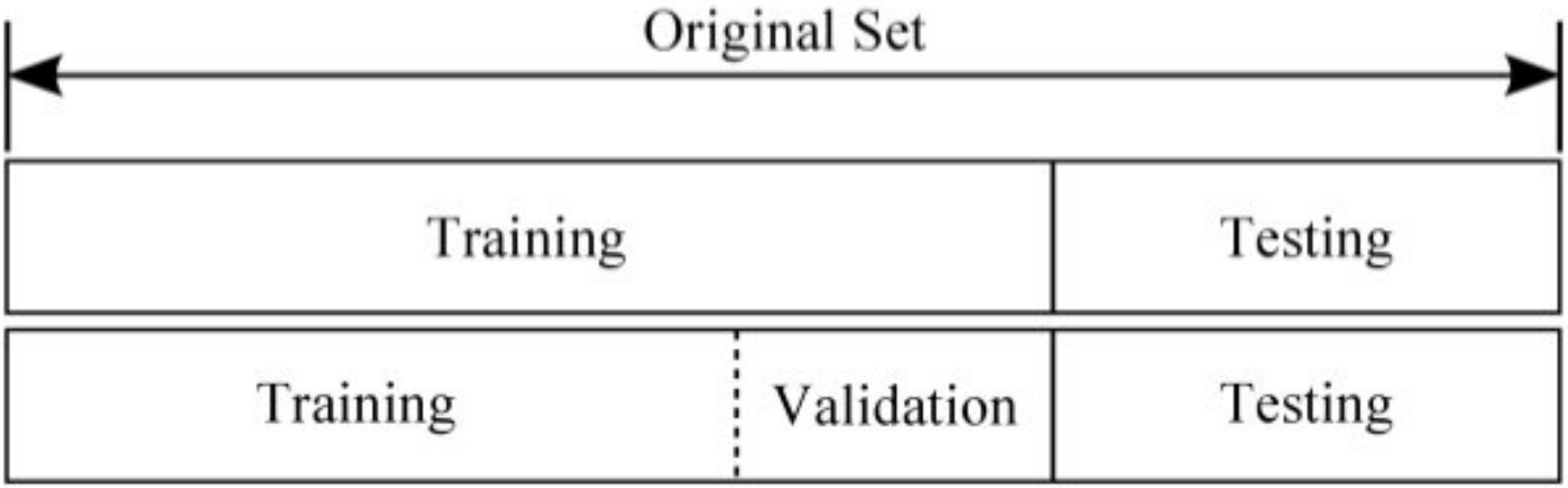
Data

Cédric Buche

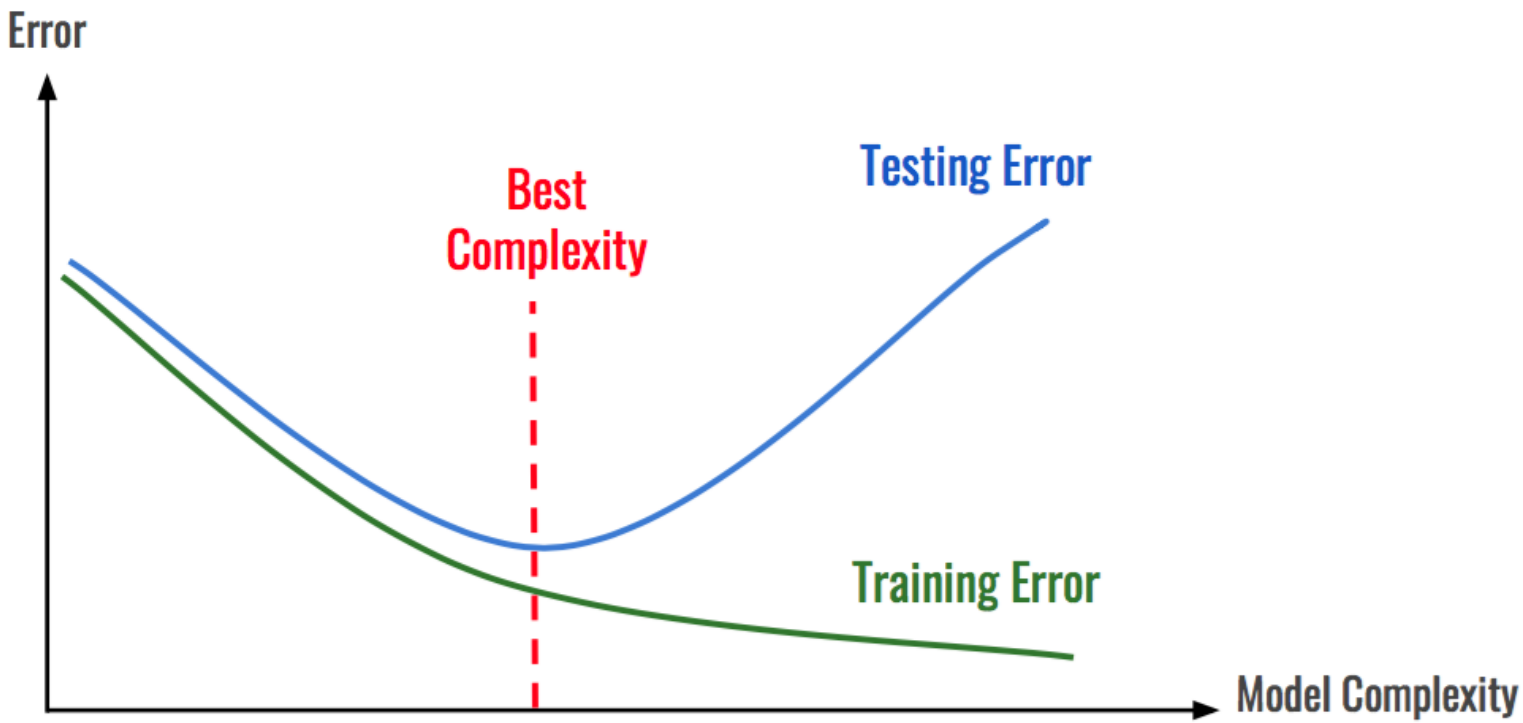
ENIB

September 9, 2019

Dataset


Original Set
Training
Testing
Training
Validation
Testing

Dataset


Error
Best
Complexity
Testing Error
Training Error
Model Complexity

```
def split_data(data,prob):
        # split data into fractions [prob, 1 - prob]
        results=[],[]
        for row in data:
                results[0 if random.random() < prob else 1].append(row)
        return results
```

```
def train_test_split(x,y,test_pct):
        # pair corresponding values
        data = zip ( x , y )
        # split the data set of pairs
        train , test = split_data ( data , 1 - test_pct )
        x_train , y_train = zip ( * train )
        x_test , y_test = zip ( * test )
        return x_train , x_test , y_train , y_test
```

```
model = SomeKindOfModel ()
x_train , x_test , y_train , y_test = train_test_split ( xs , ys , 0.33 )
model.train ( x_train , y_train )
performance = model.test ( x_test , y_test )
```

Hyper Parameter tuning

- the parameters of the learning phase: hyper-parameters.
- example: maximum number of values that will be tested in a node of a decision tree, or the number of trees that will contain a random forest.
- no formal method to find the optimal values from the training data.
- often use exhaustive search on ranges defined by the developer: this requires in practice to make as many learnings as combinations of parameters. This technique is called *Grid Search*. It uses one of the model's quality metrics to select the best set of hyper-parameters.

Features and Label

- the "features": we can measure them and it is from them that we will perform modeling and prediction.
- the "label": the data that we are trying to predict: in the case of supervised learning, we have the explanatory variable in the learning data.

Preparation of complex data

- voice (Automatic Speech Recognition or Speech-To-Text) : Google cloud mode or Nuance solutions
- images : Imagemagick, OpenCV2

Introduction

- data complexity: graphical analysis by data scientist
- highlight relationships between different dimensions
- quantify this relationship
- tool: linear regression

NBA: size / weight relationship

- it is hinted that the weight must increase with size, but to what extent?
- Is it possible to predict the weight of a player who knows his size?

Pandas

```
import pandas as pd
import matplotlib.pyplot as plt
from numpy.linalg import inv
import numpy as np

df = pd.read_csv('players_stats.csv')
height = df.dropna()['Height']
weight = df.dropna()['Weight']

plt.xlabel('Height (cm)')
plt.ylabel('Weight (kg)')

plt.scatter(height, weight)
plt.show()
```

NBA: size / weight relationship

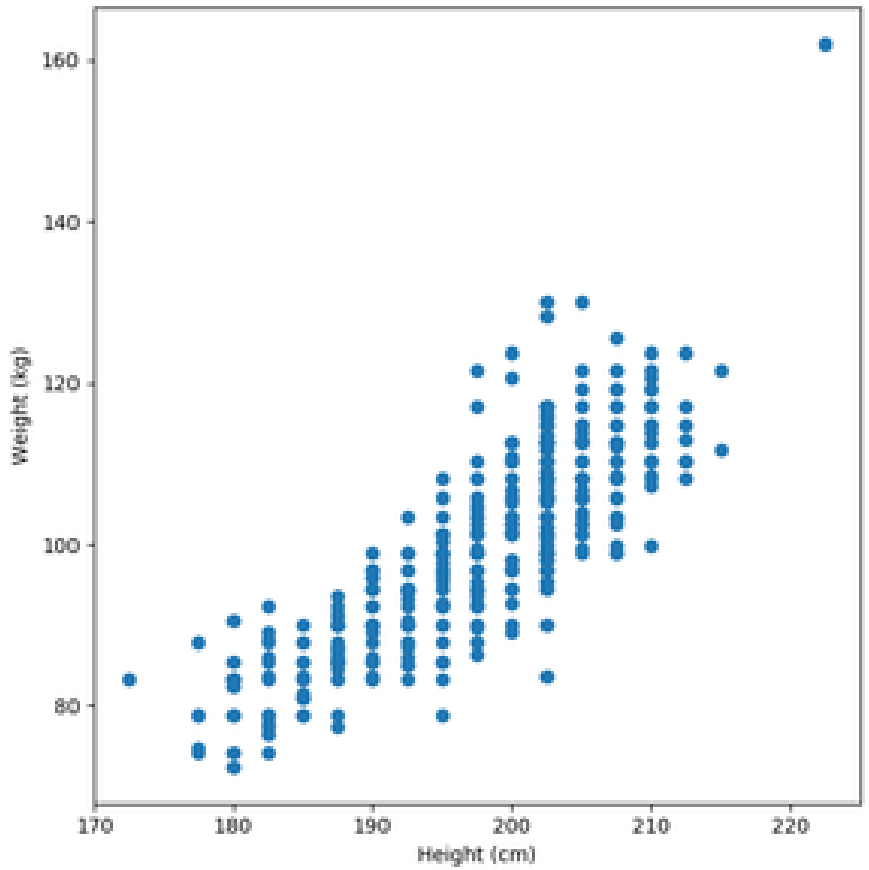


Figure: The weight of our players grows well with their size, and moreover linearly.

