Foundation of Machine Learning Coursework 1 Report

Hang Su 30005019

December 3, 2018

1 Classification

1.1 Data 1: Separate 2 Gaussians

First, generating 2 data sets from two 2-dimensional Gaussian distributions

$$x_a \sim N(x|m_a, S_a), x_b \sim N(x|m_b, S_b)$$

where the mean vectors are $m_a = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$, $m_b = \begin{pmatrix} 5 \\ 7 \end{pmatrix}$ and the covariance matrices are $S_a = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$, $S_b = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$.

Each data set contains 200 data points, so $n_a = n_b = 200$. The scatter plot of these data points is shown in figure 1.

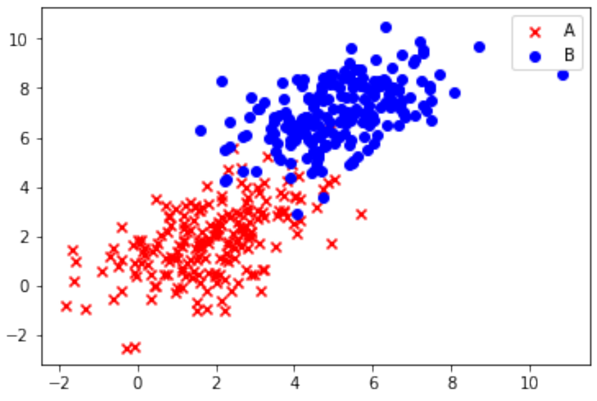


Figure 1: Scatter plot of data points

1.1.1 Explore the consequences of projecting data onto a lower dimension

In this part, giving three initial values to ω, $\omega_1 = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$, $\omega_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $\omega_3 = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$. Then according to $y_c^n = x_c^n \omega, c \in \{a, b\}$,the histograms of the distributions of these data points after projected are shown in figure 2:

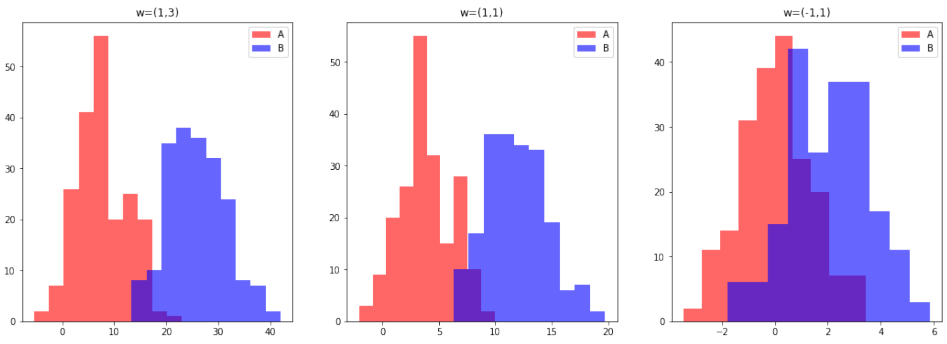


Figure 2: Histograms of projecting data points by different ω

From these three histograms, it is clear that ω_1 and ω_2 can separate type A and type B much better than ω_3. Next, if rotating the initial ω_1 by angle θ, according to the expression of the Fisher ratio and the expression of $R(\theta)$,

$$F(\omega) = \frac{(\mu_a - \mu_b)^2}{\frac{n_a}{n_a+n_b}\sigma_a^2 + \frac{n_b}{n_a+n_b}\sigma_b^2}$$

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

$$\omega(\theta) = R(\theta)\omega(0)$$

the relationship between $F(\omega)$ and rotating angle θ are shown in figure 3.

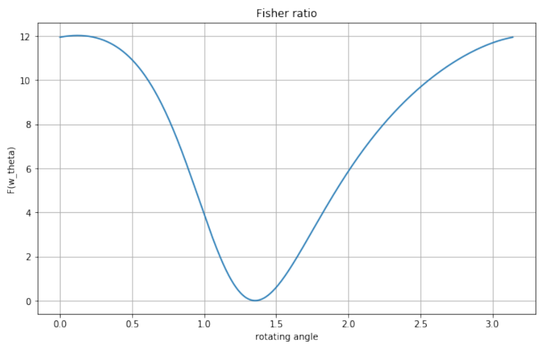


Figure 3: The dependence of $F(\omega)$ on the rotating angle θ

Through iterating $\theta \in (0, \pi)$ 1000 times and comparing the corresponding $F(\theta)$, we can get the optimal $\theta^* = 0.1226$ and the optimal $\omega^* = \begin{pmatrix} 0.6255 \\ 3.0998 \end{pmatrix}$.

1.1.2 Probability distribution

The equi-probable contour plot and the direction of the optimal choice of vector ω^* are shown in figure 4.

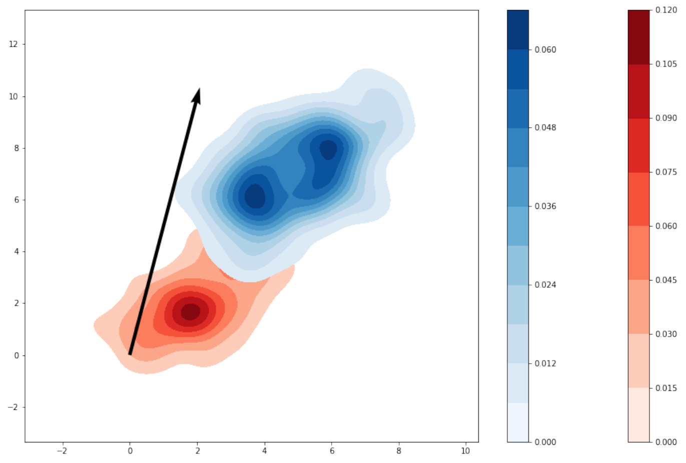
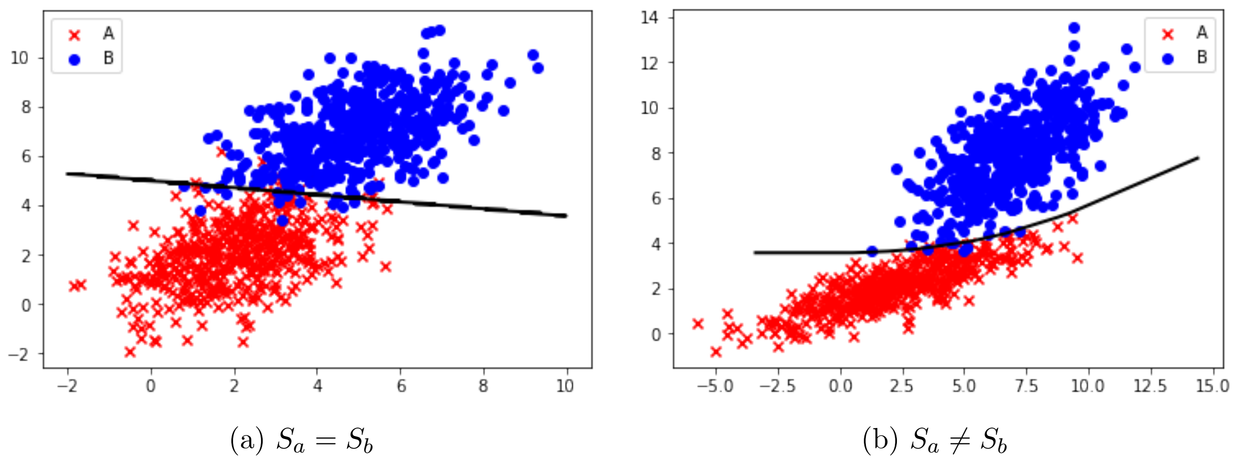


