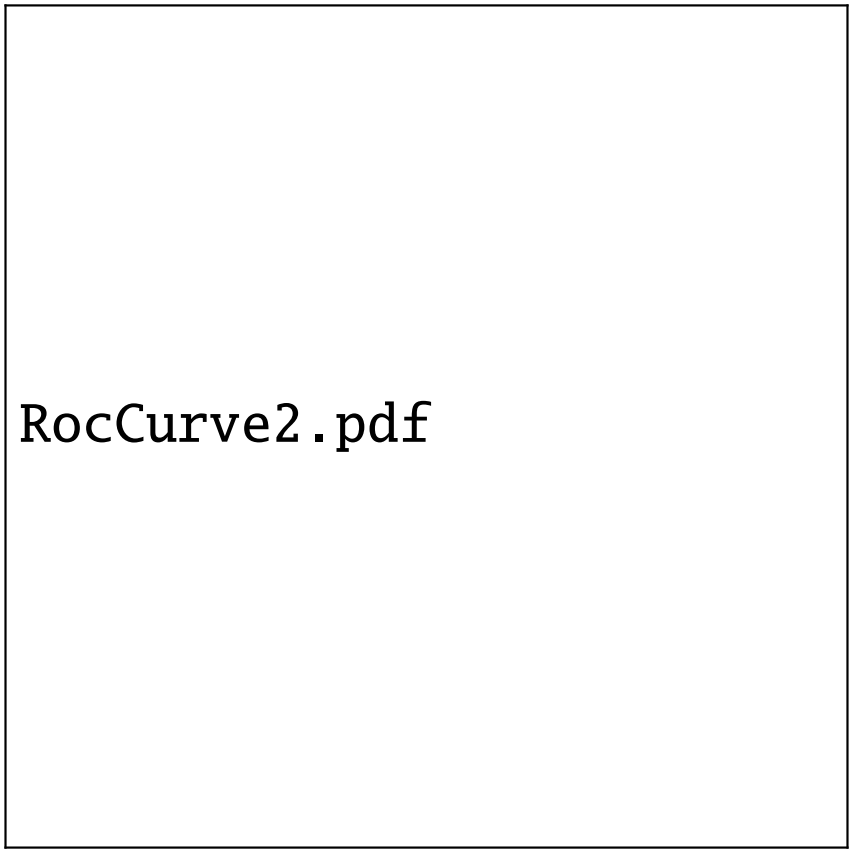
- Plotting the ROC curve for all γ results in the following curve.



Receiver Operating Characteristics

- The higher the ROC curve, the better the performance.
- A random guess has diagonal ROC curve.
- In the DC level case, the performance increases if the noise variance σ^2 decreases. Below are the ROC plots for various values of σ^2.

Receiver Operating Characteristics



Receiver Operating Characteristics

- This gives rise to a widely used measure for detector performance: the *Area Under (ROC) Curve*, or AUC criterion.

Composite hypothesis testing

- In the previous examples the parameter values specified the distribution completely; e.g., either $A = 1$ or $A = 0$.
- Such cases are called *simple hypotheses testing*.
- Often we can't specify exactly the parameters for either case, but instead a range of values for each case.
- An example could be our DC model $x[n] = A + w[n]$ with

$$\mathcal{H}_1 : A \neq 0$$
$$\mathcal{H}_0 : A = 0$$

Composite hypothesis testing

- The question can be posed in a probabilistic manner as follows:

  What is the probability of observing $x[n]$ if $\mathcal{H}_0$ would hold?

- If the probability is small (e.g., all $x[n] \in [0.5, 1.5]$, and let's say the probability of observing $x[n]$ under $\mathcal{H}_0$ is 1 %), then we can conclude that the null hypothesis can be *rejected* with 99% confidence.

An example

- As an example, consider detecting a biased coin in a coin tossing experiment.
- If we get 19 heads out of 20 tosses, it seems rather likely that the coin is biased.
- How to pose the question mathematically?
- Now the hypotheses is

$$\mathcal{H}_1 : \text{coin is biased: } p \neq 0.5$$
$$\mathcal{H}_0 : \text{coin is unbiased: } p = 0.5,$$

where p denotes the probability of a head for our coin.

An example

- Additionally, let's say, we want 99% confidence for the test.
- Thus, we can state the hypothesis test as: "what is the probability of observing at least 19 heads assuming $p = 0.5$?"
- This is given by the binomial distribution

$$\underbrace{\binom{20}{19} 0.5^{19} \cdot 0.5^{1}}_{\text{19 heads}} \underbrace{+}_{\text{or}} \underbrace{0.5^{20}}_{\text{20 heads}} \approx 0.00002.$$

- Since $0.00002 < 1\%$, we can reject the null hypothesis and the coin is biased.

An example

- Actually, the 99% confidence was a bit loose in this case.
- We could have set a 99.98% confidence requirement and still reject the null hypothesis.
- The upper limit for the confidence (here 99.98%) is widely used and called the *p-value*.
- More specifically,

  The p-value is the probability of obtaining a test statistic at least as extreme as the one that was actually observed, assuming that the null hypothesis is true.

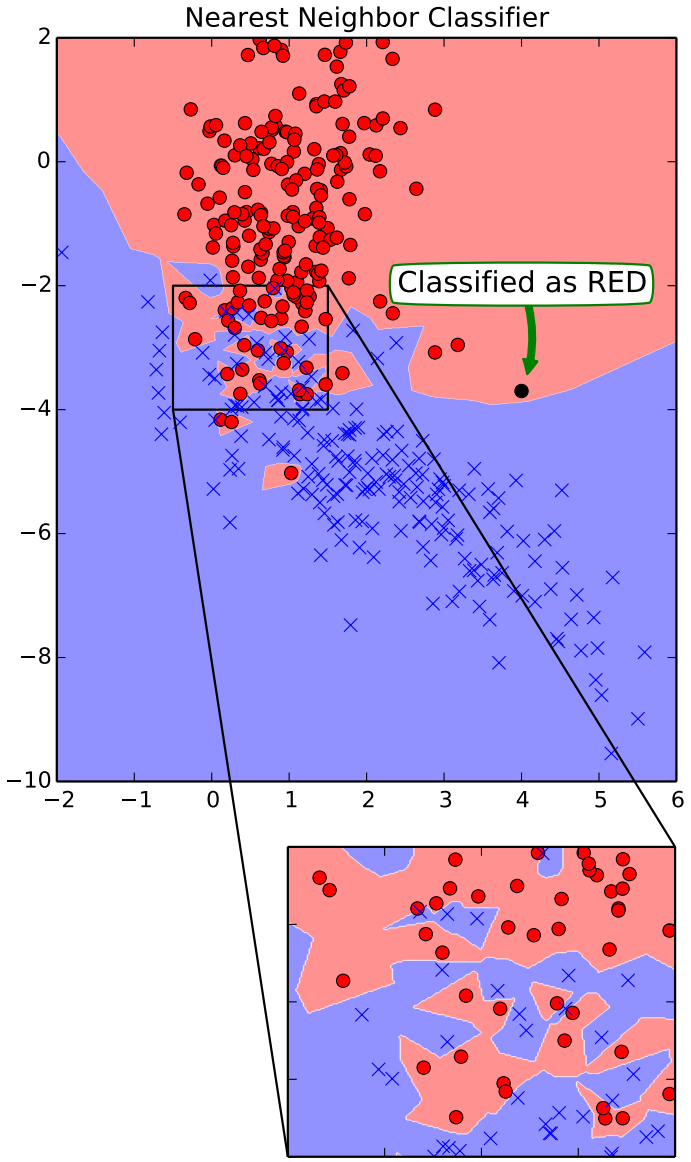
Classification

Different Classifiers

- In the following we will take a non-mathematical overview of the following widely used classifiers.
  - Nearest Neighbor classifier
  - Linear classifiers (with linear boundary)
  - The support vector machine (with nonlinear boundary)
  - Ensemble classifiers: Random Forest

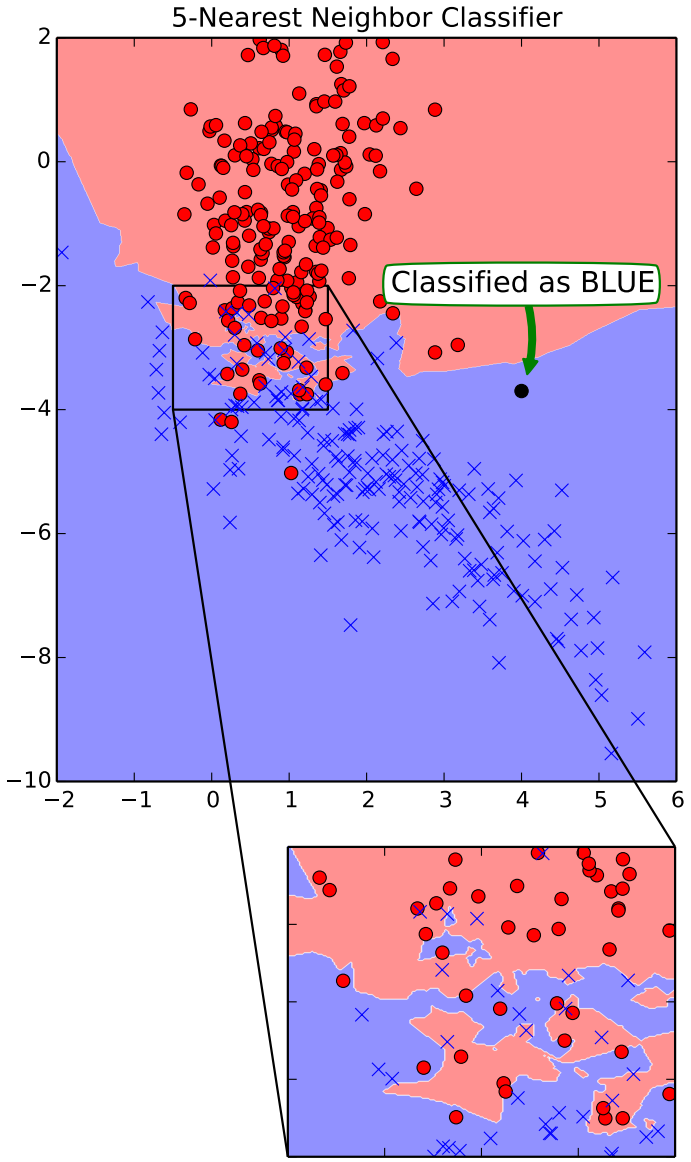
Nearest Neighbor Classifier

- Probably the most natural idea is to forget the boundary approach and just see what kind of samples are nearby.
- This is the idea behind the **Nearest neighbor** classifier: Just copy the class label of the most similar training sample to the unknown test sample.



K-Nearest Neighbor Classifier

- The problem with Nearest Neighbor is its fragility to changes in the training data.
- The classification boundary may change a lot by moving only one training sample.
- The robustness can be increased by taking a majority vote of more nearby samples.
- The **K-Nearest neighbor** classifier selects the most frequent class label among the *K* nearest training samples.