The mathematical tool

- establish a mathematical relationship between height and weight
- regression: fit a mathematical model to a set of measures
- linear regression: $y = a * x + b$ where x is named predictor, while y is the variable to predict.
- NBA, x is the size of the players, while y is their weight.
- we have a set of samples of y values for various values of x
- link model and samples:

$$e = \sum_{i=0}^{n} (a * x_i + b - y_i)^2$$

The mathematical tool

We will work in matrix form:

$$e=(X*A-Y)^T(X*A-Y)=(X*A-Y)^2$$

Y is a column vector containing yi

X is a matrix consisting of two columns. The first contains the predictors xi while the second contains only 1.

The mathematical tool

A meanwhile, is a line vector containing $[ab]$. The derivative of e with respect to the parameters we wish to optimize, a and b, contained in A, is:

$$\frac{\partial e}{\partial A} = \frac{\partial (X*A-Y)^T}{\partial A} * (X*A-Y) = X^T (X*A-Y)$$

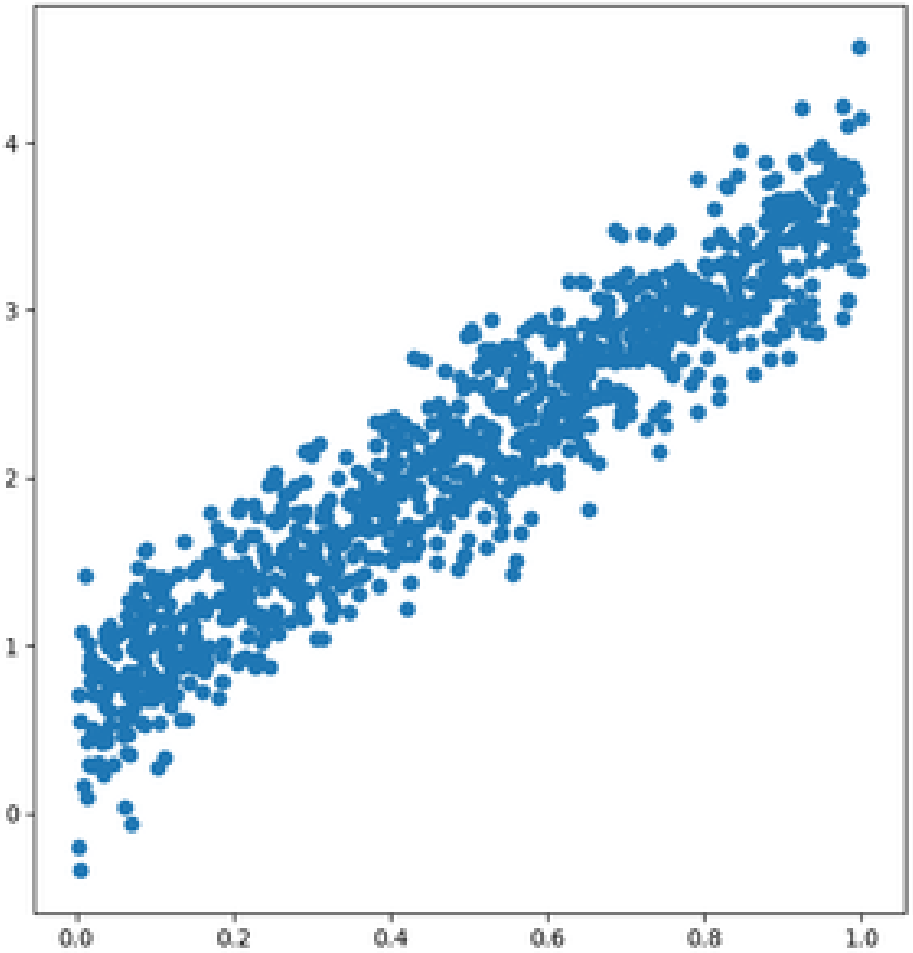
e reaches its minimum when this expression is null, that is:

$$X^T (X*A-Y) = 0$$
$$X^T X * A = X^T Y$$
$$A = (X^T X)^{-1} X^T Y$$

An example of linear data distributed according to a Gaussian

```
import numpy as np
importmath
importrandom
from numpy.linalg import inv
import matplotlib.pyplot as plt

nbSamples =1000

X = np.matrix([[random.random(), 1] for x in range(nbSamples)])
Y = np.matrix([3*x[0].item(0)+ 0.666 for x in X]).transpose()

Gnoise = np.random.normal(0.0,0.1,len(Y))
Ynoisy = np.matrix([Y[i].item(0)+ Gnoise[i] for i in range(len(Y))]).transpose()

plt.scatter(np.asarray(X[:,0]), np.asarray(Ynoisy))
plt.show()
```

An example of linear data distributed according to a Gaussian.



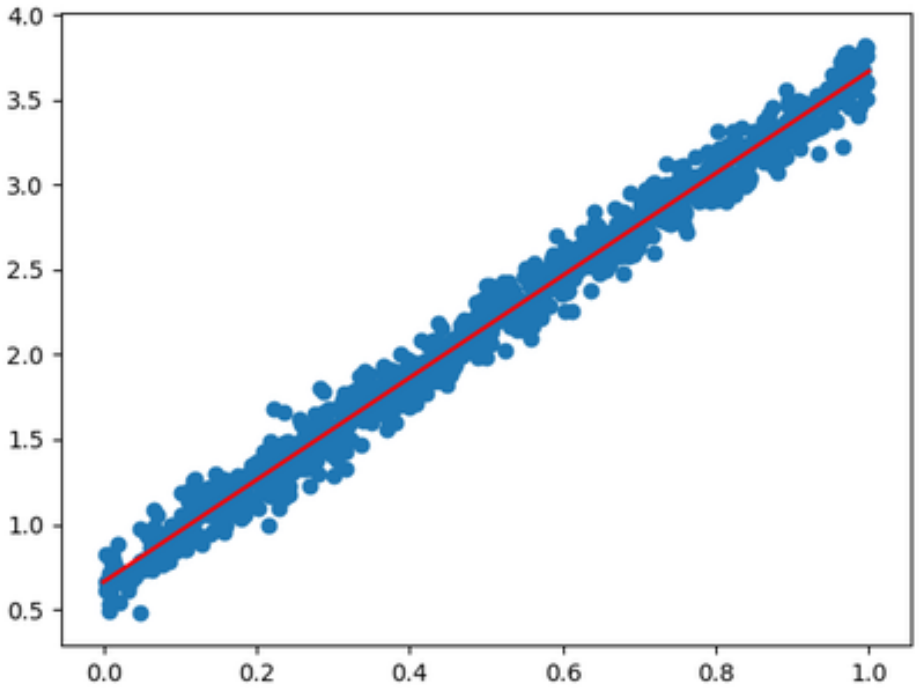
An example of linear data distributed according to a Gaussian.

```
# Find a and b
A = inv(X.transpose()*X)*X.transpose()*Ynoisy
print(A)
>[[3.00512112]
>[0.66163949]]
```

An example of linear data distributed according to a Gaussian.

```
x =[0,1]
y =[[x[0],1],[x[1],1]]*A
plt.scatter(np.asarray(X[:,0]), np.asarray(Ynoisy))
plt.plot(x, y, color='r')
plt.show()
```

An example of linear Gaussian distributed data, and the associated linear regression.



NBA

```
import pandas as pd
import matplotlib.pyplot as plt
from numpy.linalg import inv
import numpy as np

df = pd.read_csv('players_stats.csv')
height = df.dropna()['Height']
weight = df.dropna()['Weight']

X = np.zeros((len(height),2))
X[:,0]= height
X[:,1]=1
Xm = np.matrix(X)
```

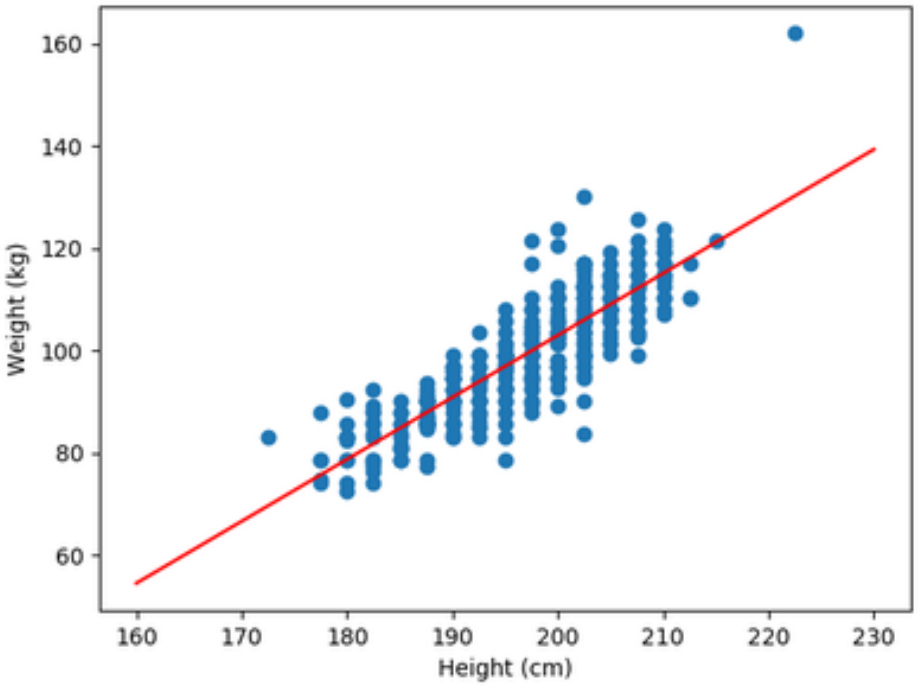
NBA

```
Y = np.matrix(weight.as_matrix())
A = inv(Xm.transpose()*Xm)*Xm.transpose()*Y.transpose()
```