Figure 4: Equi-probable contour and optimal ω^*

According to Bayes' rules:

$$
\begin{aligned}
log-odds &= \ln\left(\frac{P(c=a|x^n)}{P(c=b|x^n)}\right) \\
&= \ln\left(\frac{P(x^n|c=a)P(c=a)}{P(x^n|c=b)P(c=b)}\right) \\
&= \ln\left(\frac{P(x^n|c=a)}{P(x^n|c=b)}\right) + \ln\left(\frac{P(c=a)}{P(c=b)}\right)
\end{aligned}
$$

so when I set $log - odds = 0$, I got a list of data points on the decision boundary. As you can see in figure 5, the shapes of dicision boundaries for $S_a = S_b$ and $S_a \neq S_b$ are different, it is a straight line when $S_a = S_b$ and a curve when $S_a \neq S_b$.



(a) $S_a = S_b$ (b) $S_a \neq S_b$

Figure 5: Decision boundary

1.1.3 Explore the consequence of using unbalanced Fishers' ratio formula

If using the formula

$$F_{unbalanced}(\omega) = \frac{(\mu_a - \mu_b)^2}{\sigma_a^2 + \sigma_b^2}$$

which does not account for the different fractions of data in each class to calculate the optimal ω^*, and the result is $\theta = 0.1478\ \omega^* = \begin{pmatrix} 0.5473 \\ 3.1146 \end{pmatrix}$, almost equivalant to the result got from using the formula $F(\omega)$, which means that the fractions of data in each class has no significant influence in finding the optimal ω^*. However, according to Bayes' rule, $log - odds = \ln\left(\frac{P(x^n|c=a)}{P(x^n|c=b)}\right) + \ln\left(\frac{P(c=a)}{P(c=b)}\right)$, for example, if the fraction of class A is much bigger than the fraction of type B, the value of $\ln\left(\frac{P(c=a)}{P(c=b)}\right)$ will be extremely big/small, even tend to be infinity/negative infinity. Thus the value of $log - odds$ will always bigger/smaller than 0 in this case, which will leads to a result that this model can not separate the data set into two classes.

1.2 Data 2: Iris Data

1.2.1 Find the optimal direction ω^*

To find the optimal direction ω^* which can separate the three classes of iris best, first, calculate the mean vectors of them, and the results are $\mu_0 = \begin{pmatrix} 5.006 \\ 3.418 \\ 1.464 \\ 0.244 \end{pmatrix}$, $\mu_1 = \begin{pmatrix} 5.936 \\ 2.770 \\ 4.260 \\ 1.326 \end{pmatrix}$ and $\mu_2 = \begin{pmatrix} 6.588 \\ 2.974 \\ 5.552 \\ 2.026 \end{pmatrix}$. Next, we can get the within-class covariance matrix Σ_W and the between-class covariance matrix Σ_B according to the formula $\Sigma_B = \sum_c \frac{N_c}{N}(\mu_c - \mu)(\mu_c - \mu)^T$. Entering this step, I got 4 eigenvalues and 4 eigenvectors, the optimal eigenvalue and corresponding eigenvector are $\lambda^* = 0.327$ and $\omega^* = \begin{pmatrix} 0.2536 \\ -0.1204 \\ 0.7233 \\ 0.6309 \end{pmatrix}$ through picking out the maximum eigenvalue.

1.2.2 histograms of the three classes

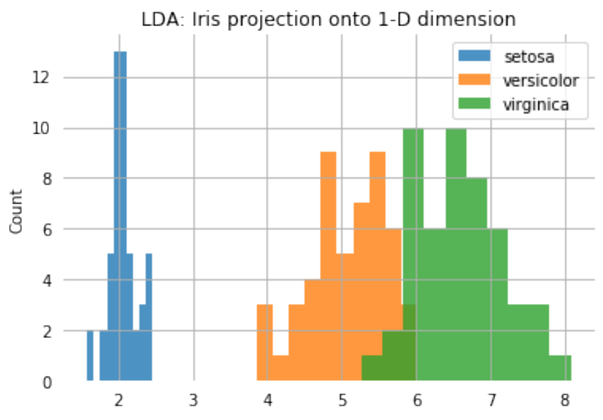
After projecting the data points by the optimal eigenvector ω^*, the histograms of the three classes of iris data in 1-D dimension looks like figure 6(a).

1.2.3 Project on a different vector $\omega = \omega^* + \alpha$

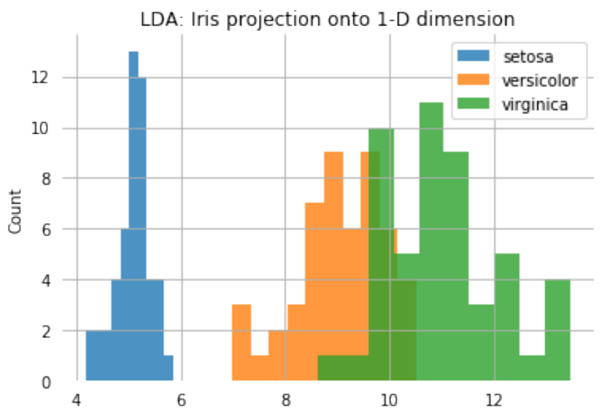
When I use the vector $\omega = \omega^* + \alpha$ (α is another eigenvector), the histogram looks like figure 6(b). It is clear that the area of the overlap part is larger than previous, I think this because of that the direction of eigenvector can retain some features of the meta data points. Thus if I want to project these data on $\omega = \omega^* + \alpha$, I should choose α out of generalised eigenvectors.