Other Problems with the Nearest Neighbor

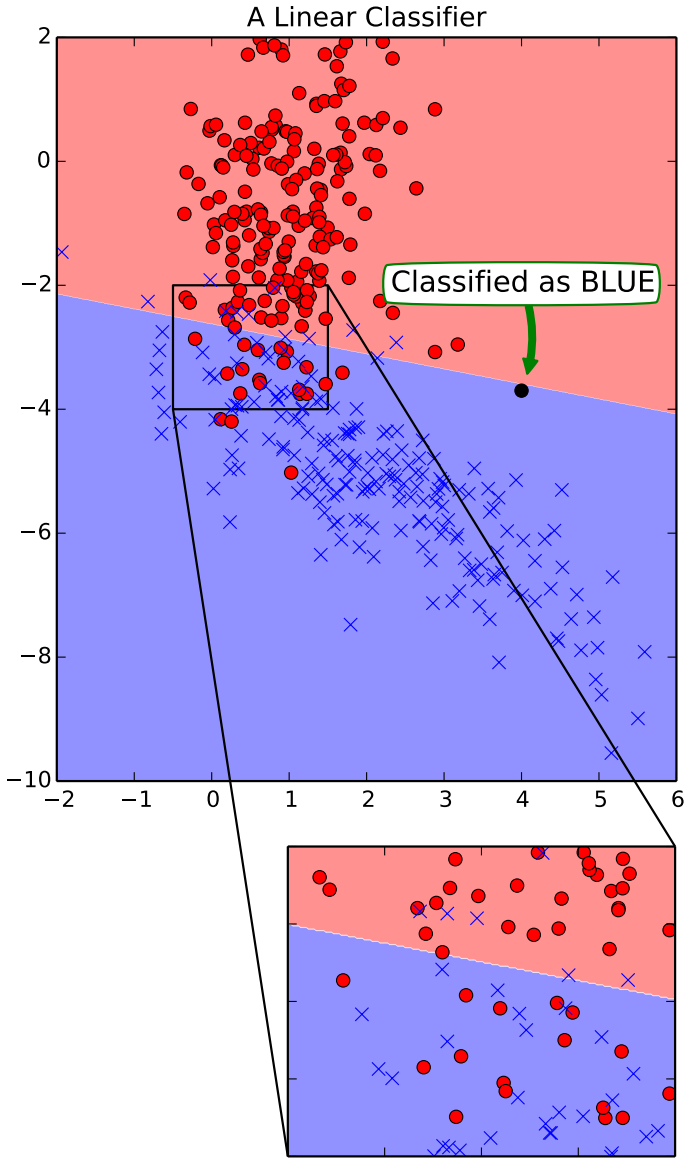
- Nearest neighbor is prone to overlearning: Especially the 1-NN forms local regions around every isolated data point.
- It is highly unlikely that these represent the general trend of the whole population.
- Moreover, the training step extremely fast while the classification step becomes extremely slow (consider training data with billion high-dimensional samples).
- Therefore, more compact representations are preferred: Training time is usually not critical while execution time is.

Linear Classifiers

- A linear classifier learns a linear decision boundary between classes.
- In mathematical terms, the classification rule can be written as

$$F(\mathbf{x}) = \begin{cases} \text{Class 1}, & \text{if } \mathbf{w}^T\mathbf{x} < b \\ \text{Class 2}, & \text{if } \mathbf{w}^T\mathbf{x} \geqslant b \end{cases}$$

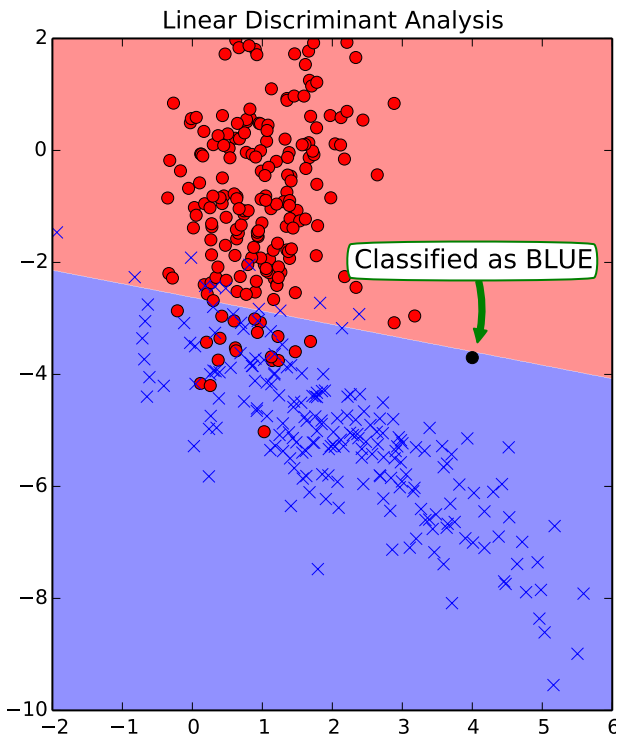
  where the weights $\mathbf{w}$ are learned from the data.
- The expression $\mathbf{w}^T\mathbf{x} = \sum_k w_k x_k$ essentially transforms the multidimensional data $\mathbf{x}$ to a real number, which is then compared to a threshold b.

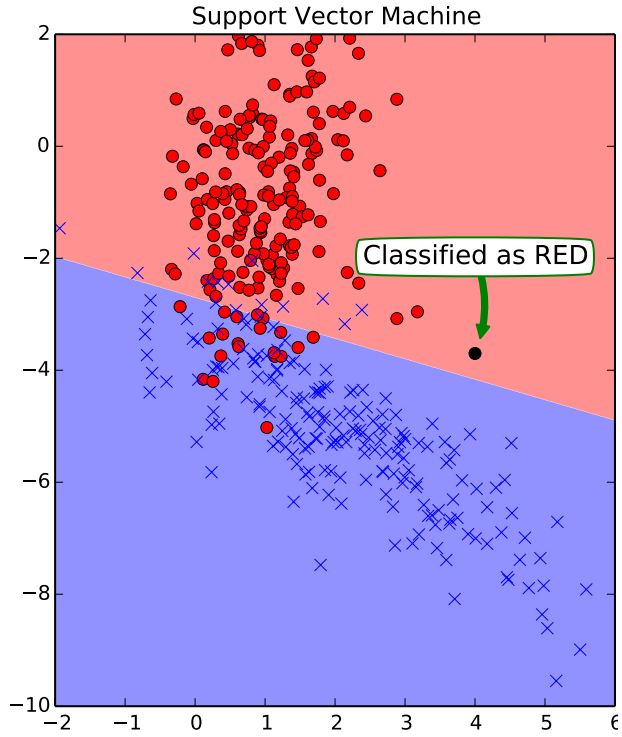


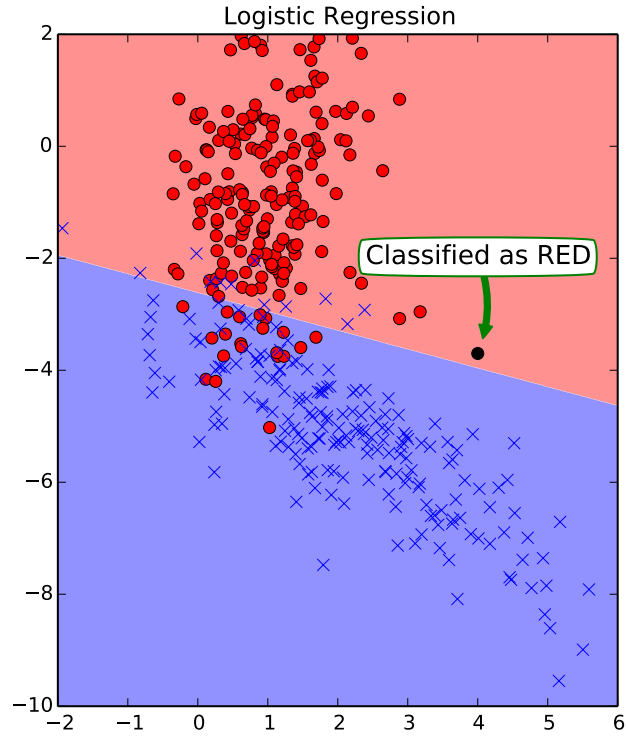
Flavors of Linear Classifiers

There exists many algorithms for learning the weights **w**, including:

- *Linear Discriminant Analysis (LDA)*
- *Support Vector Machine (SVM)*
- *Logistic Regression (LR)*



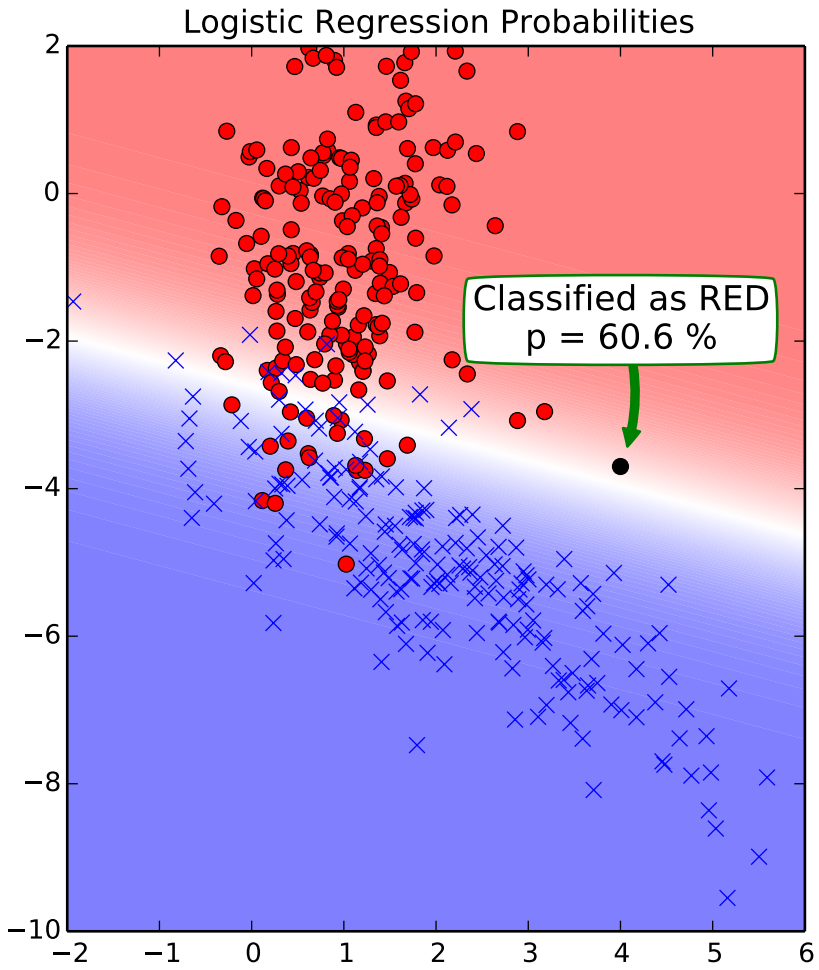




Details

- The LDA:
  - The oldest of the three: Fisher, 1935.
  - "Find the projection that maximizes class separation", *i.e.*, pull the classes as far from each other as possible.
  - Closed form solution, fast to train.
- The SVM:
  - Vapnik and Chervonenkis, 1963.
  - "Maximize the margin between classes".
  - Slowest of the three, performs well with sparse high-dimensional data.
- Logistic Regression (*a.k.a. Generalized Linear Model*):
  - History traces back to 1700's, but proper formulation and efficient training algorithm by Nelder and Wedderburn in 1972.
  - Statistical algorithm: "Maximize the likelihood of the data".
  - Outputs also class probability. Has been extended to automatic feature selection.