NBA

```
x =[160,230]
y =[[x[0],1],[x[1],1]]*A

plt.xlabel('Height␣(cm)')
plt.ylabel('Weight␣(kg)')
plt.scatter(height, weight)
plt.plot(x, y, color='r')
plt.show()
```

NBA



The least squares method allows us to say that a player of 2.10m must weigh not far from 116 kilos

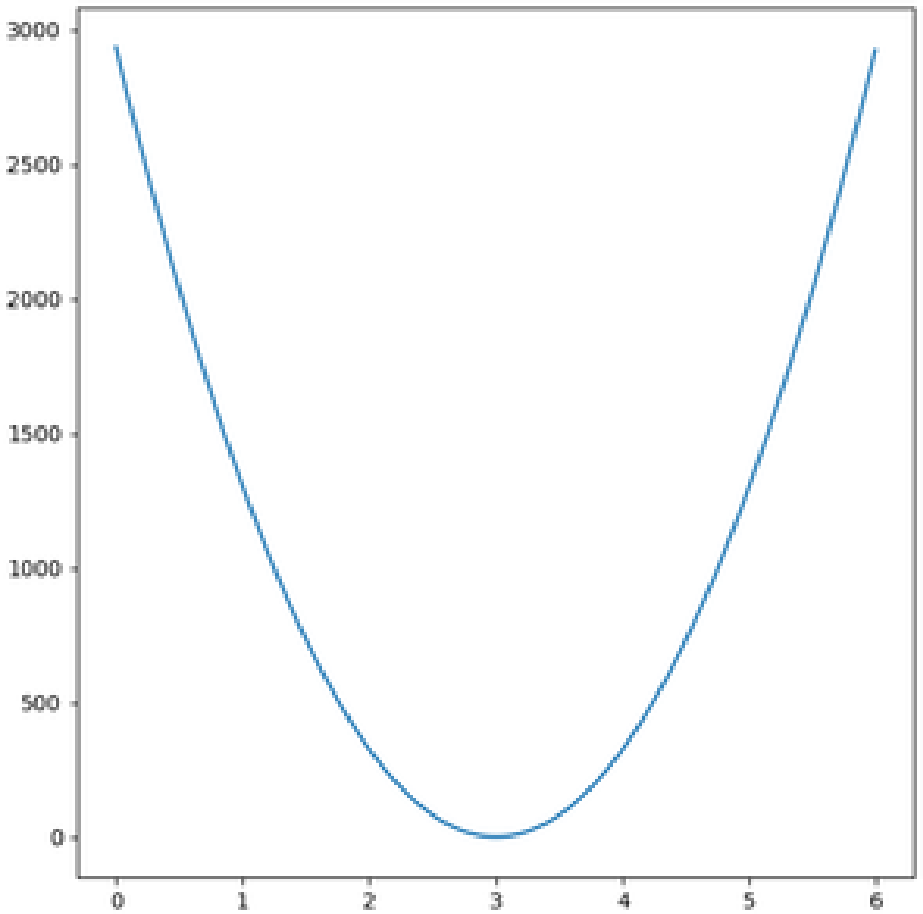
Computer tools

This method works very well, but may become impractical if the number of columns of X becomes too large, the cost of an inversion being in the general case in $O(n3)$. The memory cost can also become prohibitive.

1. work with a subset representative of the total ensemble
2. develop an inversion algorithm
3. opt for an iterative approach, where we start from (a, b) to converge progressively to.

Let's plot the error according to *a*

```
import autograd.numpy as np
from autograd import grad
import math
import random
from numpy.linalg import inv
import matplotlib.pyplot as plt

nbSamples =1000
X = np.matrix([[random.random(), 1]for x inrange(nbSamples)])
Y = np.matrix([3*x[0].item(0)+ 0.666for x in X]).transpose()

def error(X, Y, a):
   a = np.matrix([[a],[0.666]])
   e = X*a - Y
   return(e.transpose()* e).item(0)

def genError(X, Y):
   return lambda a : error(X, Y, a)

err = genError(X, Y)
xs = [x *6.0/ nbSamples for x inrange(nbSamples)]
e = [err(x)for x in xs]
plt.plot(xs, e)
```


3000
2500
2000
1500
1000
500
0
0
1
2
3
4
5
6

Iterative Approach

```
grad_err = grad(err)

def newtonStep(f0, df, x0):
    df0 = df(x0)
    x1 = x0 - f0/ df0
    return x1

def newtonSolver(f, df, x0):
    count =0
    f0 = f(x0)
    whileTrue:
        x0 = newtonStep(f0, df, x0)
        print("iter %d: %f"%(count, x0))
        count +=1
        f0 = f(x0)
        if f0< 1e-6:
            break
    return x0

newtonSolver(err, grad_err,0)
```

Iterative Approach

```
iter 0 : 1.500000
iter 1 : 2.250000
iter 2 : 2.625000
iter 3 : 2.812500
iter 4 : 2.906250
iter 5 : 2.953125
iter 6 : 2.976562
iter 7 : 2.988281
iter 8 : 2.994141
iter 9 : 2.997070
iter 10 : 2.998535
iter 11 : 2.999268
iter 12 : 2.999634
iter 13 : 2.999817
iter 14 : 2.999908
iter 15 : 2.999954
```

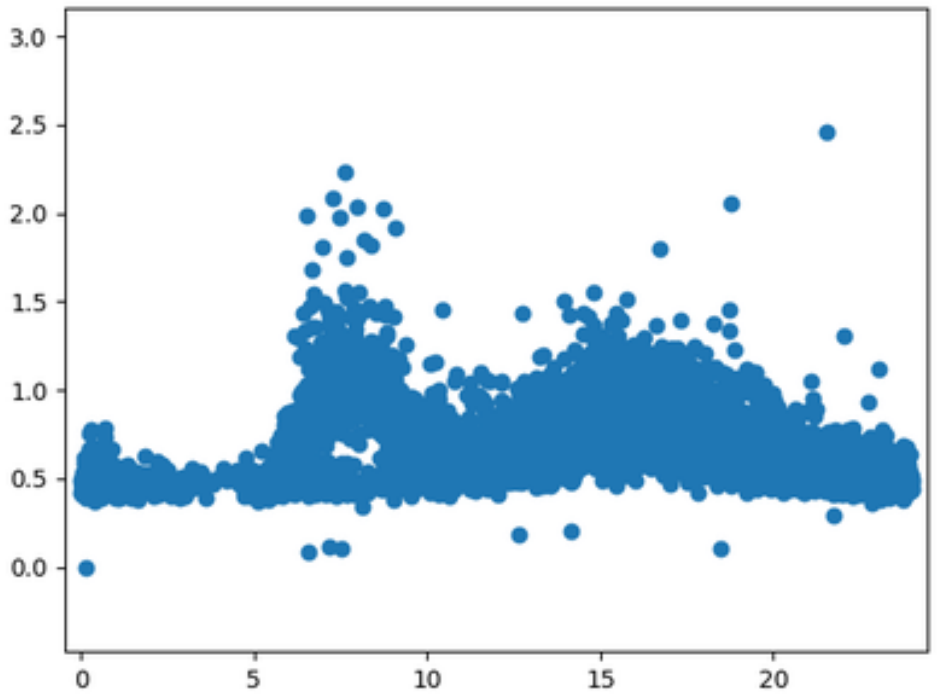
A non-linear problem

- New York : 7 years of taxi and limousine journeys (1.1 billion trips)
- route information for YellowCabs, GreenCabs and ForHireVehicle (FHV)
- the FHV only have three measurements per way
- Yellow and GreenCabs:
  - the distance;
  - the collection point;
  - the drop point;
  - the price of the trip;
  - the amount of the tip;
  - the number of passengers.

from JFK Airport to Manhattan's UpperEastSide.

```
import pandas as pd
from dateutil importparser
import matplotlib.pyplot as plt

cols =['PULocationID','DOLocationID','tpep_pickup_datetime','tpep_dropoff_datetime','
      trip_distance']

dfJ = pd.read_csv('yellow_tripdata_2017-01.csv', usecols=cols)
dfF = pd.read_csv('yellow_tripdata_2017-02.csv', usecols=cols)
dfM = pd.read_csv('yellow_tripdata_2017-03.csv', usecols=cols)
dfA = pd.read_csv('yellow_tripdata_2017-04.csv', usecols=cols)
dfMy = pd.read_csv('yellow_tripdata_2017-05.csv', usecols=cols)

df = dfJ.append(dfF).append(dfM).append(dfA).append(dfMy)

#236 manhattan upper east side
JFK_MU = df[(df['PULocationID']==132)&(df['DOLocationID']==236)]


JFK_MU.to_csv("JFKraw.csv", columns=cols)

pu = [parser.parse(dt)for dt in JFK_MU['tpep_pickup_datetime'].values]
do = [parser.parse(dt)for dt in JFK_MU['tpep_dropoff_datetime'].values]
dur = [(b -a).total_seconds()/ 3600.0for a, b inzip(pu, do)]
startTime = [dt.hour+ dt.minute/ 60.0for dt in pu]

plt.scatter(startTime, dur)
plt.show()
```

Travel time between JFK and Upper East Side depending on time of departure.