1.3 Compare

In the separation 2-gaussian task, the final optimal ω^* is definitely a engeivecter of the design matrix. meanwhile, the dicision boundary vector is orthogonal with the optimal projection vector ω^*.



(a) Project data by $\omega = \omega^*$



(b) Project data by $\omega = \omega^* + \alpha$

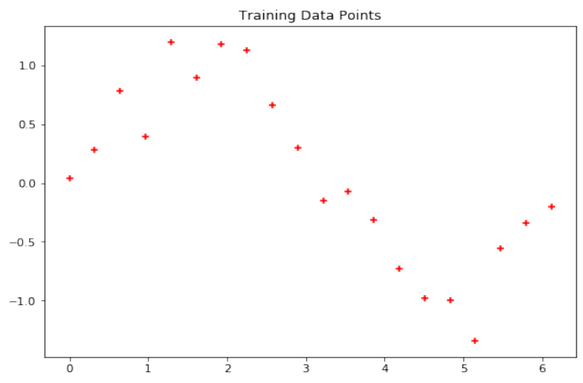
Figure 6: LDA:Iris projection onto 1-D dimension

But I must say that the method using generalised eigenvector is better than the method used in 2-gaussian separation excercise, because rotating angles to find the optimal projection direction only can be used when reducing data into 1-D dimension. In the iris data excercise, I noticed that different eigenvector can explain different proportion of information of data set, which means we need to use 2 or more eigenvectors to do the dimensionality reduction task to reserve as much information as possible. But the rotating way can not achieve this.

2 Linear Regression with non-linear functions

2.1 Performing linear regression

As usual, I generate 40 data points and separate them into training set S_tr and test set S_ts, each of them contains 20 data points. These data points satisfy the condition that $y(x) = \sin(x) + \epsilon, \epsilon \sim N(0, 0.2), x \in [0, 2\pi]$. The scatter plots of the training set and test set are shown in figure 7.



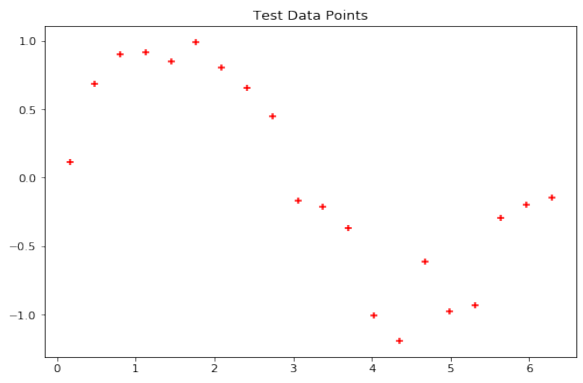


Figure 7: Scatter plot of Training set and Test set

2.1.1 Learn the weights by gradient descent

When trying to fit these data points with polynomial function, I need to find the optimal degree p of weight ω and then minimise the loss function

$$\sum_{n=1}^{N}\left(y^n - \omega_0 - \sum_{j=1}^{p}\omega_j\Phi_j(x^n)\right)^2 + \lambda C(\omega)$$

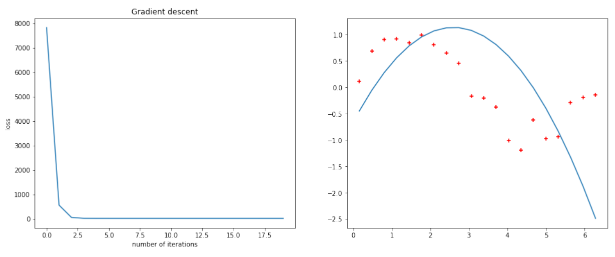
by gradient descent. The expressions used in the processes of gradient descent are as followings:

$$\begin{aligned}(\nabla_\omega L)_i &= \frac{\partial L}{\partial \omega_i}\\ &= 2\sum_{n=1}^{N}\left[y^n - \big(\omega_0 + \sum_{j=1}^{p}\omega_j\Phi_j(x^n)\big)\right]\\ &= -2\sum_{k=1}^{N}A_{ij}\big(y_k - (\omega_0 + \sum_{j=1}^{p}\omega_j A_{ij}\big)\\ &= -2[A^T(y - A_{designmat}\omega)]_i\end{aligned}$$

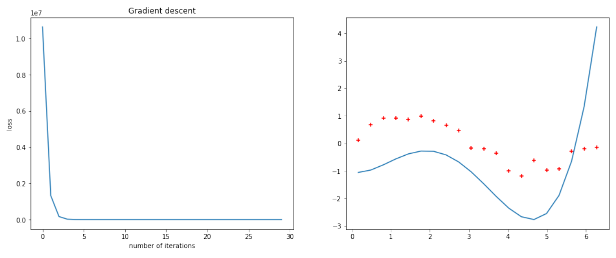
$$C(\omega) = \|\omega\|_2^2 = \sum_{i=1}^{p}\omega_i^2$$

$$\omega_{i+1} = \omega_i - \eta(\nabla_\omega L)$$

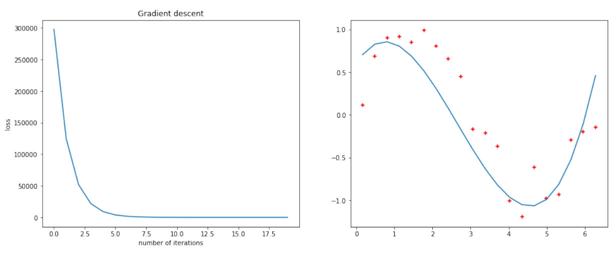
As for the value of p, I tried to give 2 to 5 one by one and then observed the figures to decide which is the best value of p. Through comparing the plots in figure 8, I can roughly say that $p = 3$ is the



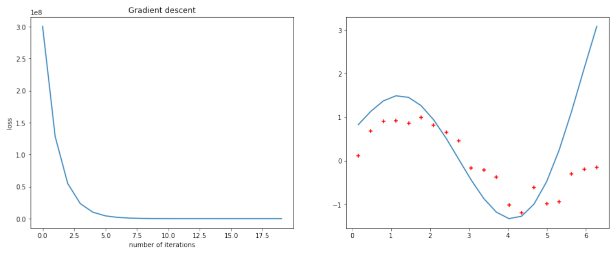
(a) $p = 2, \lambda = 0.5, \eta = 0.0001$



(c) $p = 4, \lambda = 0.5, \eta = 0.00000006$



(b) $p = 3, \lambda = 0.5, \eta = 0.000001$



(d) $p = 5, \lambda = 0.5, \eta = 0.000000001$

Figure 8: Fitting results of different degree of polynomial by gradient descent

optimal value of the degree of the polynomial function. However, when I use gradient descent to find the optimal value of ω, it is hard to find a line which can fit the data points relatively perfect, this means that the gradient descent is easy to fall into local optimum.

