Dimensionality Reduction

General

Goal: Reduce the dimension to simplify the problem and speed up the machine learning model.

Term:

The Curse of Dimensionality: weird things happen in higher-dimensional space. For example:

1) The probability of choosing a point in a unit square that is 0.001 away from the border is 0.4%. But for a 10000-dimensional unit hypercube, this probability would be 99.999999%

2) The average length of 2 random points in a unit square are 0.52, but in the 1000000-dim hypercube is 408.25

We will verify these facts with python later.

Manifold: a d-dimensional manifold is a part of an n-dimensional space (d<n) that locally resembles a d-dimensional hyperplane. For example: loosely speaking, a leaf is a 2d-manifold in a 3d-world since its local resemble a 2D plane.

Manifold hypothesis/assumption: high dimensional real-world datasets lie close to a much lower-dimensional manifold. The way to select this manifold is different for each data set.

In [120]:
```
#Curse of dimensionality illustration

#1) Choosing a poin in hypercube
import random
random.seed(2)

num_sample=999
num_dim=10000

#1000 points in a unit square
square_pts=np.random.uniform(0,1,size=(num_sample,2))
#1000 points in of unit hypercube
hypercube=np.random.uniform(0,1,size=(num_sample,num_dim))

#Function that count howmany point is in the "restricted area" away the border
def closed2border(points,distance_border):

    #Thought: if one of the coordinate is within the range, then the point is within the restricted area
    count=0
    for point in points:
        for cord in point:
            if cord > (1-distance_border):
                count+=1
                break
    return count

print("Probability of random points in a square:", closed2border(square_pts,0.001)/num_sample)
print("Probability of random points in a {}-dim hypercube:".format(num_dim),
                                closed2border(hypercube,0.001)/num_sample)
```

```
Probability of random points in a square: 0.004004004004004004
Probability of random points in a 10000-dim hypercube: 1.0
```

In [138]:
```
# 2) The average length of 2 random points in a unit square are 0.52, but in the 1000000-dim hypercube is 40
8.25

from scipy.spatial import distance
num_sample=999
num_dim=100000 #I reduced the size cause 1 million dim takes too long

#1000 points in a unit square
square_pts=np.random.uniform(0,1,size=(num_sample,2))
#1000 points in of unit hypercube
hypercube=np.random.uniform(0,1,size=(num_sample,num_dim))


def average_distance(points):
    half_len=int(len(points)/2)
    pairs=list(zip(points[:half_len+1],points[half_len:2*half_len+2]))
    mean_dist = np.mean([distance.euclidean(p1, p2) for p1, p2 in pairs])
    return mean_dist

print("Average distance in a square:", average_distance(square_pts))
print("Average distance in a {}-dimensional hypercube:".format(num_dim), average_distance(hypercube))
```

```
Average distance in a square: 0.5155844529601187
Average distance in a 100000-dimensional hypercube: 129.10576760497972
```

Principal Component Analysis (PCA)

Idea: **Detect the best hyperplane for the data and project the data on our hyperplane**

The "best" hyperplane is the one that can preserve the variances of the data. In other words, this plane can capture the most characteristics of the data.

Note that, a plane can be broken down into **principal components** which are unit vector that is orthogonal to each other. There are many ways to choose **principal components** but we want to choose those that preserve the most variances of the data on its axis. One way to achieve it is to find the **eigenvectors of the covariance matrix**. The explaination for this will be in another paper. The values of the corresponding eigenvalue of each eigenvector will tell how much variances each vector preserves. Finally, we can combine these vector to form a matrix transformation that project data points to a new hyperplane.

Assume that our data has d-dimension, (not counting the label). There are 3 steps in total to achieve our goal:

1) Compute the covariance matrix of the whole dataset.