The Class Probabilities of Logistic Regression

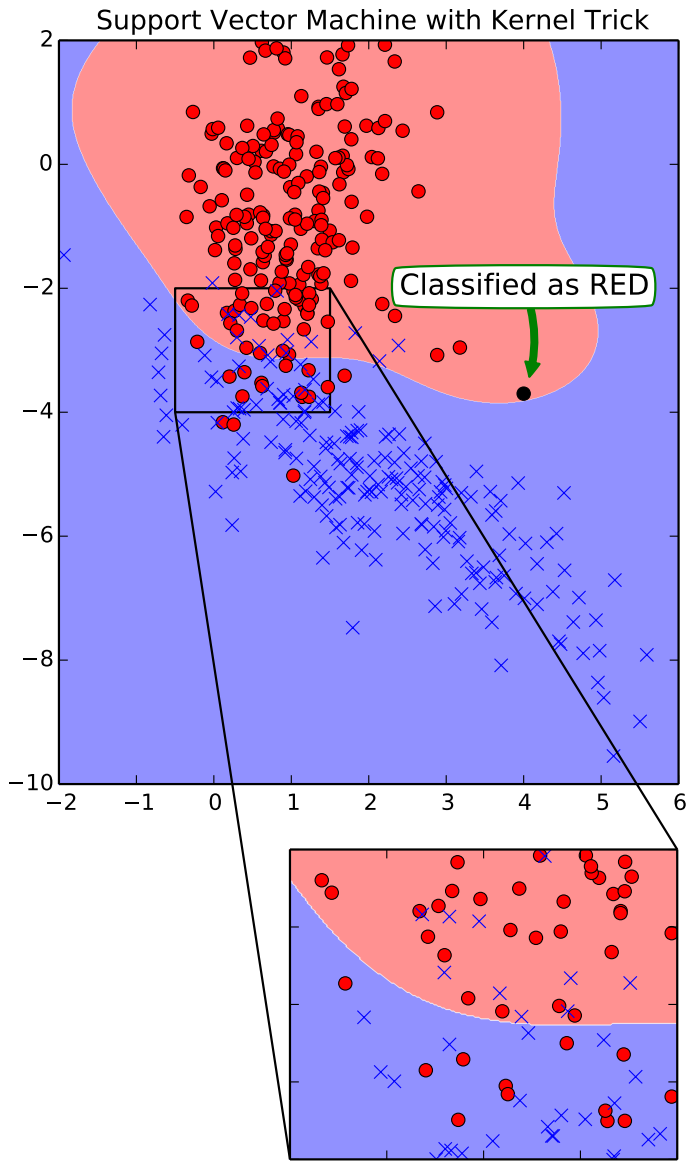


Nonlinear SVM

- The Support Vector Machine can be extended to nonlinear class boundaries.
- This is based on the *kernel trick* that implicitly maps the data to a higher dimensional space.
- The same effect is achieved by appending "fake" dimensions to the feature vector; *e.g.*,

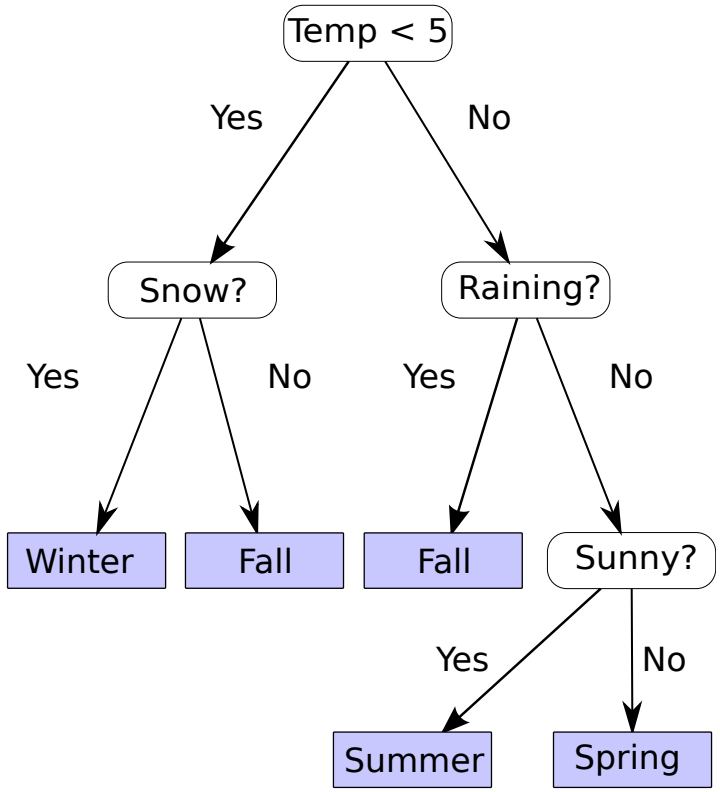
$$\mathbf{x} = (x_1, x_2) \quad \Longrightarrow \quad \mathbf{x}' = (x_1, x_2, x_1^2, x_2^2)$$

  The classifier is then trained in 4-D.
- The kernel trick is a lot faster than the explicit map, and allows mappings to even ∞-dimensional spaces!



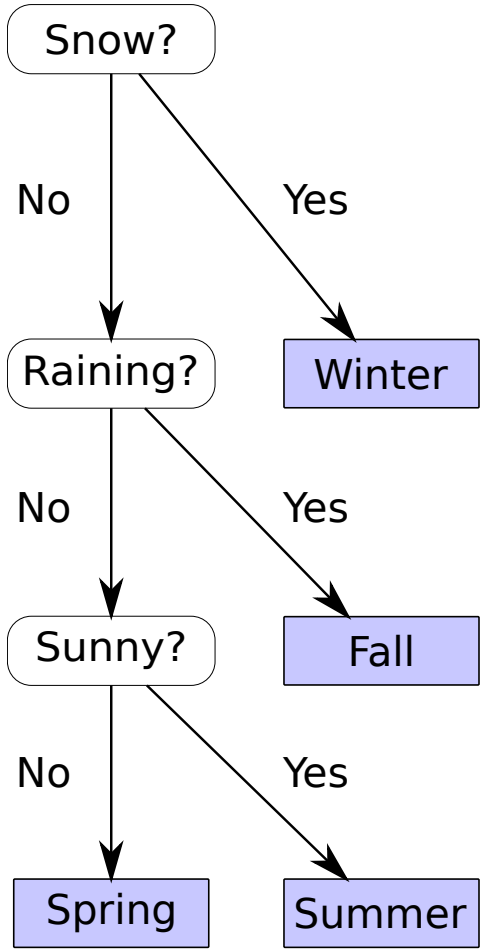
Random Forest

- Random forest (RF) is popular tree based classifier, proposed by Breiman in 1993.
- The RF is based on a collection of *decision trees*.
- A decision tree is a straightforward if-then-else-like diagram that combines individual attributes into a decision.
- Decision trees are easily trained to learn the data.
- For example, the tree on the right predicts the time of year from the following measured attributes: {Temperature, Snow, Rain, Sunny}.



Random Forest

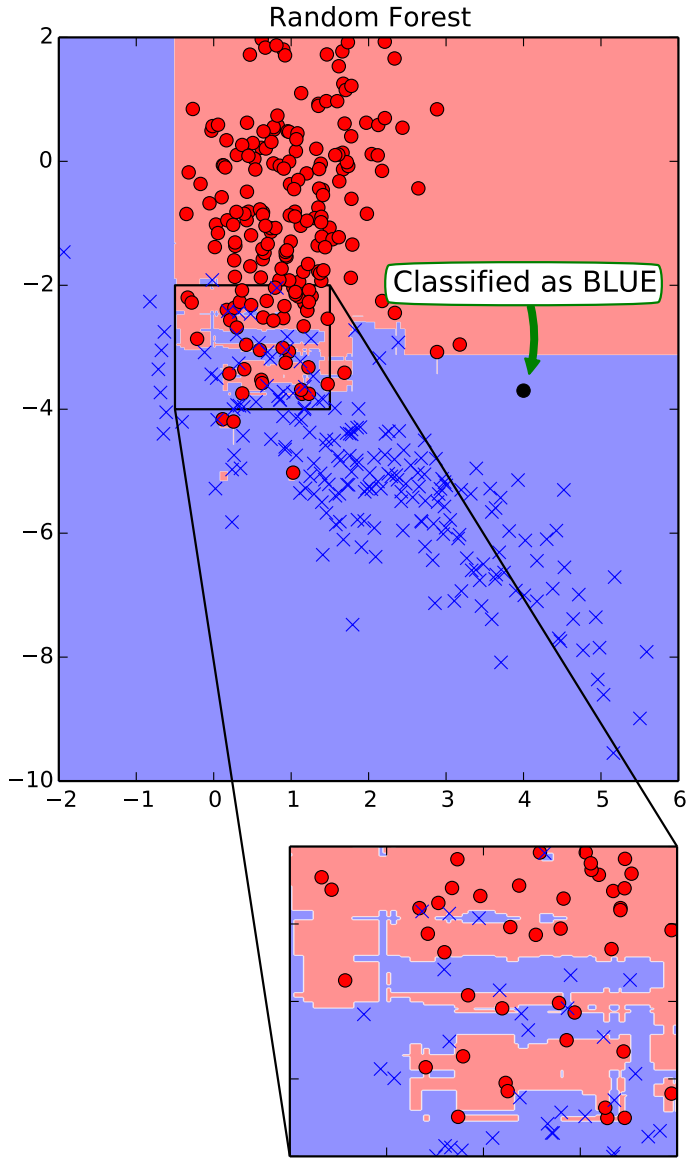
- The problem of decision trees is that they *overlearn* the data, *i.e.*, the training data is memorized with a poor ability to generalize.
- With no restrictions, the DT will have one root-leaf-path per sample, which is essentially same as nearest neighbor.
- RF avoids this by training many "imperfect" trees.
- Each tree is trained with a subset of samples and a subset of features, *i.e.*, some of the attributes are hidden from the training.