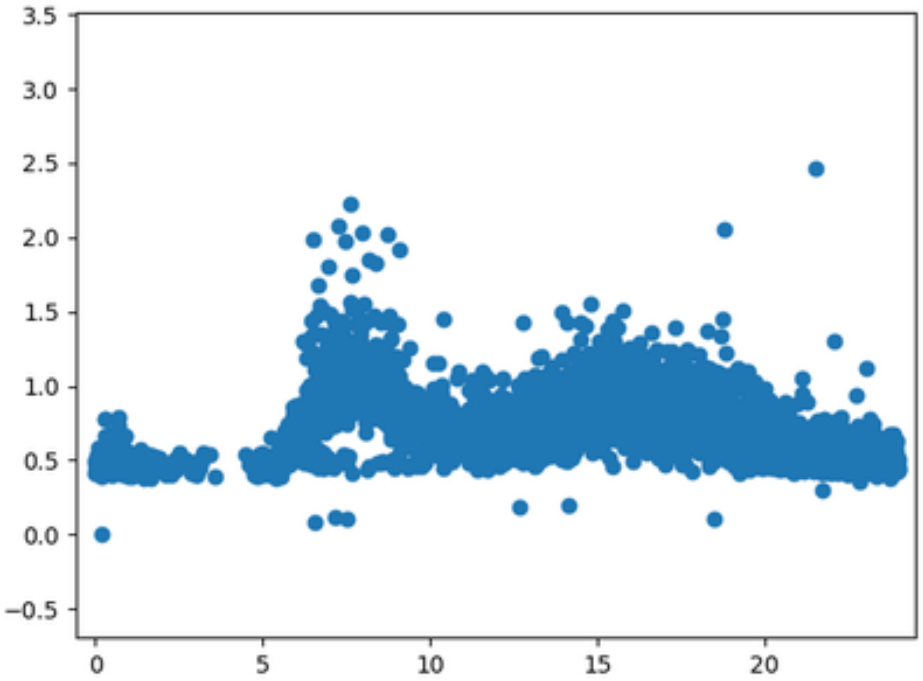


Cleaning

- two peaks are around 7am and 4pm
- the peak of 7am is not always a real one
- It's a safe bet that these easy-going points are just weekend days (and probably holidays)

```
import pandas as pd
from dateutil importparser
import matplotlib.pyplot as plt

cols =['PULocationID','DOLocationID','tpep_pickup_datetime','tpep_dropoff_datetime','
      trip_distance']

dfJ = pd.read_csv('yellow_tripdata_2017-01.csv', usecols=cols)
dfF = pd.read_csv('yellow_tripdata_2017-02.csv', usecols=cols)
dfM = pd.read_csv('yellow_tripdata_2017-03.csv', usecols=cols)
dfA = pd.read_csv('yellow_tripdata_2017-04.csv', usecols=cols)
dfMy = pd.read_csv('yellow_tripdata_2017-05.csv', usecols=cols)

df = dfJ.append(dfF).append(dfM).append(dfA).append(dfMy)

JFK_MU = df[(df['PULocationID']==132)&(df['DOLocationID']==236)]

JFK_MU['weekday']= JFK_MU['tpep_pickup_datetime'].apply(lambda x :parser.parse(x).weekday())

JFK_MU = JFK_MU[JFK_MU['weekday']<5]

pu = [parser.parse(dt)for dt in JFK_MU['tpep_pickup_datetime'].values]
do = [parser.parse(dt)for dt in JFK_MU['tpep_dropoff_datetime'].values]
dur = [(b -a).total_seconds()/ 3600.0for a, b inzip(pu, do)]
startTime = [dt.hour+ dt.minute/ 60.0for dt in pu]

plt.scatter(startTime, dur)
plt.show()
```

All aberrations (7am) are almost disappeared.



Cerce

- example of data that clearly does not fit into a linear model
- use a linear regression: *splines*
  - interval $[xmin, xmax]$ on which the spline is defined is divided into n control points x_i.
  - At each of these points of control, we add a new line, which alters the pace of the curve defined at this point.
  - we build a series of functions, generally noted $I^i_{plus}(x)$ which are zero until x_i and the value is $x - x_i$ from x_i.

```
def Iplus(xi, x):
        if x>= xi: return x - xi
        else: return 0.0
```

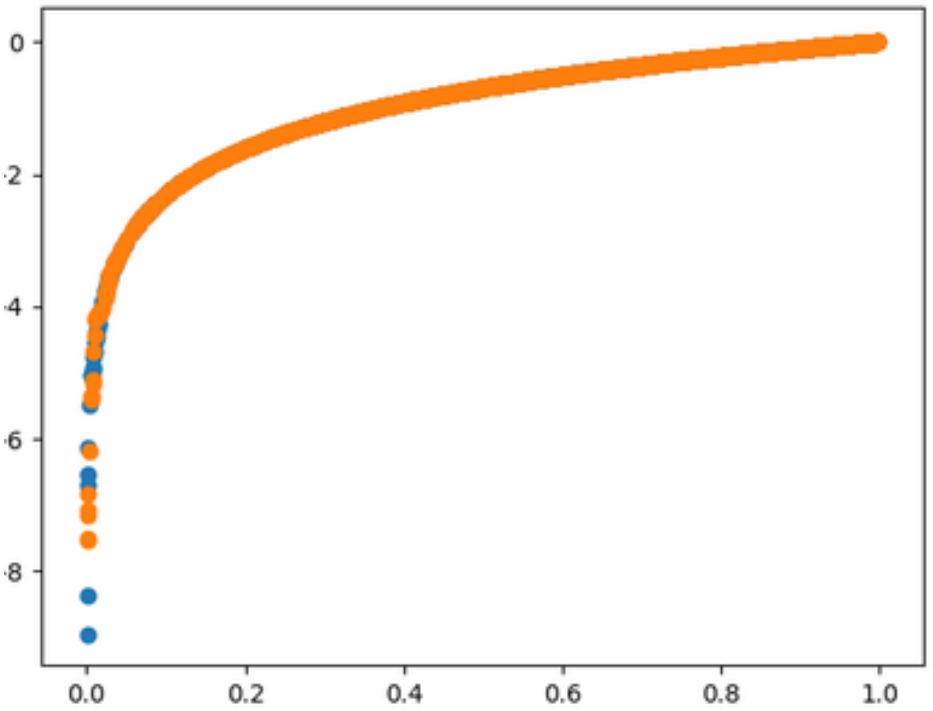
This allows you to start a new line at each control point. Once this function has been defined, the calculation of the ordinate of this spline for a given abscissa is straightforward:

$$y = S(x) = \sum_{i=0}^{n-1} a_i I^i_{plus}(x) + b$$

```
def splinify(xMin, xMax, step, x):
    a = [Iplus(xMin + i *step, x)for i inrange(int((xMax - xMin) / step))]
    a.reverse()
    return a +[1]

np.dot(x, A)
```

Case study

```
import numpy as np
import math
import random
from numpy.linalg import inv
import matplotlib.pyplot as plt

nbSamples =1000

X = np.matrix([[random.random(), 1]for x inrange(nbSamples)])
Y = np.matrix([math.log(x[0].item(0))for x in X]).transpose()

def Iplus(xi, x):
        if x>= xi: return x - xi
        else: return 0.0

def splinify(xMin, xMax, step, x):
        a = [Iplus(xMin + i *step, x)for i inrange(int((xMax - xMin) / step))]
        a.reverse()
        return a +[1]

Xm = np.matrix([splinify(0.0, 1.0, 0.01, x[0].item(0))for x in X])
A = inv(Xm.transpose()*Xm)*Xm.transpose()*Y
Yreg = np.matrix([[np.dot(x, A).item(0)]for x in Xm])

plt.scatter(np.asarray(X[:,0]), np.asarray(Y))
plt.scatter(np.asarray(X[:,0]), np.asarray(Yreg))
plt.show()
```


0
2
4
6
8
0.0
0.2
0.4
0.6
0.8
1.0

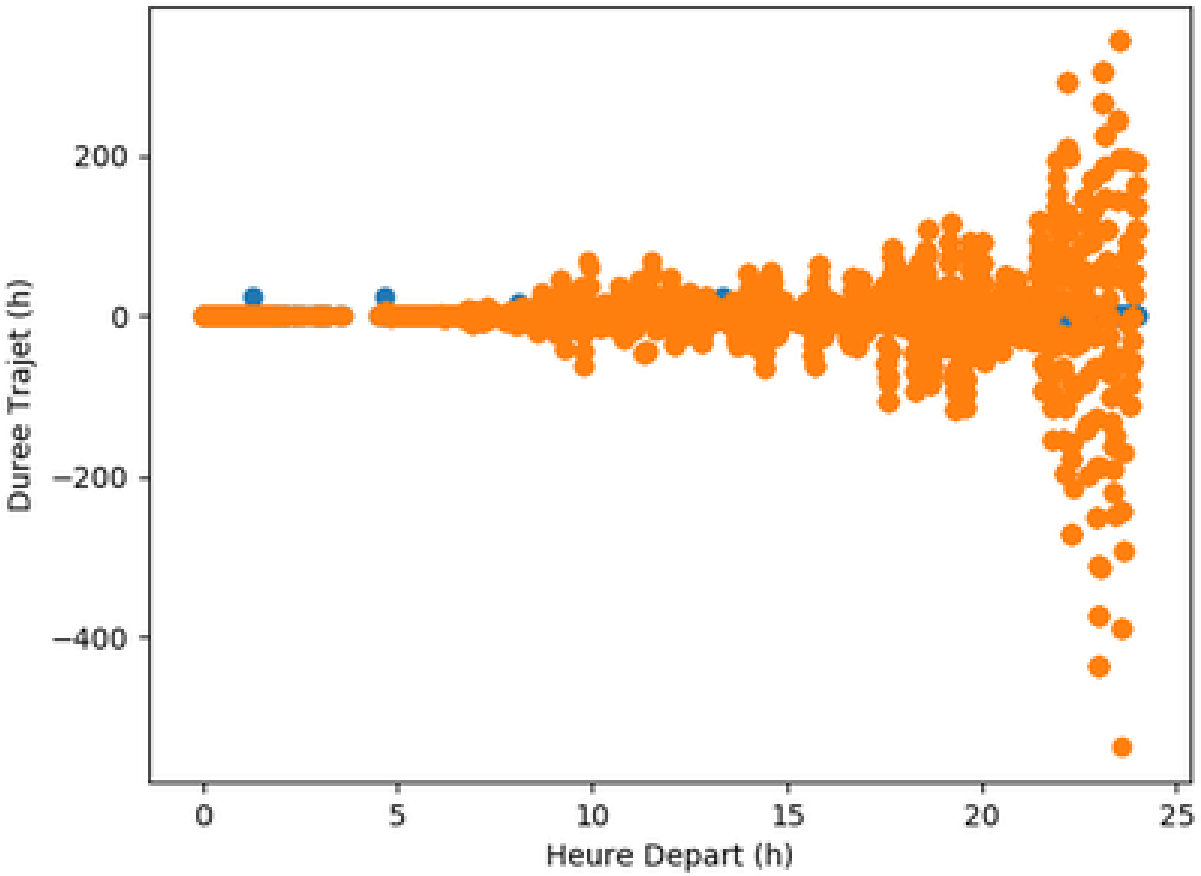
JFK → Upper Manhattan

```
import numpy as np
import math
import random

from numpy.linalg import inv
import pandas as pd
from dateutil importparser
import matplotlib.pyplot as plt

cols =['PULocationID','DOLocationID','tpep_pickup_datetime','tpep_dropoff_datetime','
      trip_distance']

df = pd.read_csv('JFKraw.csv', usecols=cols)


#236 manhattan upper east side
JFK_MU = df[(df['PULocationID']==132)&(df['DOLocationID']==236)]

JFK_MU['weekday']= JFK_MU['tpep_pickup_datetime'].apply(lambda x :parser.parse(x).weekday())

JFK_MU = JFK_MU[JFK_MU['weekday']<5]
```