2.1.2 Obtain weights from a certain analytical expression

In this part, I plot figures for different values of p. The expressions I used in this parts is:

$$\omega = (X^T X + \lambda I_{p+1})^{-1} X^T y$$

And the fitting results are shown in figure 9. It is obvious that the fitting results are better than those of gradient descent.

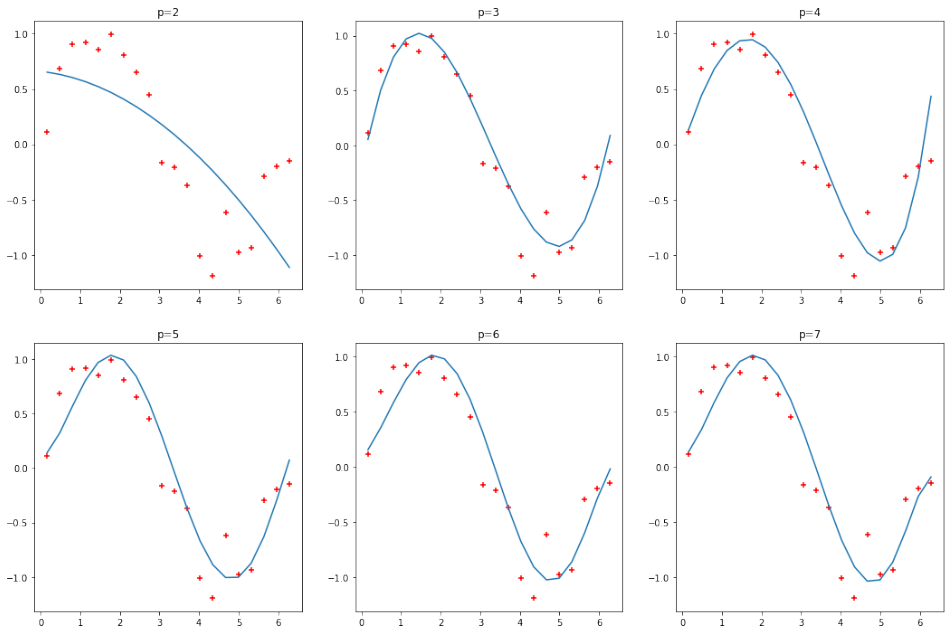


Figure 9: Obtain ω from the analytical expression for different p

2.1.3 Measure the performance of model

The relationship between mean of the squared residuals and the degree(p) of the polynomial are shown in figure 10. Through this line chart, I find that if we fix the value of λ, mean of the squared

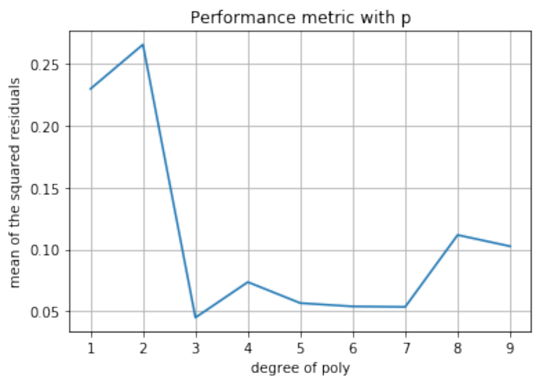


Figure 10: Error of the model with the degree of the polynomial

residuals will reach the bottom at $p = 3$. Moreover, we can observe the strength of the regularisation coefficient λ through the value change of mean of the squared residuals in figure 11. At the beginning, I chose λ from a large range, but I found that the value of the mean of the squared residuals slightly decrease in the range around $(0, 1)$, and then dramatically go up. Thus I zoom out the range of λ to $(0, 1)$, the trends of the error can be clearly seen. From this figure, I can say that the optimal value of λ is approximately 0.5.

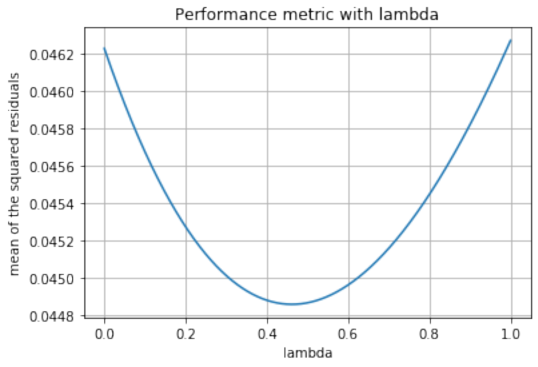
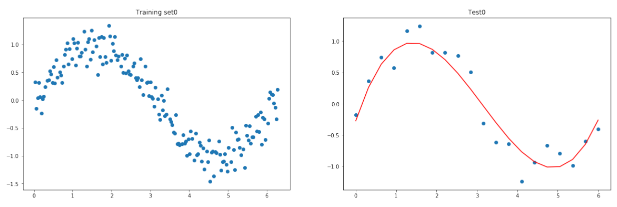


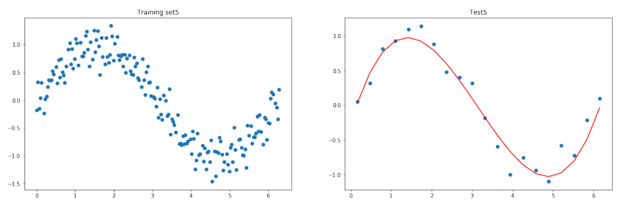
Figure 11: The strength of the regularisation coefficient λ

2.2 How does linear regression generalise?(10-fold cross-validation)

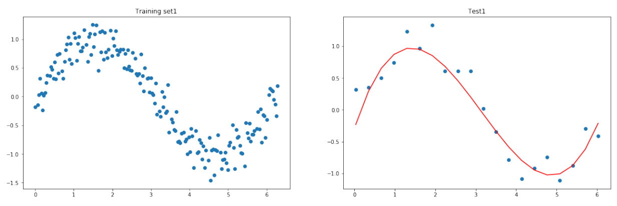
In this part, I generated a new data set which contains 200 data points and seperated them into 10 equal parts. Next, let each of them to be the test set for one time to test the performance of the model got from the other 9 data sets. Meanwhile, I set $p = 3, \lambda = 0.5$ this time, which are the optimal value of them according to the conclusion drewn from 2.1.3. The plots of this ten training sets and the training results can be seen in figure 12. Every iteration generated a value of ω, so I chose the mean value $\omega^* = \begin{pmatrix} -0.2647 \\ 1.9666 \\ -0.9026 \\ 0.0958 \end{pmatrix}$ of these 10 ω as the final optimal value. Finally, I chose the mean of the squared residuals which is around $error = 0.045405$ as the mean error to evaluate the performance of the models.