Decision Tree without the *Temperature* attribute.

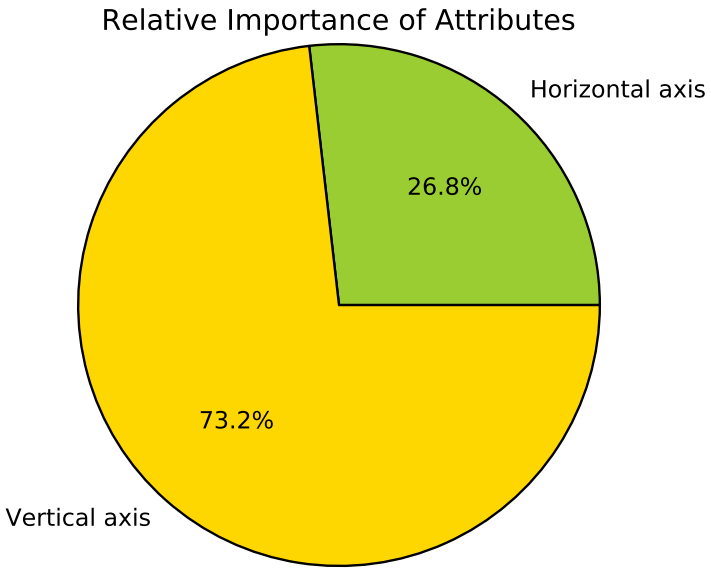
Random Forest

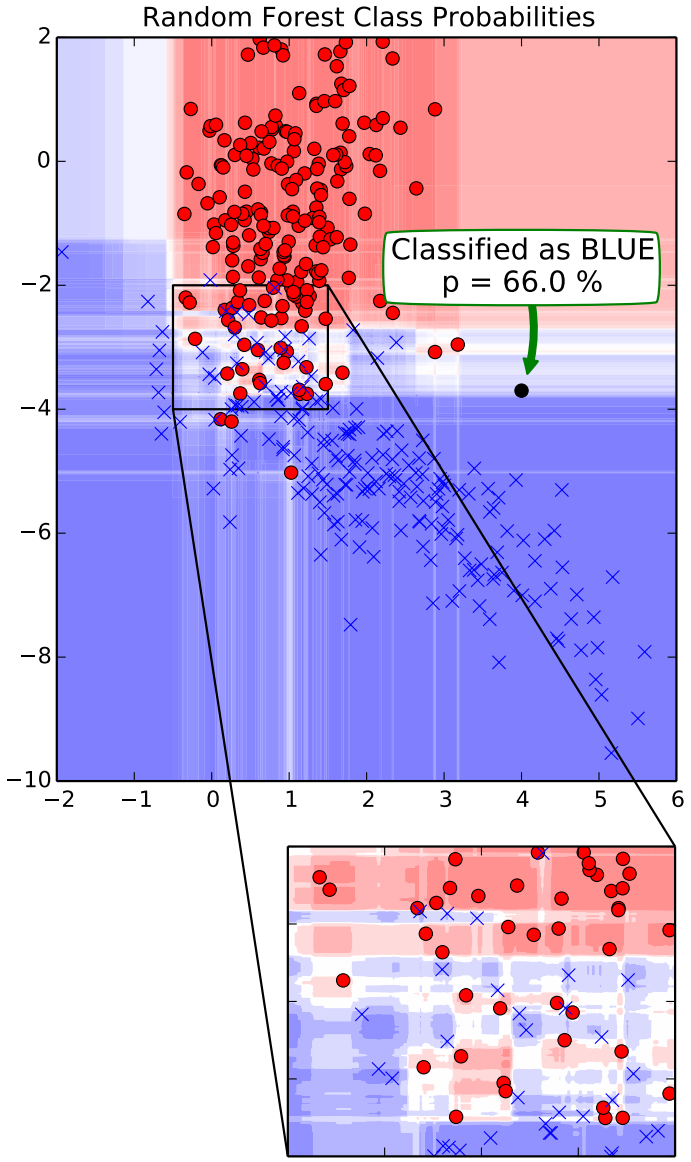
- After training many trees each with different samples and features, the RF predicts the class by taking the majority vote: what is the most frequent label.
- The number of trees varies from a few dozen to few thousand—default in Python is 10.



Random Forest

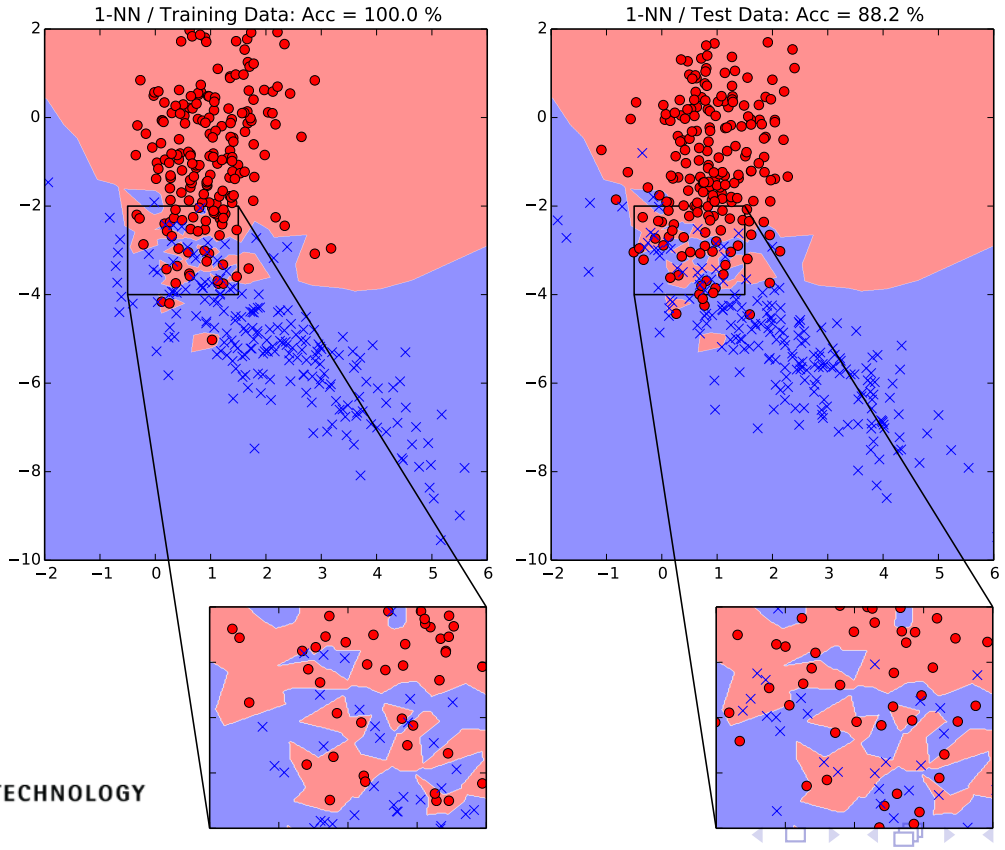
- The collection of trees gives a natural way of estimating class probabilities: Just use the proportion of trees voting for each class.
- RF's also includes a method for assessing feature importances.





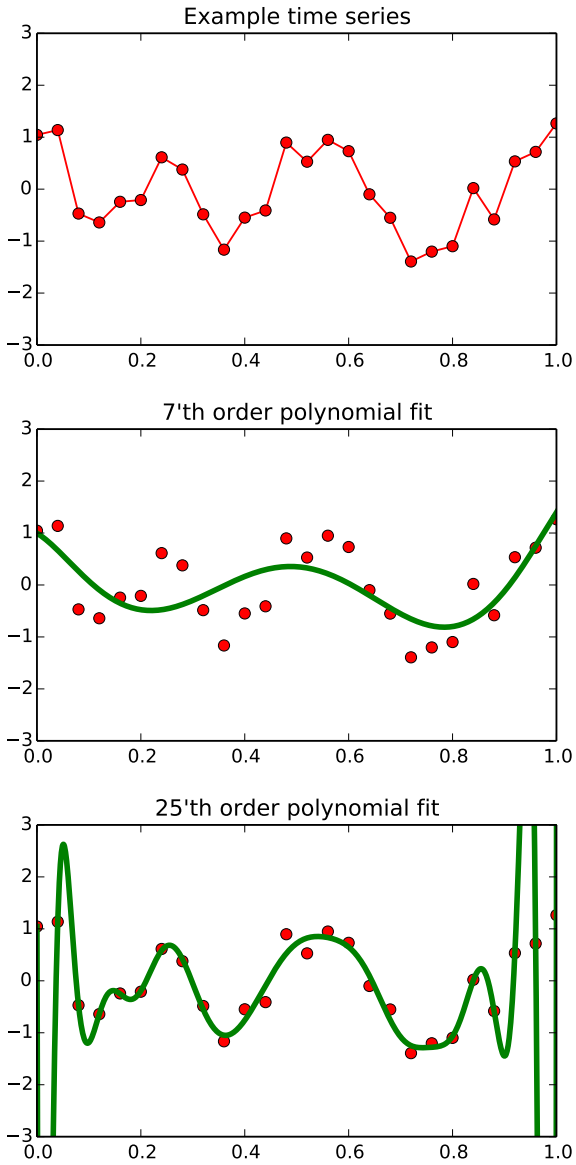
Generalization

- The important thing is how well the classifier works with unseen data.
- An overfitted classifier memorizes the training data and does not generalize.



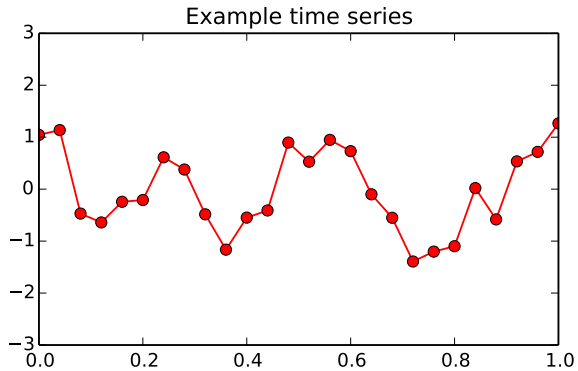
Overfitting

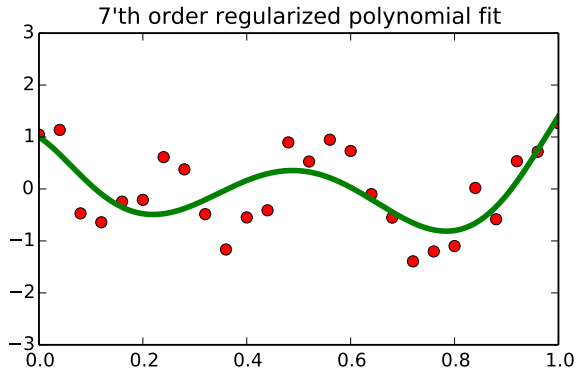
- Generalization is also related to *overfitting*.
- On the right, a polynomial model is fitted to a time series to minimize the error between the samples (red) and the model (green).
- As the order of the polynomial increases, the model starts to follow the data very faithfully.
- Low-order models do not have enough expression power.
- High-order models are over-fitting to noise and become "unstable" with crazy values near the boundaries.