JFK → Upper Manhattan

```
pu = [parser.parse(dt)for dt in JFK_MU['tpep_pickup_datetime'].values]
do = [parser.parse(dt)for dt in JFK_MU['tpep_dropoff_datetime'].values]
dur = [(b -a).total_seconds()/ 3600.0for a, b inzip(pu, do)]
startTime = [dt.hour+ dt.minute/ 60.0for dt in pu]

X = startTime
Y = dur

def Iplus(xi, x):
        if x>= xi:      return x - xi
        else:   return 0.0

def splinify(xMin, xMax, step, x):
        a = [Iplus(xMin + i *step, x)for i inrange(int((xMax - xMin) / step))]
        a.reverse()
        return a +[1]

Xm = np.matrix([[Iplus(0.5, x), Iplus(0, x), 1]for x in X])

# Find a and b
Xm = np.matrix([splinify(np.min(X), np.max(X), 0.1, x)for x in X])
A = inv(Xm.transpose()*Xm)*Xm.transpose()*np.matrix(Y).transpose()
Yreg = np.matrix([[np.dot(x, A).item(0)]for x in Xm])

plt.scatter(X, np.asarray(Y))
plt.scatter(X, np.asarray(Yreg))
plt.show()
```

JFK → Upper Manhattan



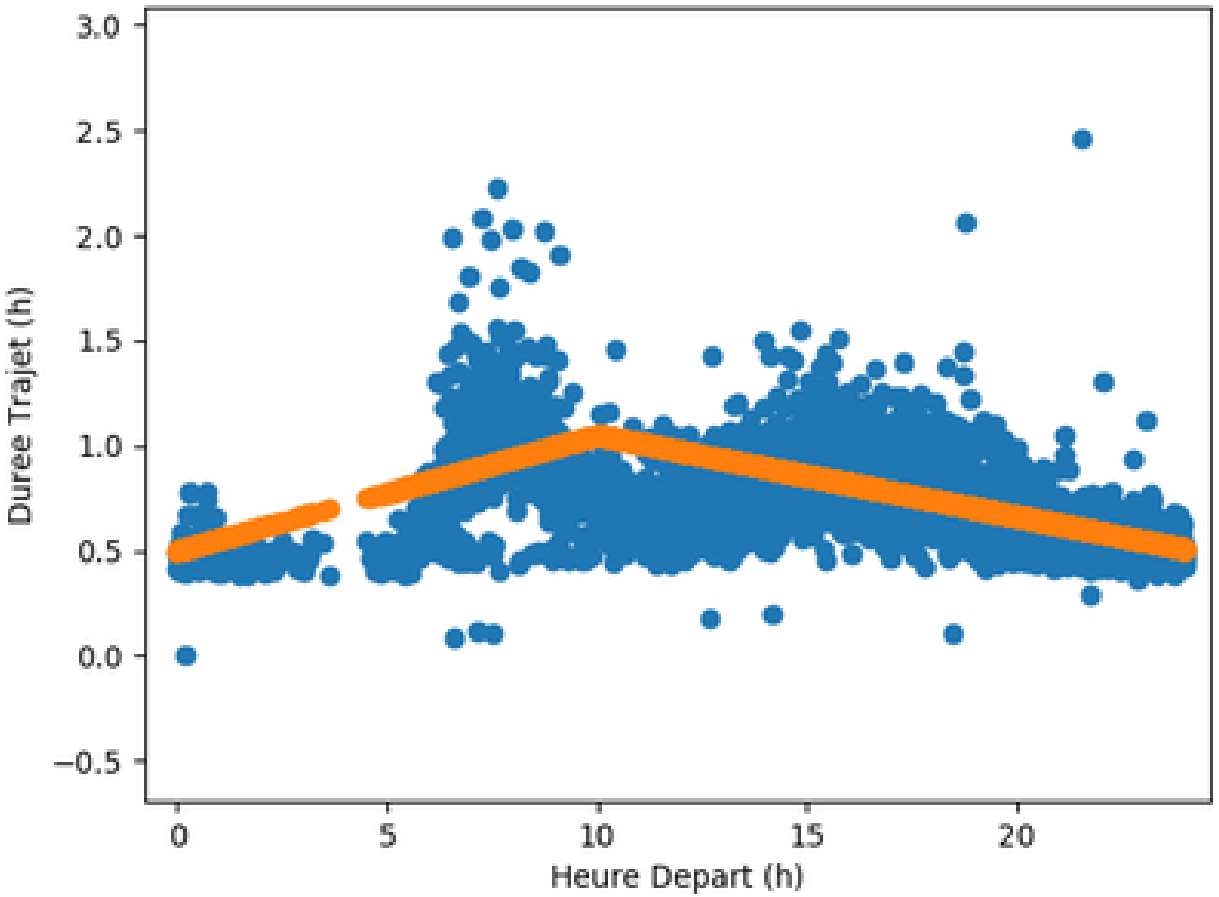
overfitting: essential distinction between learning set and validation set !!

Compromise bias / variance

- step = 0.1 (arbitrary)
- abscissa extending to [0.25]
- our spline is found with no less than 250 nodes.
- large number of degrees of freedom: allows to deform a lot.
- principle of understood bias / variance. That is to say that the data scientist, when he chooses a model for these data, must arbitrate between a too simple model, which would lead to a significant bias, and a model that is too complex, too flexible, that generates too much variance . That's what we just did.

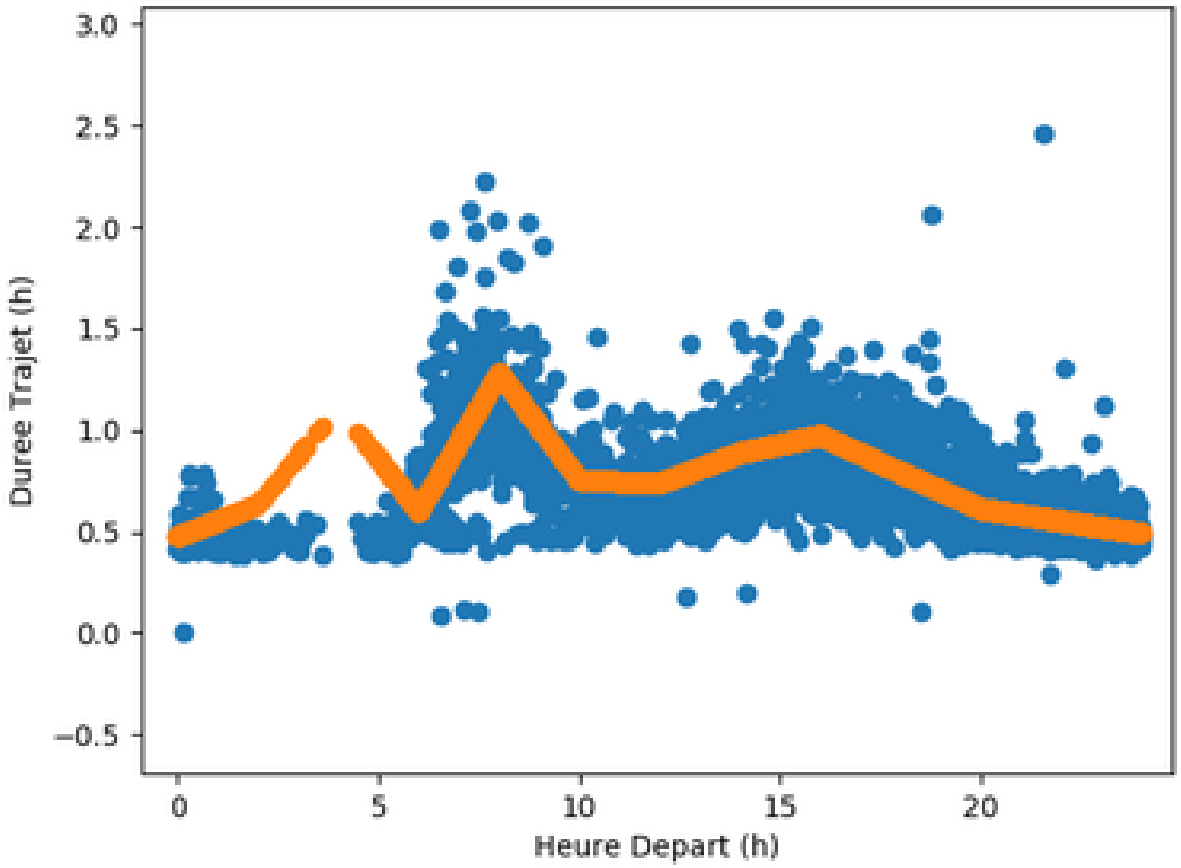
Compromise bias / variance

step = 10 (spline = 3 nodes)