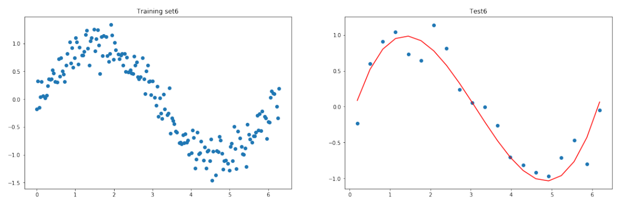
(a) Fold 1



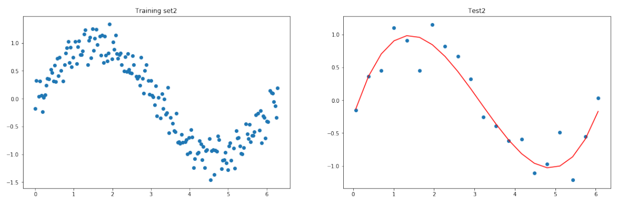
(f) Fold 6



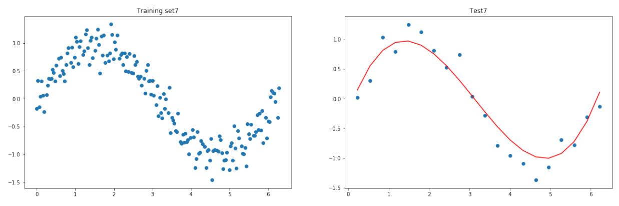
(b) Fold 2



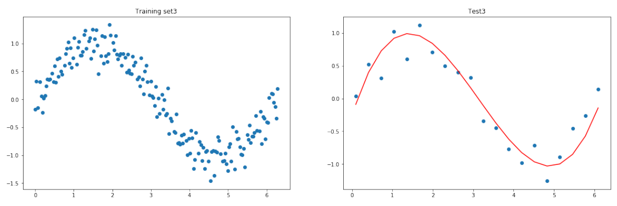
(g) Fold 7



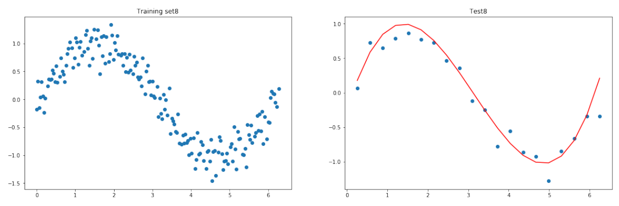
(c) Fold 3



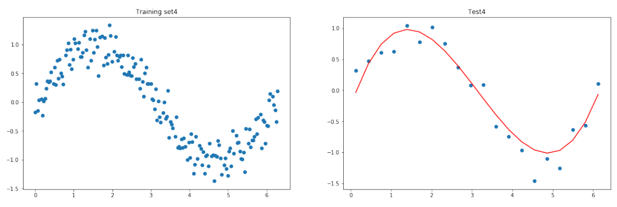
(h) Fold 8



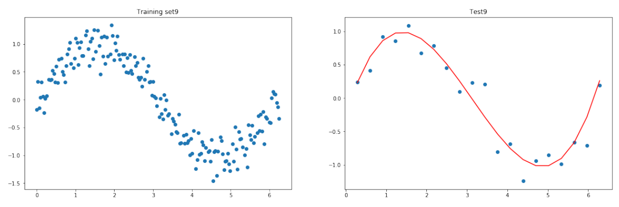
(d) Fold 4



(i) Fold 9



(e) Fold 5



(j) Fold 10

Figure 12: 10-folds cross-vallidation

Cw_3.1

December 3, 2018

1 3.1 Seperate 2 Gaussians

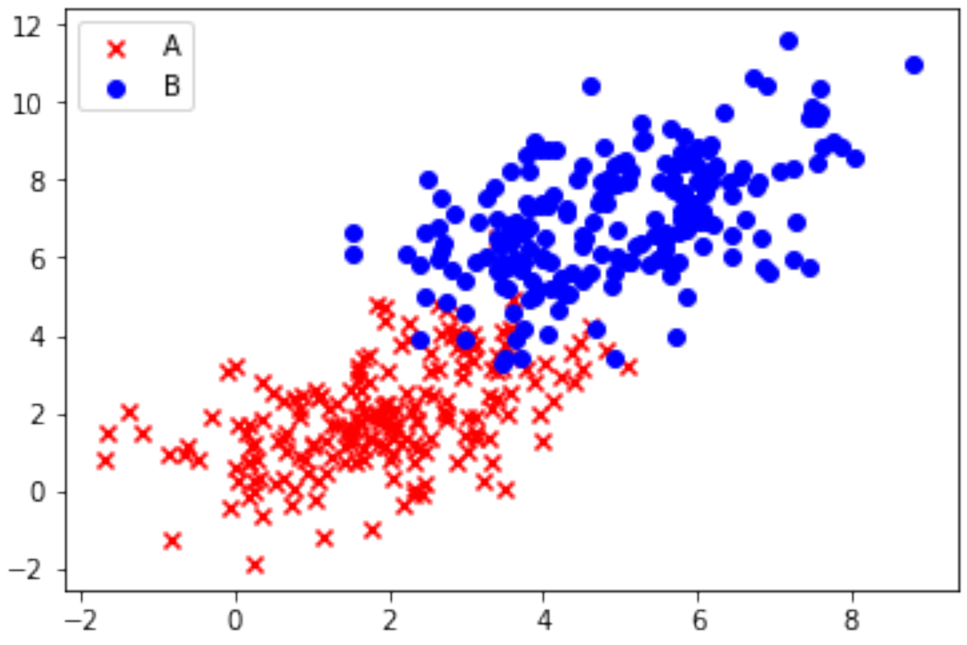
```
In [1]: %matplotlib inline
        import math as math
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
```