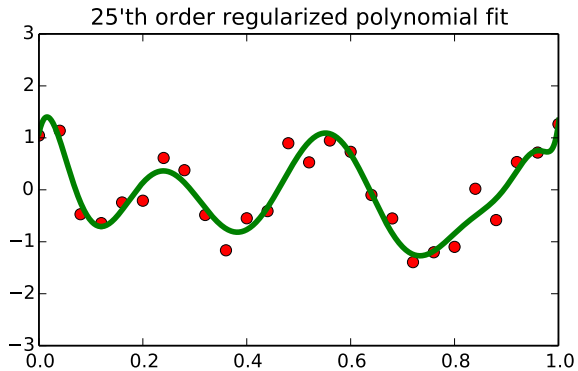


Regularization

- Regularization adds a penalty term to the fitting error.
- The model is encouraged to use small coefficients.
- Large coefficients are expensive, so the model can afford to fit only to the major trends.
- On the right, the high order model has a good expression power, but does still not follow the noise patterns.

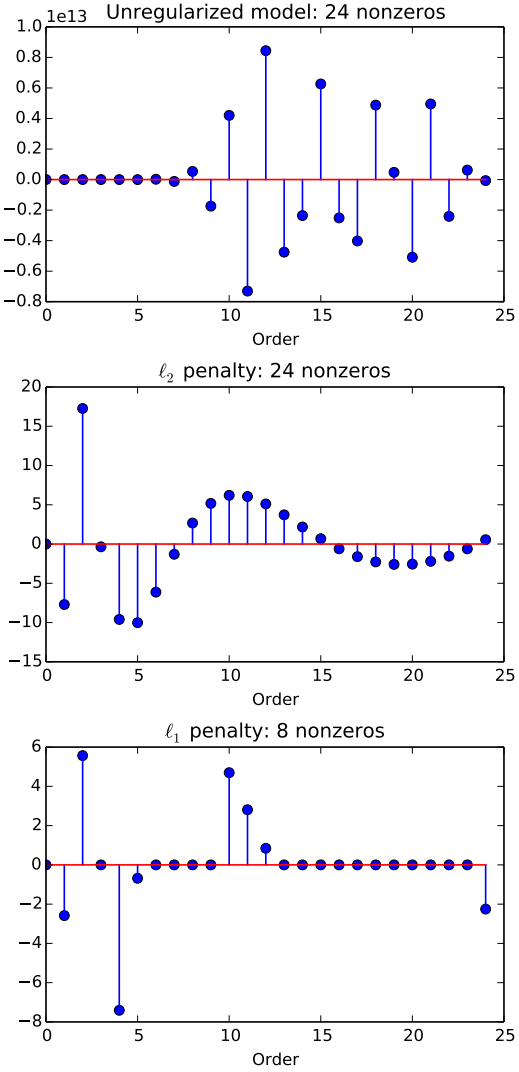






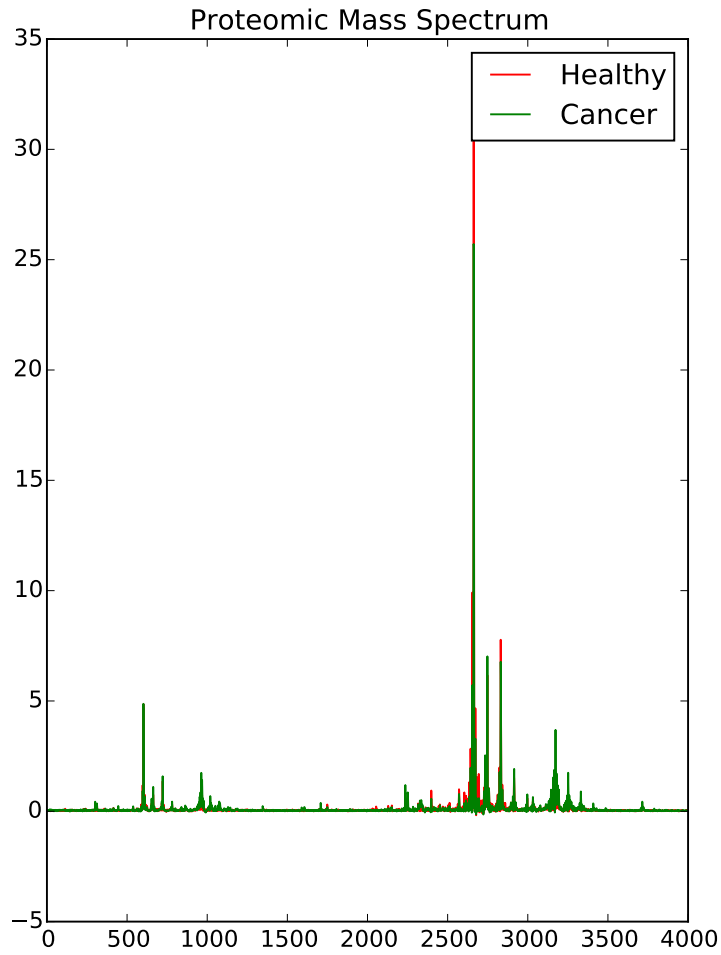
Sparsity

- Regularization also enables the design of *sparse* classifiers.
- In this case, the penalty term is designed such that it favors **zero** coefficients.
- A zero coefficient for a linear operator is equivalent to discarding the corresponding feature altogether.
- The plots illustrate the model coefficients without regularization, with traditional regularization and sparse regularization.
- The importance of sparsity is twofold: The model can be used for feature selection, but often also generalizes better.



Example: Classifier Design for Ovarian Cancer Detection

- We will study an example of classifying proteomic fingerprints (mass-spectra measured from ovarian cancer patients and healthy controls.[a]
- 121 cancer patients and 95 healthy controls.
- Mass spectra consists of 4000 features (already downsampled at the preprocessing step).



[a] Conrads, Thomas P., et al. "High-resolution serum proteomic featu for ovarian cancer detection." *Endocrine-Related Cancer* (2004).

Reading the Data

- The data is in csv (comma separated values) format, where each line contains 4000 MS measurements for one person.
- The attached code reads each line and stores the 4000 measurements into a list.
- The class labels are read in a similar manner.
- Alternatively, we could have read using numpy's `loadtxt` function, or the Pandas module; http://pandas.pydata.org/

Code:

```
13 # First read the observations to X
14
15 obs_file = "observations.csv"
16 X = []
17 f = open(obs_file, "r")
18
19 for line in f:
20
21     # Separate numbers in the line:
22     items = line.split(",")
23
24     # Transform each from string to float:
25     numbers = [float(x) for x in items]
26
27     # Append to X
28     X.append(numbers)
29
30 f.close()
31 print "Data shape:", np.shape(X)
```

Result:

```
Data shape: (216, 4000)
Labels shape: (216,)
```

Training a Classifier

- The attached code trains and applies a Logistic Regression classifier for the ovarian cancer problem, with essentially 3 lines of code:

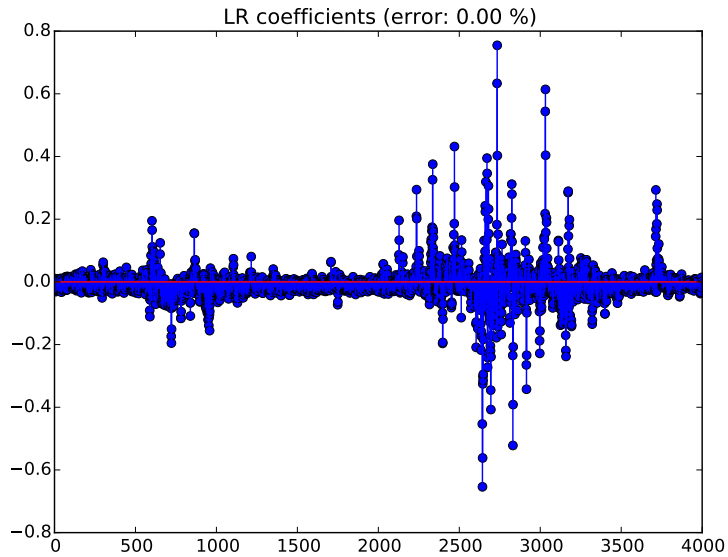
```
from sklearn.linear_model \
     import LogisticRegression
clf = LogisticRegression()
clf.fit(X, y)
prediction = clf.predict(X)
```

- Note that the above prediction is 100 % accurate; an unrealistic result due to testing on the (rather small) training set.

Code:

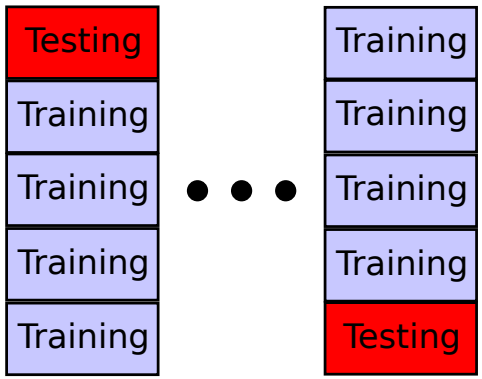
```
62 # Train an LR classifier:
63 clf = LogisticRegression()
64 clf.fit(X, y)
65
66 # Plot the coefficients:
67 plt.figure()
68 plt.stem(clf.coef_.ravel())
69 plt.savefig("../images/LR_coef.pdf")
70
71 # See how good we are:
72 prediction = clf.predict(X)
73 error = np.mean(np.abs(prediction - y)) / y.shape[0]
74 plt.title("Prediction error: %.2f %%" % (100.0 * error))
```

Result:



Cross-validation

- The generalization ability of a classifier needs to be tested with **unseen** samples.
- In practice this can be done by splitting the data into separate training and test sets.
- A standard approach is to use K-fold cross-validation:
  - Split the training data to K parts (called folds)
  - Use each fold for testing exactly once and train with the other folds
  - The error estimate is the mean of the K estimates



Cross-validation in sklearn

- Cross-validation is extremely elegant in sklearn.
- Essentially 1 line of code is required:

```
from sklearn.cross_validation \
     import cross_val_score
scores = cross_val_score(clf, X, y)
```

- The cross-validator receives the classifier and the data and returns a list of scores for each test fold.
- The function can receive additional parameters that modify the number of folds, possible stratification (balancing the classes in folds), etc.