Compromise bias / variance

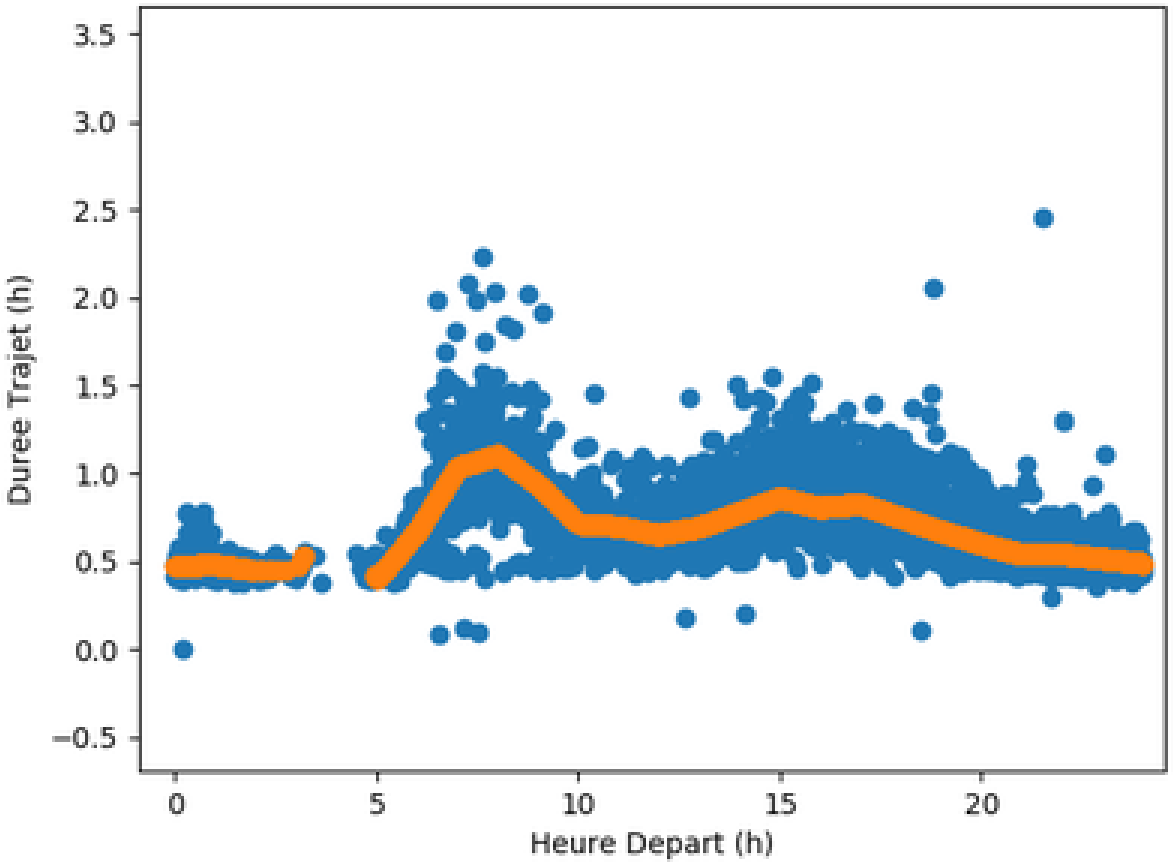
step = 2



Aberrant points

modeling error around 4:20. This error is due to the presence of outliers, which are either measurement errors or extraordinary cases of plugs, failures, etc.

```
# Find a and b
Xm = np.matrix([splinify(np.min(X), np.max(X), 1.0, x)for x in X])
A = inv(Xm.transpose()*Xm)*Xm.transpose()*np.matrix(Y).transpose()
Yreg = np.matrix([[np.dot(x, A).item(0)]for x in Xm])

Yfiltered = [Y[i]for i in range(len(Y)) if ((math.fabs((Y[i]-Yreg[i]) / Y[i]) < 0.9) and (Y[i] >
       0.2) and(Y[i]<2.5))]
Xfiltered = [X[i]for i in range(len(Y)) if ((math.fabs((Y[i]-Yreg[i]) / Y[i]) < 0.9) and (Y[i] >
       0.2) and(Y[i]<2.5))]

Xm = np.matrix([splinify(np.min(Xfiltered), np.max(X), 1.0, x)for x in Xfiltered])
A = inv(Xm.transpose()*Xm)*Xm.transpose()*np.matrix(Yfiltered).transpose()
Yfilteredreg = np.matrix([[np.dot(x, A).item(0)]for x in Xm])

plt.xlabel('Heure␣Depart␣(h)')
plt.ylabel('Duree␣Trajet␣(h)')
plt.scatter(X, np.asarray(Y))
plt.scatter(Xfiltered, np.asarray(Yfilteredreg))
plt.show()
```


Duree Trajet (h)
Heure Depart (h)
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
5
10
15
20

Introduction

- number of variables in a dataset becomes too large.
- precise analysis in each of the dimensions, it takes a set of measures quite gigantic
- difficult for a human to understand the relationships between so many variables.

Example

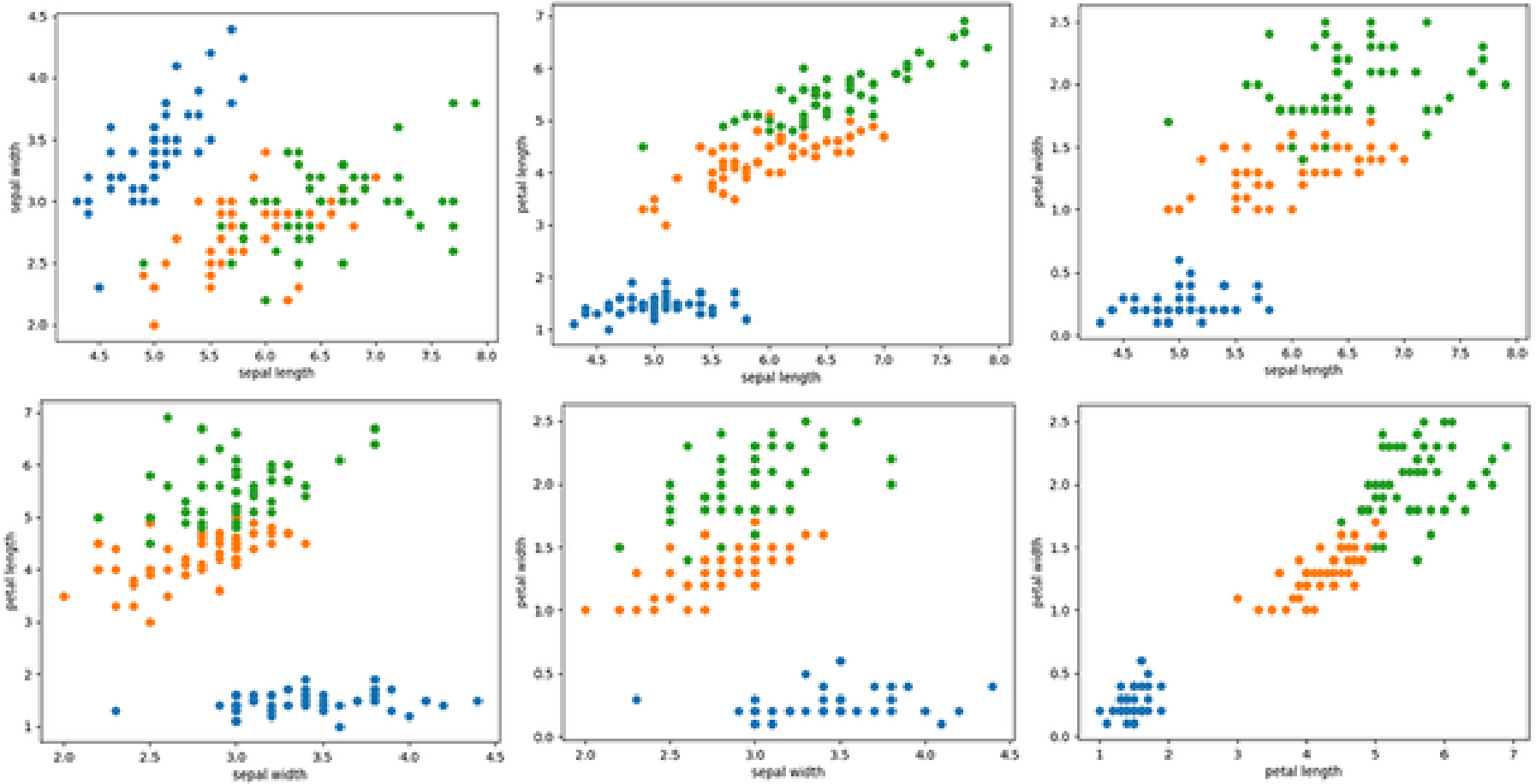
3 different iris species, brings together four different measures:

- the length of the sepals;
- the width of the sepals;
- the length of the petals;
- the width of the petals

Comparisons two by two of the variables of the set

```
import matplotlib.pyplot as plt
from sklearn import datasets
iris = datasets.load_iris()


x = iris.data
y = iris.target

labels =['sepal␣length','sepal␣width','petal␣length','petal␣width']
for xx inrange(4):
        for yy inrange(4):
                if yy> xx:
                print xx, yy
                plt.xlabel(labels[xx])
                plt.ylabel(labels[yy])
                plt.scatter(iris.data[y==0][:, xx], iris.data[y==0][:,yy])
                plt.scatter(iris.data[y==1][:, xx], iris.data[y==1][:,yy])
                plt.scatter(iris.data[y==2][:, xx], iris.data[y==2][:,yy])
                plt.show()
```

Comparisons two by two of the variables of the set

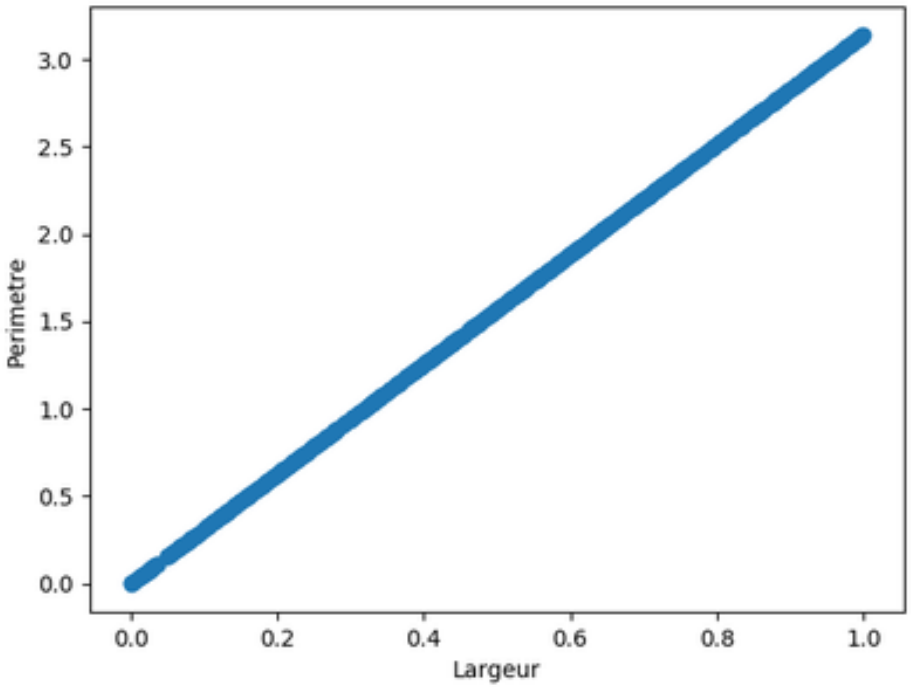


PCA

- Principal component analysis: reduce the size of the studied ensemble by identifying the dimensions that carry the most information
- if one of the predictors has the same value for all samples, then it does not provide any information
- identify the axes that carry the most information, in an orderly manner
- This is almost always a linear combination of predictors.