1.1 1. Project data onto a lower dimension

```
In [2]: # mean, covariance matrices
        ma = np.array([2,2])
        mb = np.array([5,7])
        sa = np.array([[2,1],[1,2]])
        sb = np.array([[2,1],[1,2]])
        # data points
        xa1, xa2 = np.random.multivariate_normal(ma, sa, 200).T
        xb1, xb2 = np.random.multivariate_normal(mb, sb, 200).T
```

```
In [3]: # plot
        plt.scatter(xa1, xa2, marker='x', color='r', label='A')
        plt.scatter(xb1, xb2, marker='o', color='b', label='B')
        plt.legend()
```

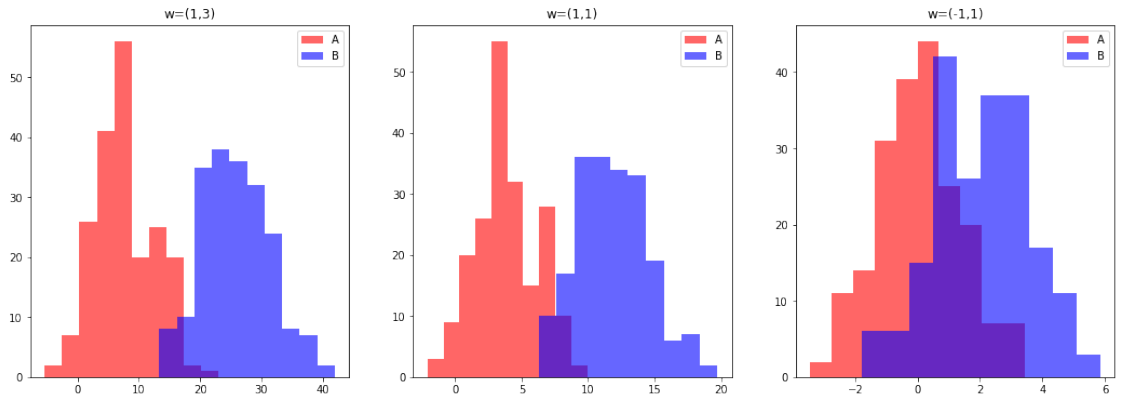
```
Out[3]: <matplotlib.legend.Legend at 0x1190f5a20>
```



- Choose w, calculate y

```
In [4]: # choose 3 random w
        w1 = np.array([1,3])
        w2 = np.array([1,1])
        w3 = np.array([-1,1])
        xa = np.array([xa1,xa2])
        xb = np.array([xb1,xb2])
        ya1 = np.dot(w1,xa)
        yb1 = np.dot(w1,xb)
        ya2 = np.dot(w2,xa)
        yb2 = np.dot(w2,xb)
        ya3 = np.dot(w3,xa)
        yb3 = np.dot(w3,xb)
        fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(18,6))
        ax[0].hist(ya1, color='r', label='A', alpha=0.6)
        ax[0].hist(yb1, color='b', label='B', alpha=0.6)
        ax[0].legend()
        ax[0].set_title("w=(1,3)")
        ax[1].hist(ya2, color='r', label='A', alpha=0.6)
        ax[1].hist(yb2, color='b', label='B', alpha=0.6)
        ax[1].legend()
        ax[1].set_title("w=(1,1)")
        ax[2].hist(ya3, color="r", label='A', alpha=0.6)
        ax[2].hist(yb3, color='b', label='B', alpha=0.6)
        ax[2].legend()
        ax[2].set_title("w=(-1,1)")
```