Code:

```
77 # Cross-validate to estimate accuracy
78
79 scores = cross_val_score(clf,
80                          X,
81                          y,
82                          cv = 10,
83                          n_jobs = 2)
84
85 for fold in range(10):
86     print "Fold %d score: %.2f %%" % \
87           (fold, 100*scores[fold])
88
89 print "-" * 10
90 print "CV accuracy: %.2f %%" % \
91       (100*np.mean(scores))
```

Result:

```
Fold 0 score: 95.65 %
Fold 1 score: 100.00 %
Fold 2 score: 100.00 %
Fold 3 score: 100.00 %
Fold 4 score: 86.36 %
Fold 5 score: 100.00 %
Fold 6 score: 100.00 %
Fold 7 score: 100.00 %
Fold 8 score: 100.00 %
Fold 9 score: 90.48 %
----------
CV accuracy: 97.25 %
```

Cross-validating a set of Classifiers

- A nice property of sklearn is that each predictor conforms to the same interface (*i.e.*, each implements `.fit()`, and `.predict()` methods).
- Thus, we can examine a list of them in a `for` loop easily.
- It seems that the linear classifiers (Logistic Regression and Linear SVM) are the best performers for this case.

Code:

```
100 # Test a collection of classifiers
101
102 classifiers = [LogisticRegression(),
103                 LDA(),
104                 LinearSVC(),
105                 SVC(),
106                 RandomForestClassifier(),
107                 KNeighborsClassifier()
108                 ]
109
110 names = ['LogReg',
111          'LDA',
112          'LinSVM',
113          'SVM',
114          'RF',
115          'NN']
116
117 for i, clf in enumerate(classifiers):
118
119     scores = cross_val_score(clf,
120                              X,
121                              y,
122                              cv = 10,
123                              n_jobs = 2)
124     print "Classifier %s: score = %.2f %%" % \
125           (names[i], 100*np.mean(scores))
```

Result:

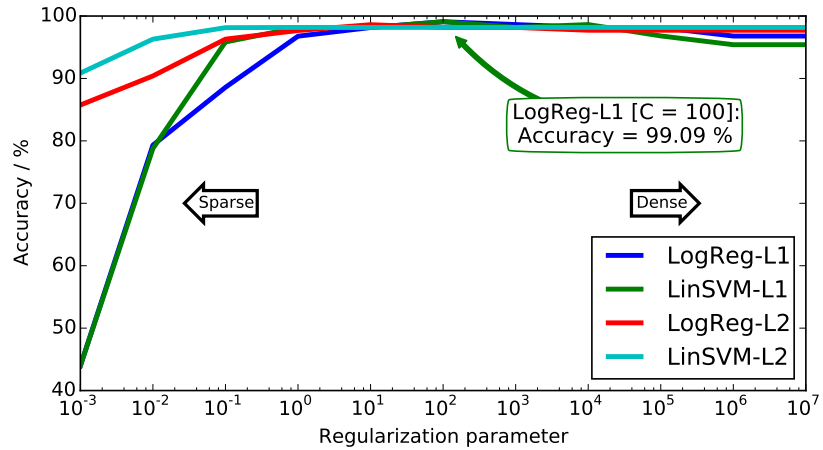
```
Classifier LogReg: score = 97.25 %
Classifier LDA: score = 94.39 %
Classifier LinSVM: score = 97.25 %
Classifier SVM: score = 79.82 %
Classifier RF: score = 89.35 %
Classifier NN: score = 88.50 %
```

Regularization

- Now that linear classifiers seem to be suitable for this data, we will see how regularization affects the results.
- Remember: Regularization penalizes for large coefficients and thus prevents overlearning. Moreover, ℓ_1 regularization makes the coefficients sparse.
- The code on the right calculates the CV-10 error for a range of regularization parameters.
- The resulting accuracies are plotted at the bottom.

```
130 cRange = 10.0**np.arange(-3,8)
131
132 classifiers = [LogisticRegression(penalty = 'l1'),
133                 LinearSVC(penalty = 'l1', dual = False),
134                 LogisticRegression(penalty = 'l2'),
135                 LinearSVC(penalty = 'l2')
136                 ]
137
138 names = ['LogReg-L1',
139          'LinSVM-L1',
140          'LogReg-L2',
141          'LinSVM-L2'
142          ]
143
144 plt.figure()
145
146 bestScore = 0
147
148 for i, clf in enumerate(classifiers):
149
150     accuracies = []
151
152     for C in cRange:
153         print names[i], C
154         clf.C = C
155         scores = cross_val_score(clf,
156                                  X,
157                                  y,
158                                  cv = 10,
159                                  n_jobs = 4)
160         meanScore = 100*np.mean(scores)
161         accuracies.append(meanScore)
```

Result:

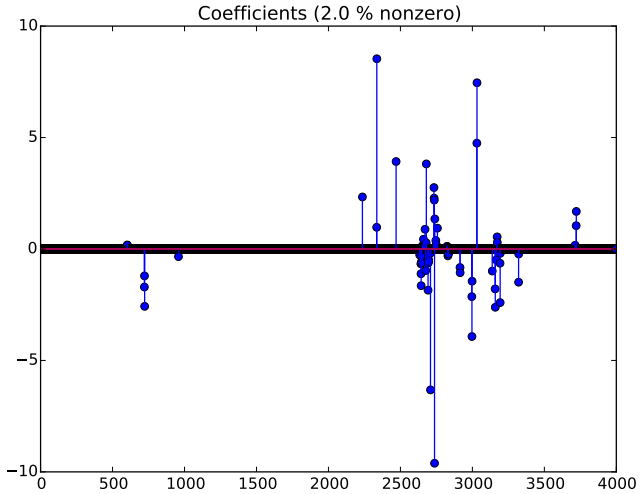


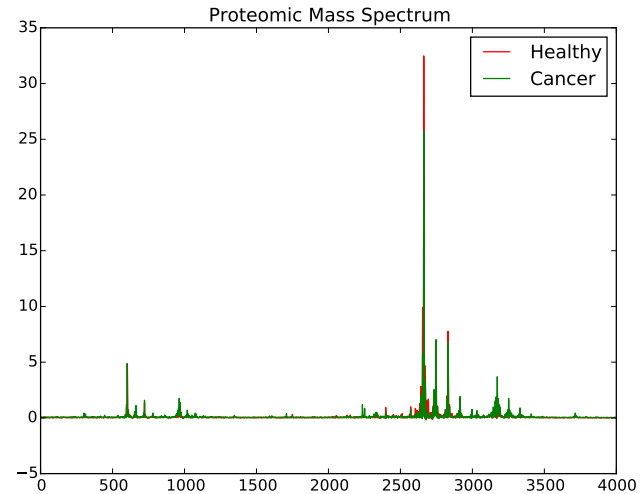
Analysis of the Classifier

- The ℓ_1 regularized Linear SVM was the most accurate classifier.
- It is defined by its coefficients $\mathbf{w}$.
- The nonzero coefficients correspond to the peaks of the mass spectra.
- The decision rule is very simple

$$\text{prediction} = \begin{cases} \text{cancer}, & \text{if } \sum w_k x_k \geqslant b \\ \text{healthy}, & \text{if } \sum w_k x_k < b \end{cases}$$

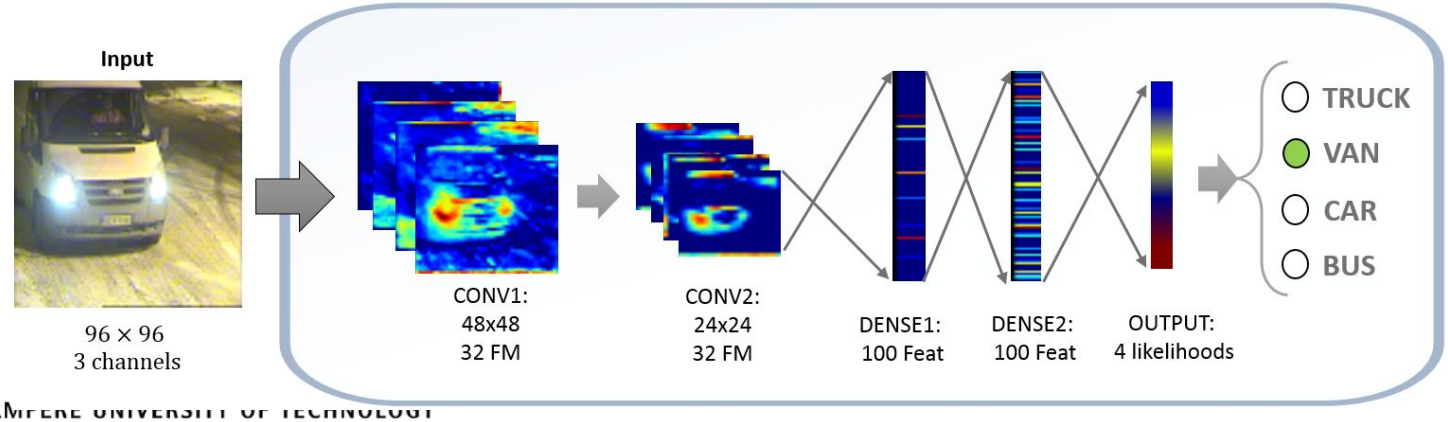
with w_k the coefficients, x_k the mass spectrum and b the threshold (which happens to be $b = 0$ this time).





Recent Developments

- One of the most exiting new research directions is *Deep Learning*, which studies *neural networks* with many layers.
- The methodology has set new records in various areas that are abundant with data, *e.g.*, speech recognition, image recognition, etc.
- The default language of most available tools is Python.
- Under the hood, the engines are typically implemented in C++ or CUDA, that links to the Python front end.



Deep Learning Tools

- The four most popular Deep learning toolboxes are (in the order of decreasing popularity):
  - **Caffe**: http://caffe.berkeleyvision.org/ from Berkeley
  - **Torch**: http://torch.ch/ from NYU (used by Facebook)
  - **pylearn2**: http://deeplearning.net/ from U. Montreal
  - **Minerva** https://github.com/dmlc/minerva from New York, Peking Univ.
- All except Torch can be used via a Python front end (Torch uses Lua language).

Applications

- **Application example 1:** ICANN MEG Mind Reading Challenge, June 2011
- **Application example 2:** IEEE MLSP 2012 Birds competition, Sept. 2013
- **Application example 3:** DecMeg2014: Decoding the Human Brain, July 2014

ICANN MEG Mind Reading Challenge

- We participated in the Mind reading challenge from MEG data organized in the ICANN 2011 conference. [2] [3] [4]

Rank	Team	Affiliation	Accuracy (%)
1.	Huttunen et al.	Tampere University of Technology	68.0
2.	Santana et al.	Universidad Politecnica de Madrid	63.2
3.	Jylänki et al.	Aalto University	62.8
4.	Tu & Sun (1)	East China Normal University	62.2
5.	Lievonen & Hyötyniemi	Helsinki Institute for Information Technology	56.5
6.	Tu & Sun (2)	East China Normal University	54.2
7.	Olivetti & Melchiori	University of Trento	53.9
8.	Van Gerven & Farquhar	Radboud University Nijmegen	47.2
9.	Grozea	Fraunhofer Institute FIRST	44.3
10.	Nicolaou	University of Cyprus	24.2

[2] H. Huttunen *et al.*, "Regularized logistic regression for mind reading with parallel validation," *Proc. ICANN/PASCAL2 Challenge: MEG Mind-Reading. Aalto University publication series*, Espoo, June 2011.