a simple 2D case

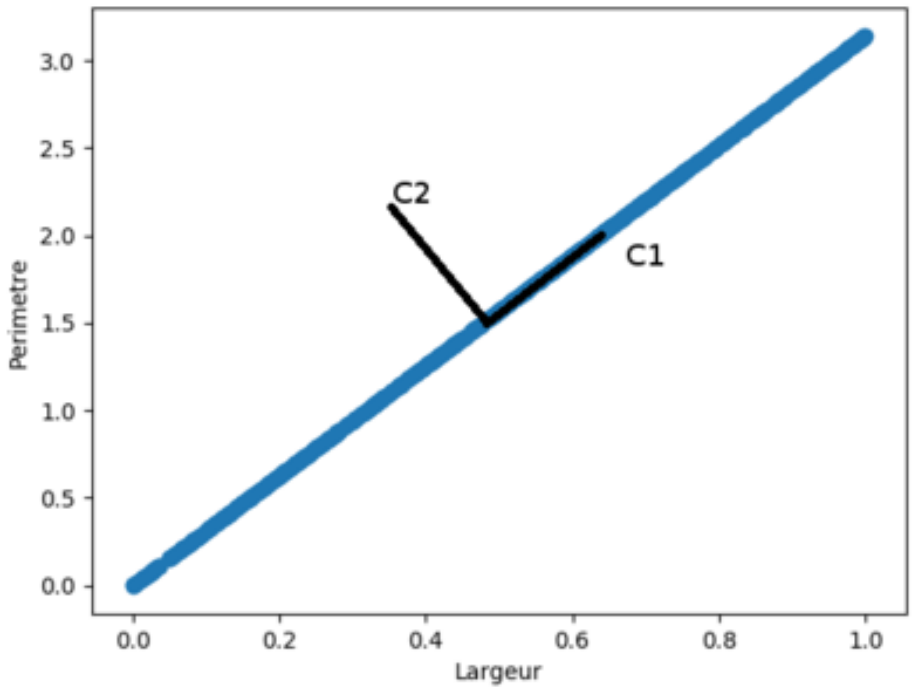
```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
from random importrandom
import numpy as np

nbSamples =1000
X0 = [random()for x inrange(nbSamples)]
X1 = [3.1416*x for x in X0]

plt.scatter(X0, X1)
plt.show()
```



The relationship between our two predictors is clearly linear. By identifying the relationship between them, it is possible to reduce our set to one dimension.

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
from random importrandom
import numpy as np

nbSamples =1000
X0 = [random()for x inrange(nbSamples)]
X1 = [3.1416*x for x in X0]

X = np.matrix((X0, X1)).transpose()
pca = PCA(n_components=2)
pca.fit(X)
print(pca.components_[0])
print(pca.explained_variance_)
```

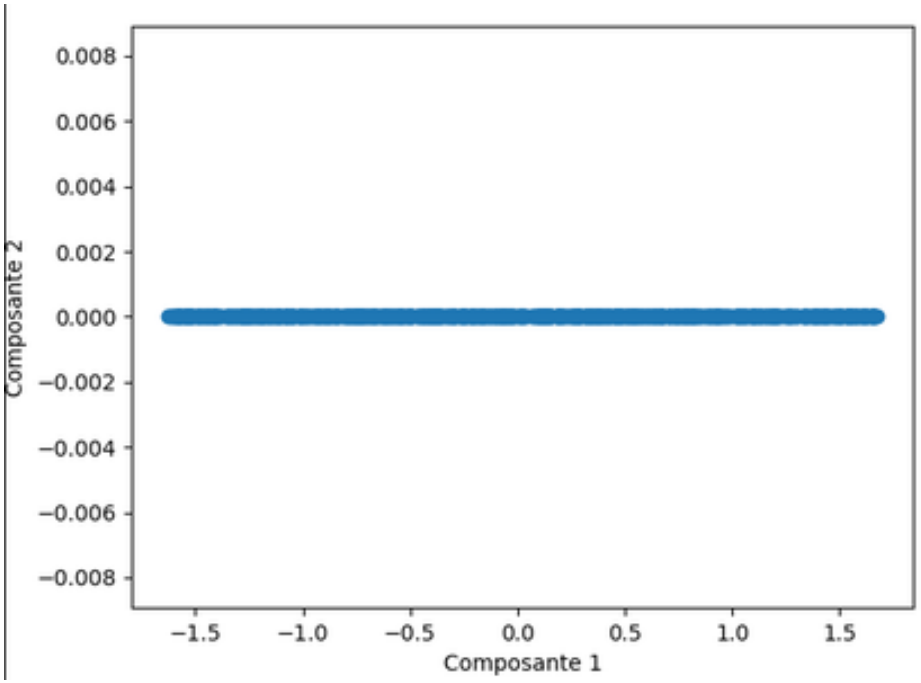
```
[[ 0.30331383 0.95289072]
[-0.95289072 0.30331383]]

[ 3.04402295e+01 2.11846137e-15]


>>> pca.components_[0][1]/ pca.components_[0][0]
3.1416000000000022

>>> np.dot(pca.components_[0], pca.components_[1])
0.0


>>> np.linalg.norm(pca.components_[0])
1.0

>>> np.linalg.norm(pca.components_[1])
1.0
```



The first axis, the one with the greatest eigenvalue, is enough to capture our whole

the matrix presented above can be considered, in the case 2D at least, as a rotation matrix.

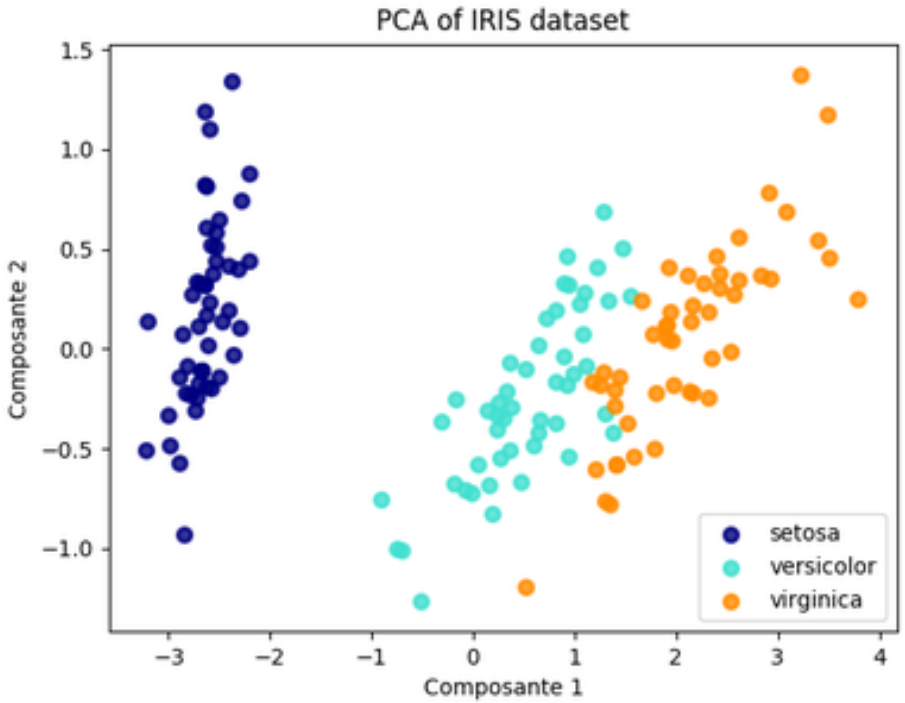
```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
from random importrandom
import numpy as np

nbSamples =1000
X0 = [random()for x inrange(nbSamples)]
X1 = [3.1416*x for x in X0]

X = np.matrix((X0, X1)).transpose()
pca = PCA(n_components=2)
X_r = pca.fit(X).transform(X)
print(pca.components_)
print(pca.singular_values_)

plt.scatter(X_r[:,0], X_r[:,1])
plt.xlabel("Composante␣1")
plt.ylabel("Composante␣2")
plt.show()
```



No doubt, the second dimension of our 2D case definitely does not help.

PCA and iris

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA

iris = datasets.load_iris()
X = iris.data
y = iris.target
target_names = iris.target_names

pca = PCA(n_components=4)
X_r = pca.fit(X).transform(X)

colors =['navy','turquoise','darkorange']
lw =2

for color, i, target_name inzip(colors,[0,1,2], target_names):
plt.scatter(X_r[y == i,0], X_r[y == i,1], color=color, alpha=.8, lw=lw,
label=target_name)
plt.legend(loc='best', shadow=False, scatterpoints=1)
plt.xlabel("Composante 1")
plt.ylabel("Composante 2")
plt.title('PCA of IRIS dataset')
plt.show()
```



Principal component analysis automatically provides a representation that separates the different types of iris.