2) Compute eigenvectors and the corresponding eigenvalues

3) Projection onto the new subspace

1) Compute the covariance matrix of the whole dataset

Assuming we have n features $\{x_1, \dots, x_n\}$, the covariance matrix A is defined so that

$$A_{i,j} = \mathrm{Cov}(x_i, x_j) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x_i})(x_j - \bar{x_j})$$

In [71]:
```
# Sample
import pandas as pd
df=pd.DataFrame({"Math":[90,90,60,60,30],"English":[60,90,60,60,30],"Art":[90,30,60,90,30]})
df.head()
```

Out[71]:

	Math	English	Art
0	90	60	90
1	90	90	30
2	60	60	60
3	60	60	90
4	30	30	30

In [91]:
```
def Cov_entry(xi,xj):
    meani=xi.mean()
    meanj=xj.mean()
    sum_=sum([(xi[index]-meani)*(xj[index]-meanj) for index in range(len(xi))])
    return sum_/5

def Cov(df):
    columns=df.columns
    matrix=[]
    for i in columns:
        row=[]
        for j in columns:
            row.append(Cov_entry(df[i],df[j]))
        matrix.append(row)
    df_new=pd.DataFrame(matrix,columns=columns, index=columns)
    return df_new

Cov(df)
```

Out[91]:

	Math	English	Art
Math	504.0	360.0	180.0
English	360.0	360.0	0.0
Art	180.0	0.0	720.0

The covariance tell the relationship between feature i, j. For example, if a student is good at math, then they are more likely better at English than Art.

2) Compute eigenvectors and the corresponding eigenvalues

An eigenvector is a vector v whose direction remains unchanged when a linear transformation is applied to it, ie:

$$Av = \lambda v$$

Note that computing eigenvector requires another numerical approximation to find root of the **characteristic polynomials**. It is kind of cumbersome to create all the necessary tools to do the task. Hence, I will use numpy. The `np.linalg.eig` function might return a different eigenvector compare to when compute it by hand (Eigenvector is not unique). However, the eigenvalues are the same.

In [73]:
```
import numpy as np

def eig_val_vec(cov_df):
    eig_val, eig_vec = np.linalg.eig(cov_df)
    return eig_val, eig_vec

eig_val, eig_vec=eig_val_vec(Cov(df))
print("Eigen vectors:")
print(eig_vec)
print("Eigen values:")
print(eig_val)
```

```
Eigen vectors:
[[ 0.6487899  -0.65580225 -0.3859988 ]
 [-0.74104991 -0.4291978  -0.51636642]
 [-0.17296443 -0.62105769  0.7644414 ]]
Eigen values:
[ 44.81966028 910.06995304 629.11038668]
```

3) Projection onto the new subspace

Let say we want to project the original data with dimention $d \times n$ to a lower dimensional space $d \times k$ (where $k < n$). There are 2 step

- Sort the eigenvectors by decreasing eigenvalues and choose k eigenvectors with the largest eigenvalues to form a $d \times k$ dimensional matrix W.
- Use this $d \times k$ eigenvector matrix to transform the samples onto the new subspace.

In [146]:
```
#Finalize
def projection(df,k):
    #Get the eigen values and vectors of the covariance matrix of df
    eig_val, eig_vec=eig_val_vec(Cov(df))

    #Sort them
    list_=list(zip(eig_val,eig_vec))
    list_.sort(key=lambda x: x[0], reverse=True)

    #Find the first k-eigenvector
    eig_vec_matrix=[i[1] for i in list_[:k]]
    eig_vec_matrix=np.array(eig_vec_matrix)

    #Transform
    new_data=eig_vec_matrix.dot(df.values.T)
    new_data=pd.DataFrame(new_data.T,columns=np.arange(len(new_data.T[0])))
    return new_data
```

In [148]:
```
projection(df,2)
```

Out[148]:

	0	1
0	-138.919338	15.969466
1	-120.813286	-48.528749
2	-101.196848	-1.774843
3	-116.687840	21.158399
4	-50.598424	-0.887422