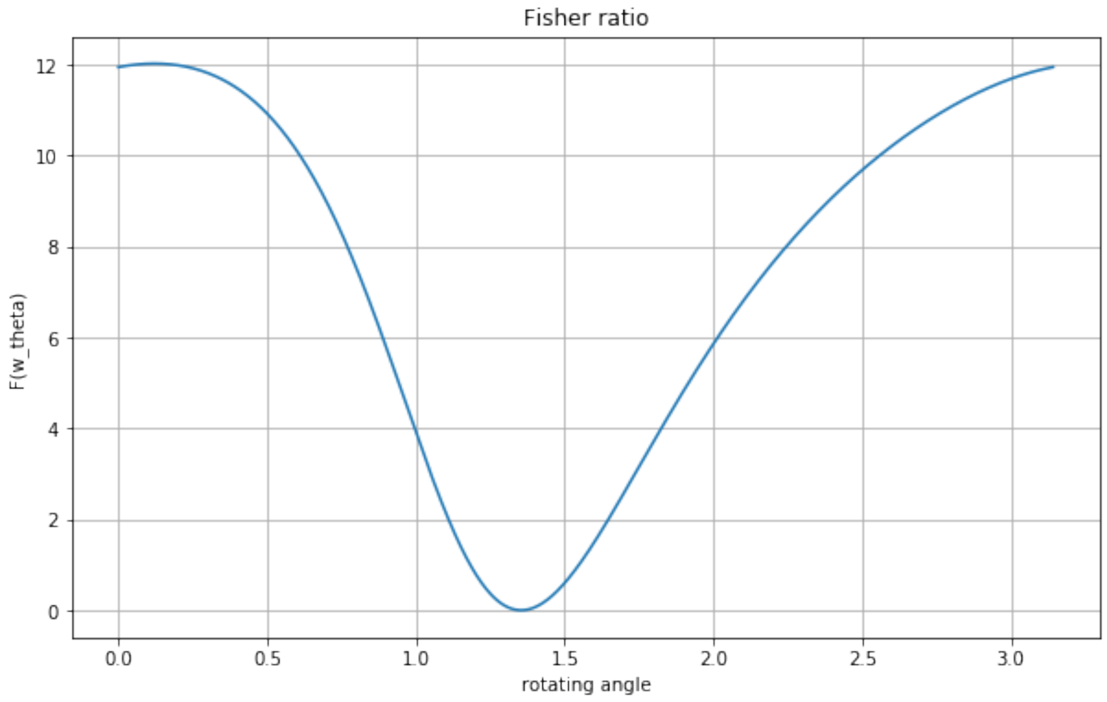
```
Out[4]: Text(0.5,1,'w=(-1,1)')
```



- define F(w)

```
In [5]: def calcul_miu(y):
            n = len(y)
            miu = 1/n * np.sum(y)
            return miu

        def calcul_sigmasqu(y,miu):
            sum_y = 0
            for y_elem in y:
                sum_y += (y_elem-miu)**2
            sigmasqu = 1/len(y) * sum_y
            return sigmasqu

        def F(miu_a,miu_b,sigmasqu_a,sqgmasqu_b,na,nb):
            molecule = (miu_a - miu_b)**2
            denominator = na/(na+nb) * sigmasqu_a + nb/(na+nb) * sigmasqu_b
            return molecule/denominator
```

```
In [6]: thetas = np.linspace(0,math.pi,1000)
        fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(10,6))
        Fw = []
        for theta in thetas:
            R = np.array([[math.cos(theta), -math.sin(theta)],[math.sin(theta),math.cos(theta)]
            w_theta = R.dot(w1)
            y_a1_theta = np.dot(w_theta,xa)
            y_a2_theta = np.dot(w_theta,xb)
            miu_a = calcul_miu(y_a1_theta)
            miu_b = calcul_miu(y_a2_theta)
            sigmasqu_a = calcul_sigmasqu(y_a1_theta,miu_a)
            sigmasqu_b = calcul_sigmasqu(y_a2_theta,miu_b)
            Fw.append(F(miu_a,miu_b,sigmasqu_a,sigmasqu_b,200,200))

        ax.plot(thetas,Fw)
        ax.set_xlabel('rotating angle')
        ax.set_ylabel('F(w_theta)')
        ax.set_title('Fisher ratio')
        plt.grid()
```



- solve the solution of F(w)

```
In [7]: # use w1 to get the solution
        thetas_sol = np.linspace(0,math.pi,1000)
        Fw_sol = []
        for theta in thetas_sol:
            R = np.array([[math.cos(theta), -math.sin(theta)],[math.sin(theta),math.cos(theta)]
            w_theta = R.dot(w1)
            y_a1_theta = np.dot(w_theta,xa)
            y_a2_theta = np.dot(w_theta,xb)
            miu_a = calcul_miu(y_a1_theta)
            miu_b = calcul_miu(y_a2_theta)
            sigmasqu_a = calcul_sigmasqu(y_a1_theta,miu_a)
            sigmasqu_b = calcul_sigmasqu(y_a2_theta,miu_b)
            Fw_sol.append(F(miu_a,miu_b,sigmasqu_a,sigmasqu_b,200,200))
        index = Fw_sol.index(max(Fw_sol))
        theta_star = thetas_sol[index]
        print(theta_star)
        R_star = np.array([[math.cos(theta_star), -math.sin(theta_star)],[math.sin(theta_star)
        w_star = R_star.dot(w1)
        print(w_star)
```

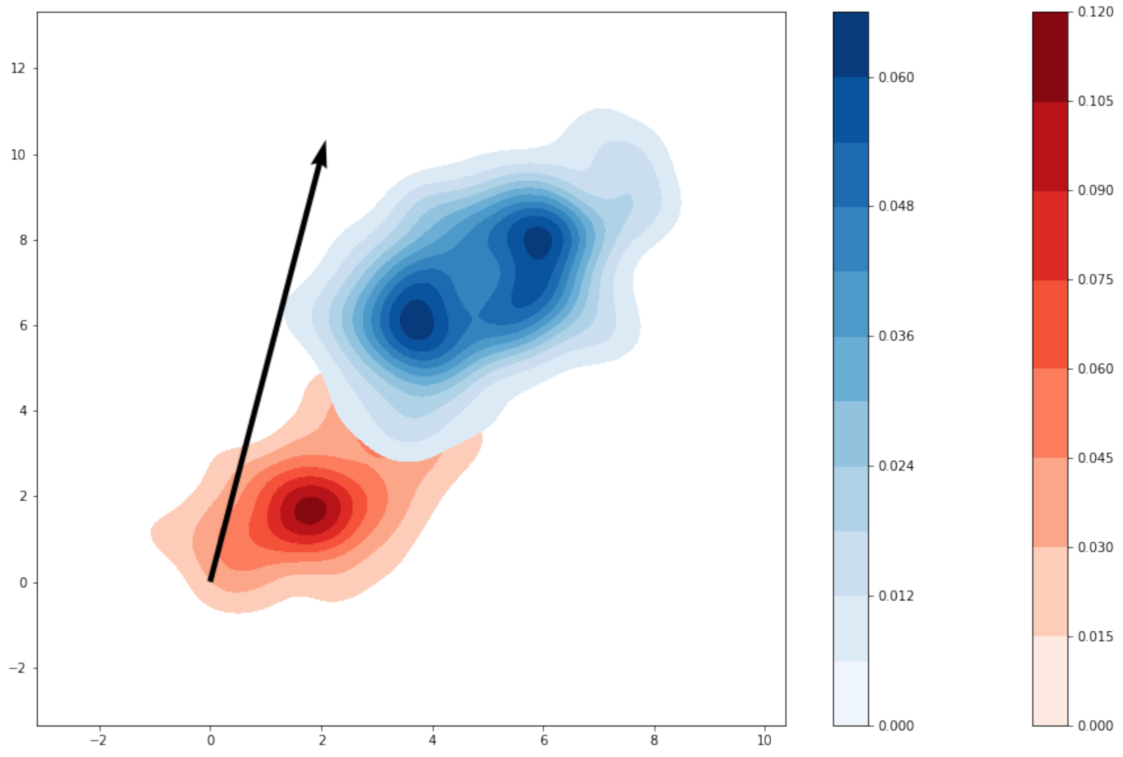
```
0.122644758248
[ 0.62547598  3.09980319]
```

1.2 2. Probability distribution, contour plots

1.2.1 2.(a)

```
In [8]: # define function of distribution
        import seaborn as sns
        fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(16,10))
        sns.kdeplot(xa1, xa2, cmap='Reds', shade=True, shade_lowest=False, cbar=True, ax=ax)
        sns.kdeplot(xb1, xb2, cmap='Blues', shade=True, shade_lowest=False, cbar=True, ax=ax)
        X=[0]
        Y=[0]
        U=w_star[0]
        V=w_star[1]
        ax.quiver(X,Y,U,V,angles='xy', scale_units='xy', scale=0.3)
```

```
Out[8]: <matplotlib.quiver.Quiver at 0x1a1b639390>
```