[3] H. Huttunen *et al.*, "MEG Mind Reading: Strategies for Feature Selection," in *Proc. of The Federated Computer Science Event 2012*, Helsinki, May 2012.

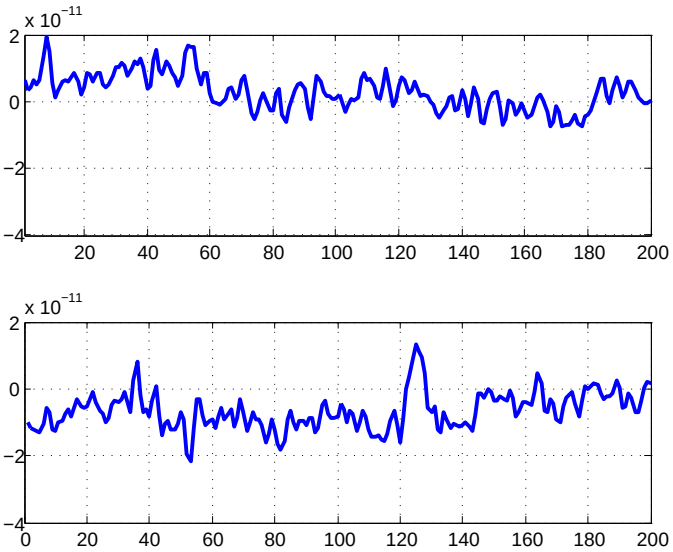
[4] H. Huttunen *et al.*, "Mind Reading with Regularized Multinomial Logistic Regression," *Machine Vision and Applications*, Aug. 2013

ICANN MEG Mind Reading Challenge Data

- The competition data consists of MEG recordings while watching five different movie categories.
- The data contains measurements of 204 planar gradiometer channels at 200Hz rate, segmented into samples of one second length.
- The task was to design and implement a classifier that takes as an input the MEG signals and produces the predicted class label.
- The training data with annotations had 727 one-second samples, and the secret test data 653 unlabeled samples.
- 50 samples of the training data were special, because they were recorded on the same day as the test data.

The Curse of Dimensionality

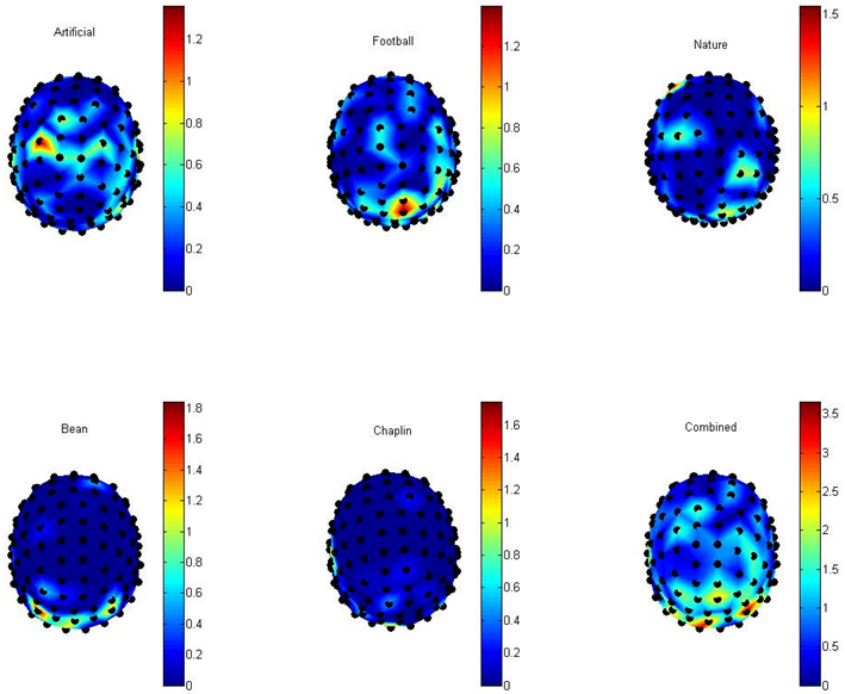
- Since each measurement is a time series, it cannot be fed to classifier directly.
- Instead, we calculated a number of common quantities.
- In all, our pool of features consists of $11 \times 204 = 2244$ features. This means $2244 \times 5 = 11220$ parameters.



Intercept	$x_i^{(1)} = \hat{b}_i$
Slope	$x_i^{(2)} = \hat{a}_i$
Variance (d)	$x_i^{(3)} = \frac{1}{N}\sum_{n=1}^{N} \tilde{s}_i^2(n)$
Std. dev. (d)	$x_i^{(4)} = \sqrt{x_i^{(3)}}$
Skewness (d)	$x_i^{(5)} = \frac{1}{N}(x_i^{(4)})^{-3}\sum_{n=1}^{N} \tilde{s}_i^3(n)$
Kurtosis (d)	$x_i^{(6)} = \frac{1}{N}(x_i^{(4)})^{-4}\sum_{n=1}^{N} \tilde{s}_i^4(n)$
Variance	$x_i^{(7)} = \frac{1}{N}\sum_{n=1}^{N}(s_i(n) - \hat{b}_i)^2$
Std. dev.	$x_i^{(8)} = \sqrt{x_i^{(7)}}$
Skewness	$x_i^{(9)} = \frac{1}{N}(x_i^{(8)})^{-3}\sum_{n=1}^{N}(s_i(n) - \hat{b}_i)^3$
Kurtosis	$x_i^{(10)} = \frac{1}{N}(x_i^{(8)})^{-4}\sum_{n=1}^{N}(s_i(n) - \hat{b}_i)^4$
Fluctuation	$x_i^{(11)} = \frac{1}{N-1}\left\lvert\sum_{n=2}^{N} \operatorname{sgn}(s_i(n) - s_i(n-1))\right\rvert$

Where the Selected Features are Located?

- As a side product we gain insight on which areas of the brain were used for prediction.

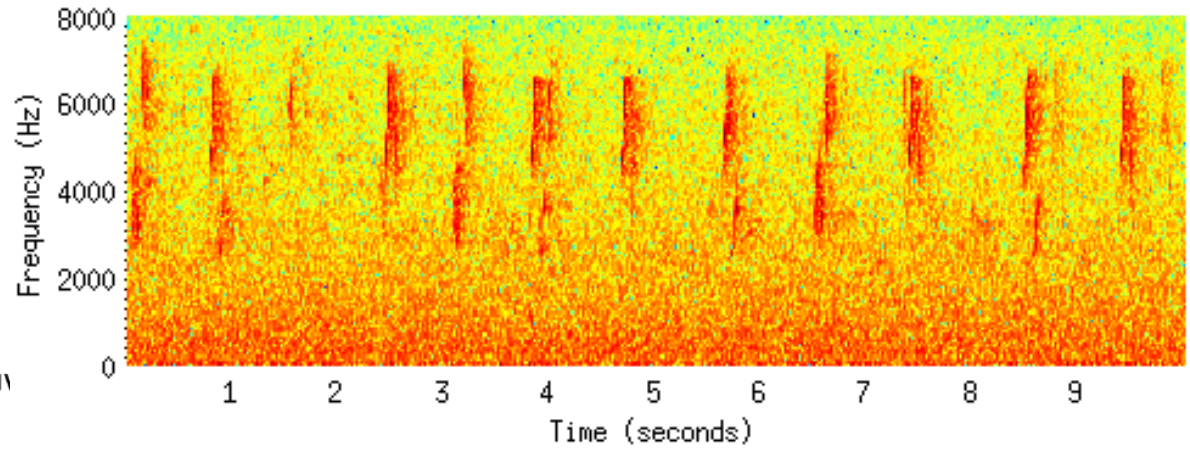


IEEE MLSP2013 Birds Competition

- The task of the participants was to train an algorithm to recognize *bird sounds* recorded at the wildlife.
- The recordings consisted of bird sounds from $C = 19$ species, and the sounds were overlapping (with up to 6 birds in a single recording).
- There were altogether 645 ten-second audio clips with sample rate $F_s = 16$ kHz.
- Half of the samples ($N_{\text{train}} = 323$) were used for model training, and half ($N_{\text{test}} = 322$) of the data was provided without annotation.

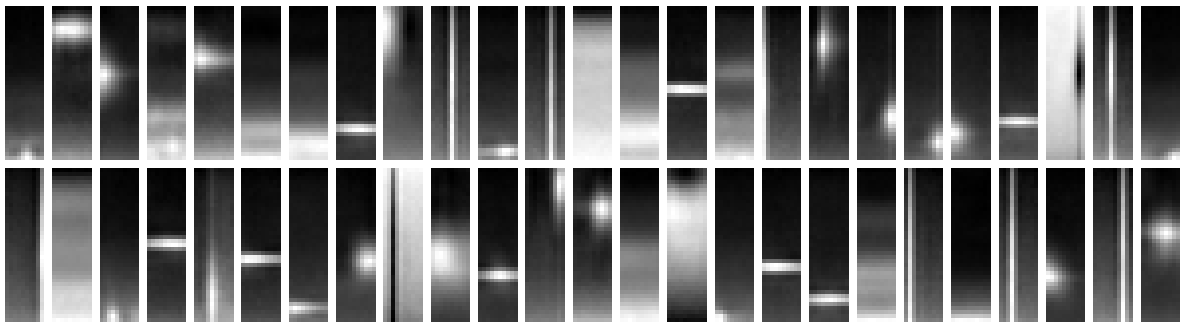
IEEE MLSP2013 Birds Competition

- The task of the participants was to train an algorithm to recognize *bird sounds* recorded at the wildlife.
- The recordings consisted of bird sounds from $C = 19$ species, and the sounds were overlapping (with up to 6 birds in a single recording).
- There were altogether 645 ten-second audio clips.
- Half of the samples were used for model training, and the participants should predict the labels of the other half.
- We participated in the challenge, and our final submission placed 7th among the 77 teams.