PCA et iris

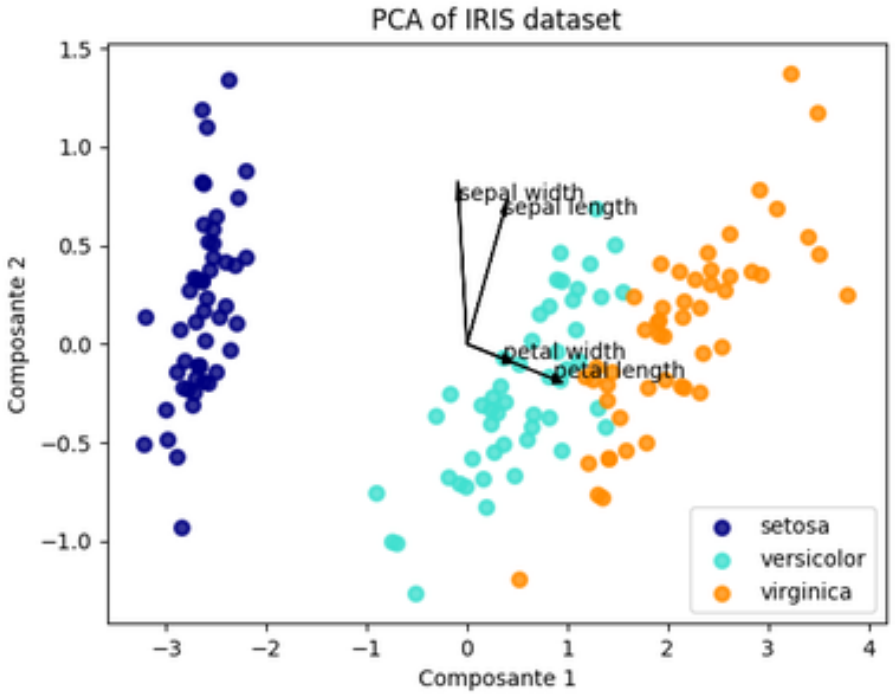
```
>>> print(pca.components_)
[[0.36158968 -0.082268890.856572110.35884393]
[0.656539880.72971237 -0.1757674 -0.07470647]
[-0.580997280.596418090.072524080.54906091]
[0.31725455 -0.32409435 -0.479718990.75112056]]
>>> print(pca.exaplained_variances_)
[25.089863986.007852543.420535381.87850234]
```

a lot of the information is contained in the first dimension

BIPLOT

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA

iris = datasets.load_iris()
X = iris.data
y = iris.target
target_names = iris.target_names
pca = PCA(n_components=4)
X_r = pca.fit(X).transform(X)
colors =['navy','turquoise','darkorange']
lw =2

for color, i, target_name inzip(colors,[0,1,2], target_names):
   plt.scatter(X_r[y == i,0], X_r[y == i,1], color=color, alpha=.8, lw=lw,
label=target_name)
plt.legend(loc='best', shadow=False, scatterpoints=1)
plt.xlabel("Composante 1")
plt.ylabel("Composante 2")
plt.title('PCA of IRIS dataset')
props =["sepal length","sepal width","petal length","petal width"]
for i inrange(4):
   x = pca.components_[0][i]
   y = pca.components_[1][i]
   plt.arrow(0,0, x, y, head_width=0.05, head_length=0.1, fc='k', ec='k')
   plt.text(x, y, props[i])
   plt.show()
```

BIPLOT


PCA of IRIS dataset
Composante 2
Composante 1
sepal width
sepal length
petal width
petal length
setosa
versicolor
virginica
1.5
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3
4

BIPLOT

```
# moyenne de la longueur du petale - setosa
np.std(iris.data[y==0][:,2])
# -> 0.17176728442867112
# moyenne de la longueur du petale - versicolor
np.std(iris.data[y==1][:,2])
# -> 0.4651881339845203
# moyenne de la longueur du petale - virginica
np.std(iris.data[y==2][:,2])
# -> 0.54634787452684397
```

The length of the petals of the setosa is clearly smaller than for versicolor and virginica.

BIPLOT

```
# moyenne de la Largeur du sepale - setosa
np.std(iris.data[y==0][:,1])
# -> 0.37719490982779713
# moyenne de la Largeur du sepale - versicolor
np.std(iris.data[y==1][:,1])
# -> 0.31064449134018135
# moyenne de la Largeur du sepale - virginica
np.std(iris.data[y==2][:,1])
# -> 0.31925538366643091
```

In this case, the values are very close: it is not a good parameter to distinguish the different species.

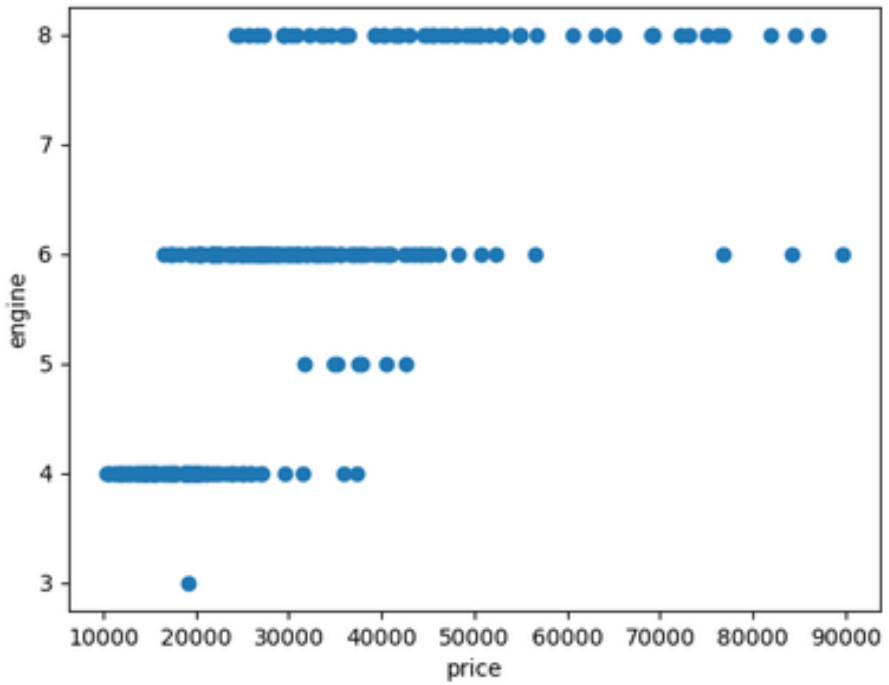
Normalization

- Principal component analysis provides a series of analysis axes that capture the variability of the data studied, in descending order.
- The data thus spread widely along the first axis, while they are fairly condensed around the last one.
- If the data are not normalized, that is, if they have not been reworked in such a way that their averages are zero, and their standard deviations are 1.0, then the analysis may be skewed by differences in units used.

Raw data

```
import pandas as pd
from sklearn import preprocessing
import matplotlib.pyplot as plt

cols =['price','invoice price','dealer cost','engine','cylinders','horsepower','weight','wheel',
       'length','width','cm per gallons','hm per gallons']
df = pd.read_csv('04cars.dat.txt', usecols=cols)

pe = df[df['price']>1000][df['engine']<10][['price','engine']]
plt.scatter(pe['price'], pe['engine'])
plt.xlabel('price')
plt.ylabel('engine')
plt.show()
```

BIPLOT

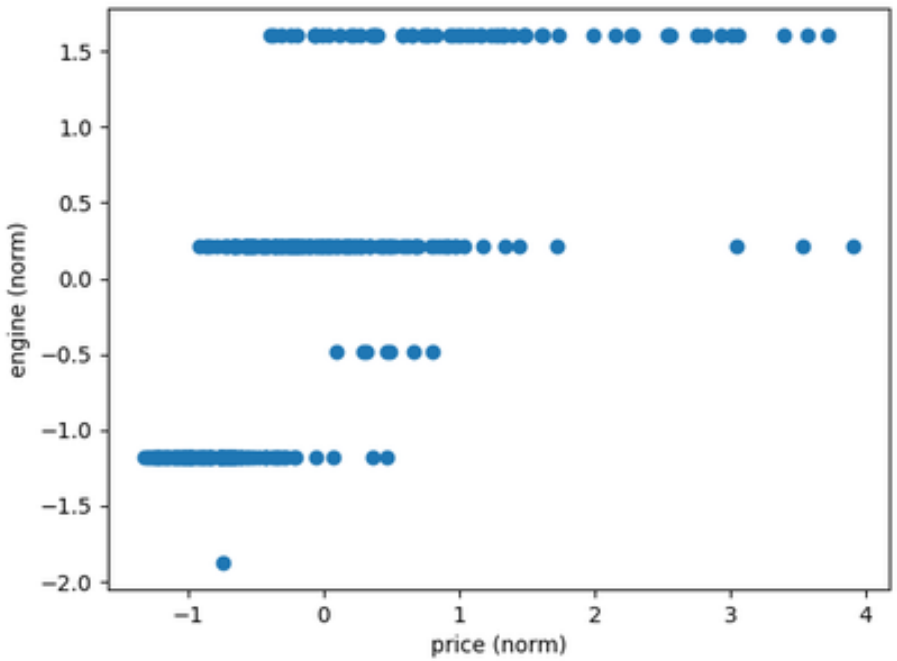

engine
price
8
7
6
5
4
3
10000
20000
30000
40000
50000
60000
70000
80000
90000

Normalized data

Let's normalize our data: a zero mean and a standard deviation of 1

```
import pandas as pd
from sklearn import preprocessing
import matplotlib.pyplot as plt

cols =['price','invoice price','dealer cost','engine','cylinders','horsepower','weight','wheel',
       'length','width','cm per gallons','hm per gallons']
df = pd.read_csv('04cars.dat.txt', usecols=cols)

pe = df[df['price']>1000][df['engine']<10][['price','engine']]
pe_scaled = preprocessing.scale(pe)

plot.scatter(pe_scaled[:,0], pe_scaled[:,1])
plt.xlabel('price (norm)')
plt.ylabel('engine (norm)')
plt.show()
```

Normalized data



Price and displacement of cars, once standardized. These two axes now seem to contain information.

PCA

```
import numpy as np
from sklearn.decomposition import PCA
import pandas as pd
from sklearn import preprocessing
import matplotlib.pyplot as plt

cols =['price','invoice price','dealer cost','engine','cylinders','horsepower','weight','wheel',
      'length','width','cm per gallons','hm per gallons']
df = pd.read_csv('04cars.dat.txt', usecols=cols)
X_scaled = preprocessing.scale(df[cols].replace('*',float('nan')).dropna().as_matrix())
pe = df[df['price']>1000][df['engine']<10][['price','engine']]
pe_scaled = preprocessing.scale(pe)
pca = PCA(n_components=2)

# raw data
pca.fit(pe)
print(pca.explained_variance_)

# normalized data
pca.fit(pe_scaled)
print(pca.explained_variance_)
```

PCA

```
# raw data
[ 2.32179369e+08 1.08822536e+00]

# normalized data
[ 1.69570742 0.31072345]
```

Correlation matrix

- in Python: `C = pe_scaled.transpose()*pe_scaled`
- This matrix contains valuable information: each element *Cij* quantifies the relationship between the variables *i* and *j*. If *Cij* is positive, then when *i* grows, then *j* as well. If, on the other hand, it is negative, then *j* decreases while *i* increases.
- In the case where *Cij* is zero, and that's where it gets interesting, then the variables *i* and *j* are not correlated. They therefore vary independently of each other.
- The particular case where the matrix is diagnonal is therefore particularly sympathetic, because in this case, the variables are all independent of each other.

Data

Cédric Buche

ENIB

September 9, 2019