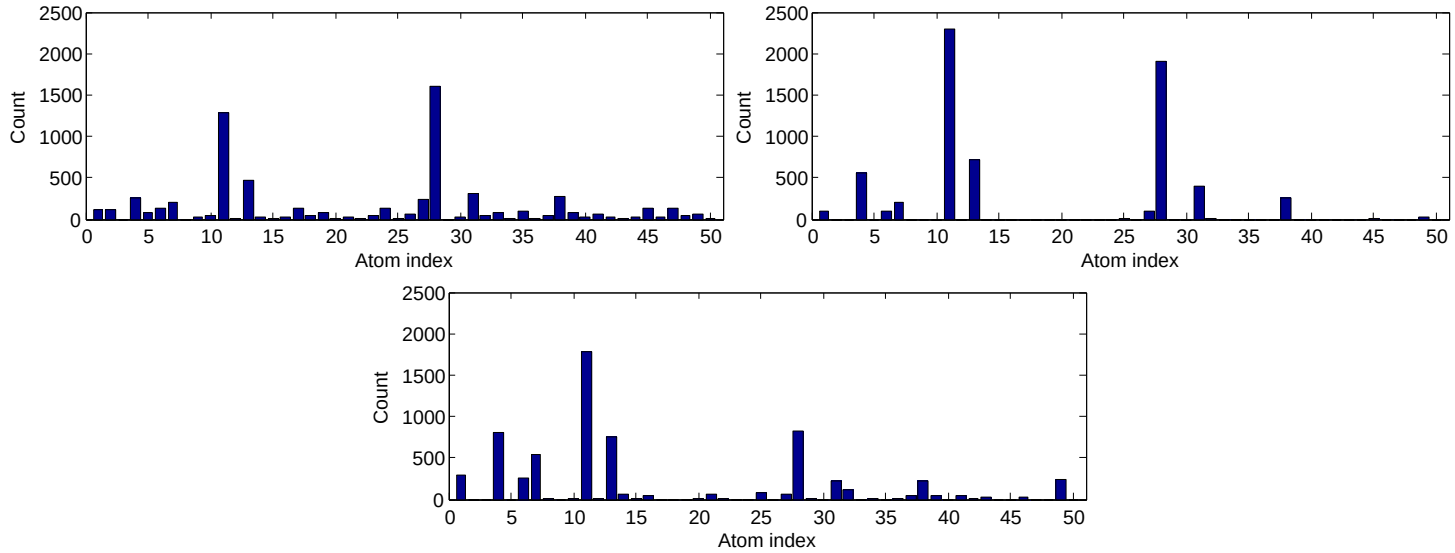
IEEE MLSP2013 Birds — Our Method

- Our approach mixed six prediction models by averaging the predicted class probabilities together.
- Most fruitful model used dictionary based Bag-of-Words features.
  - Learn a dictionary of typical patches in the spectrograms (including the non-annotated ones).
  - Count the occurrences of each dictionary atom in each spectrogram.



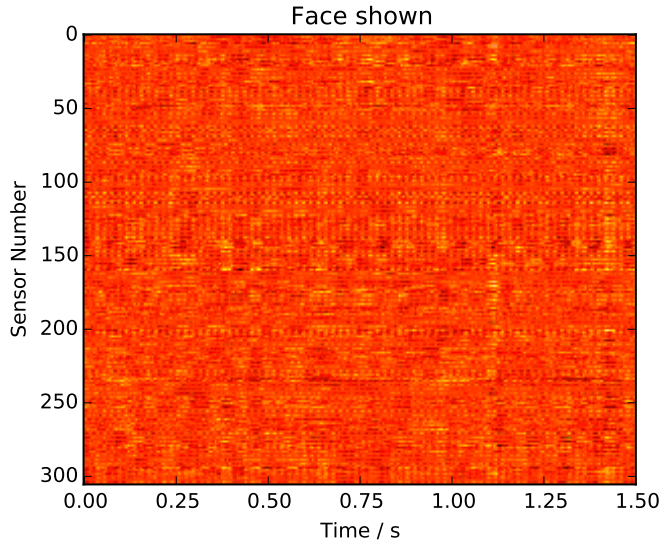
IEEE MLSP2013 Birds — Our Method

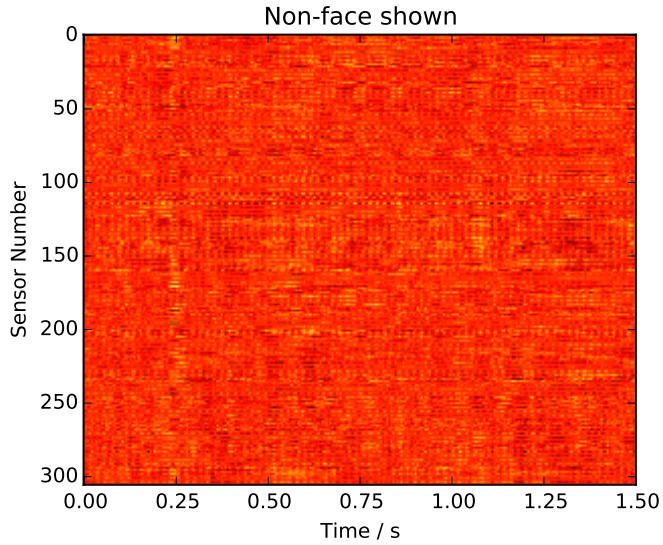
- Examples of patch histograms are shown below.
- Left: two bird species present, Right: no birds present, Bottom: a single bird present.
- One can spot the noise encoding patches (1, 4, 6, 7, 11 . . .), which are active in all histograms



DecMeg2014 Competition

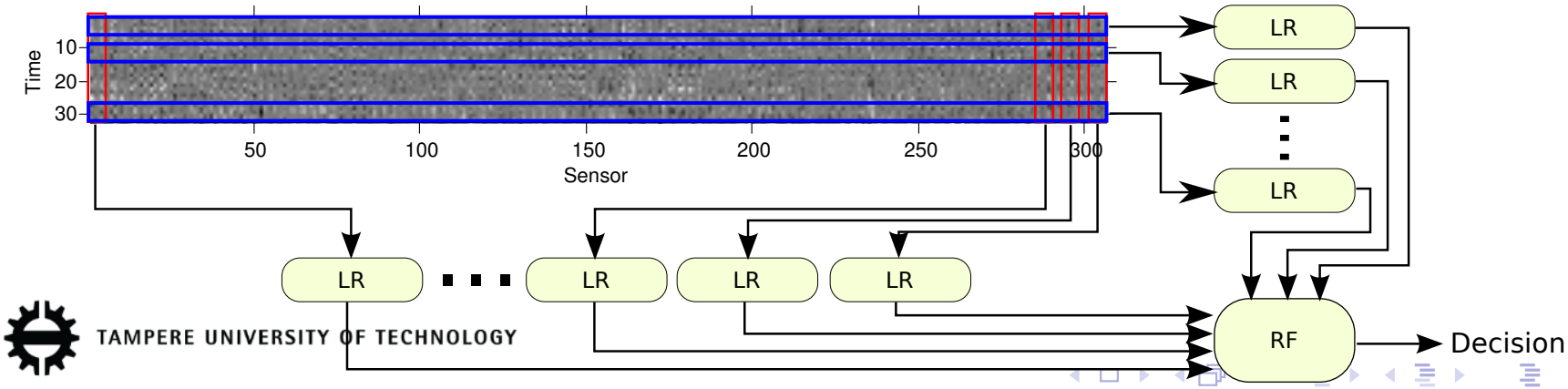
- In the 2014 DecMeg competition, the task was to predict whether test persons were shown a **face** or a **non-face** using MEG recordings of the brain.
- Sequences were 1.5 second long measurements with 306 channels; face shown at 0.5 s - 1.5 s.
- In total, 16 training and 7 test subjects; approximately 600 datapoints from each subject.
- Intersubject variation is significant.





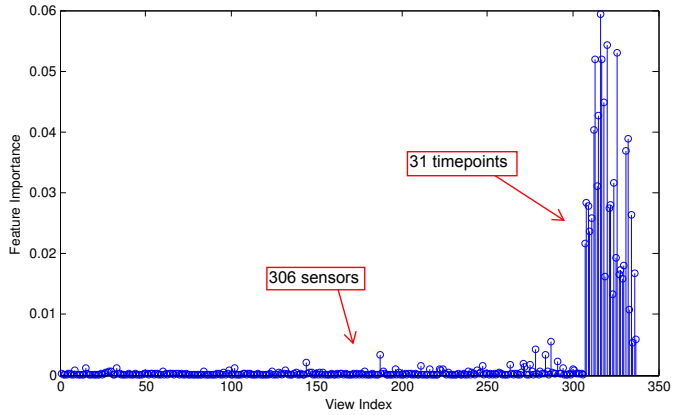
Our Model

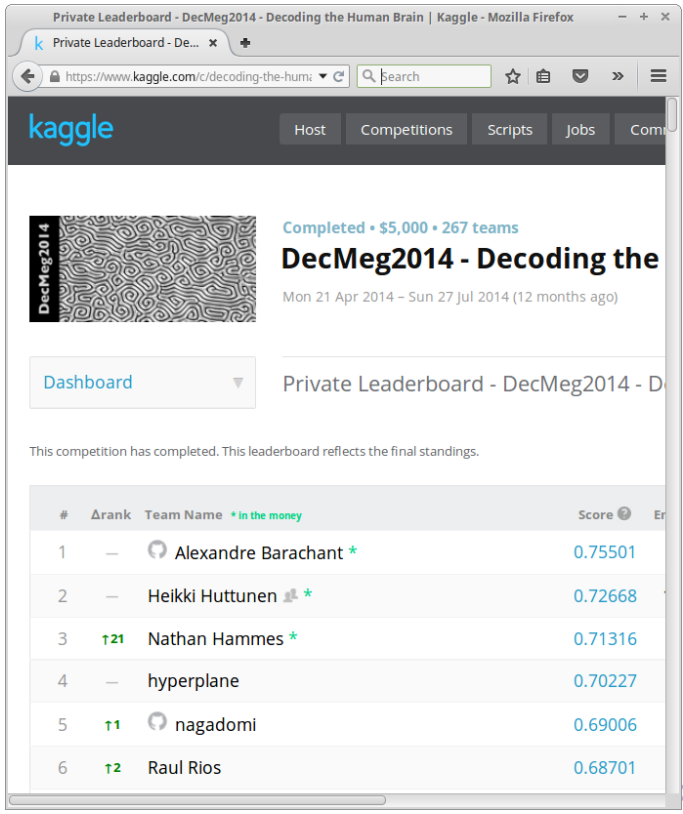
- Our is a hierarchical combination of two layers of classifiers:
  - First layer consists of Logistic Regression classifiers mapping individual sensors or individual timepoints to class probabilities.
  - The second layer is a 1000-tree Random Forest with LR predictions as inputs.
- The idea of stacking classifiers originates to Wolpert's *stacked generalization* (1992), and was inspired by our first attempts to approach the problem using a deep convolutional net.



Feature Importances

- The reliability of each 1st layer classifier can be seen from the feature importances of the Random Forest.
- The full time series from individual sensors seem a lot more informative than snapshots of the whole brain.
- Eventually our method was successful placing us on the 2nd place of 267 teams.
- The implementation is 100 % `sklearn`: https://github.com/mahehu/decmeg





Summary

- The significance of Python as a language of scientific computing has exploded.
- Machine learning has been one of the key areas in this development.
- This has been coined by the open-access attitude of the community: licensing tends to be very free.
- One of the key benefits of `scikit-learn` is the uniform API: It's easy to test a wide range of algorithms with a short piece of code.
- Deep learning is a growing trend—powered by Python.