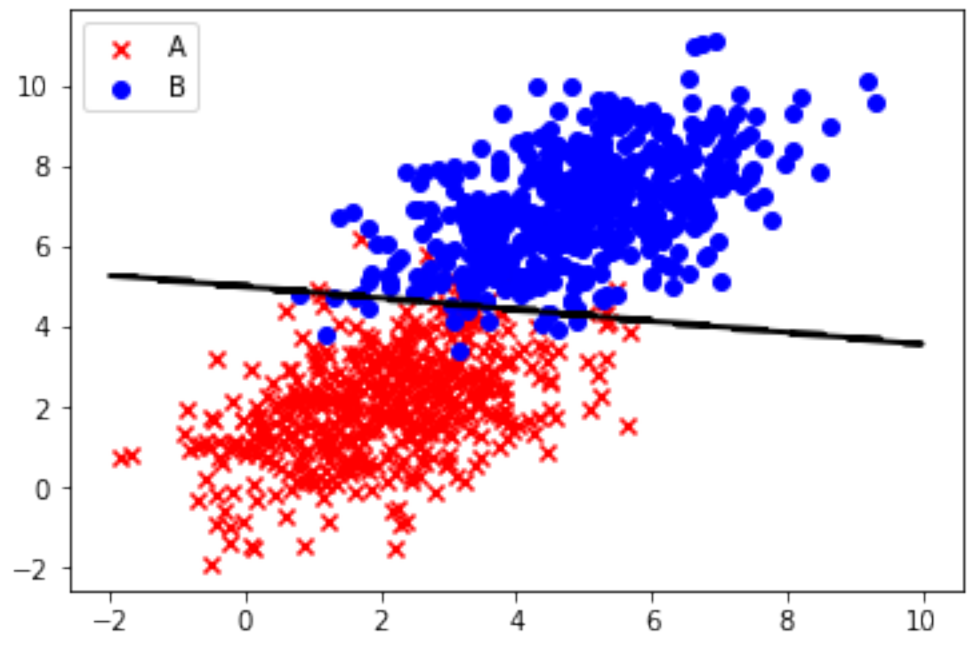
1.2.2 2.(b)

```
In [10]: # equal covariance
         # define calcul
         def prob(m,s,x):
             det_s = np.linalg.det(s)
             inv_s = np.linalg.inv(s)
             p = math.sqrt(det_s)/(2*math.pi) * math.exp(-1/2 * np.dot(np.dot((x-m), inv_s), (
             return p

         def calcul_log(x):
             pa = prob(ma,sa,x)
             pb = prob(mb,sb,x)
             return math.log(pa/pb)
```

- Sa = Sb

```
In [11]: ma = np.array([2,2])
         mb = np.array([5,7])
         sa = np.array([[2,1],[1,2]])
         sb = np.array([[2,1],[1,2]])
         xa1, xa2 = np.random.multivariate_normal(ma, sa, 500).T
         xb1, xb2 = np.random.multivariate_normal(mb, sb, 500).T
         plt.scatter(xa1, xa2, marker='x', color='r', label='A')
         plt.scatter(xb1, xb2, marker='o', color='b', label='B')
         plt.legend()

         #generate grid
         x1_range = np.linspace(-2,10,100)
         x2_range = np.linspace(-2,10,100)
         X1, X2 = np.meshgrid(x1_range,x2_range)
         Xa = []
         for i in zip(X1.flat, X2.flat):
             Xa.append(np.array(i))


         decision_boundary_x1 = []
         decision_boundary_x2 = []
         for xa in Xa:
             # print(calcul_log(xa))
             if (-0.1< calcul_log(xa)<0.1):
                 decision_boundary_x1.append(xa[0])
                 decision_boundary_x2.append(xa[1])

         plt.plot(decision_boundary_x1,decision_boundary_x2,color='black', linewidth=2)
```

```
Out[11]: [<matplotlib.lines.Line2D at 0x1a1ba48f28>]
```

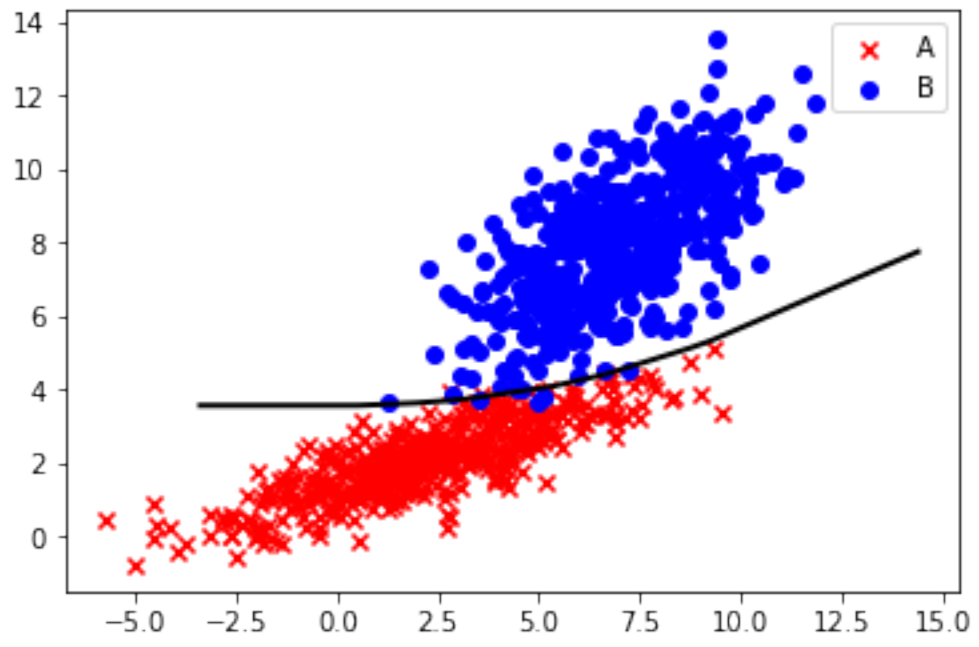


- Sa Sb

```
In [12]: ma = np.array([2,2])
         mb = np.array([7,8])
         sa = np.array([[6,2],[2,1]])
         sb = np.array([[3,2],[2,3]])
         xa1, xa2 = np.random.multivariate_normal(ma, sa, 500).T
         xb1, xb2 = np.random.multivariate_normal(mb, sb, 500).T
         plt.scatter(xa1, xa2, marker='x', color='r', label='A')
         plt.scatter(xb1, xb2, marker='o', color='b', label='B')
         plt.legend()

         #generate grid
         x1_range = np.linspace(-5,15,200)
         x2_range = np.linspace(2,15,200)
         X1, X2 = np.meshgrid(x1_range,x2_range)
         Xa = []
         for i in zip(X1.flat, X2.flat):
             Xa.append(np.array(i))

         decision_boundary_x1 = []
         decision_boundary_x2 = []
         for xa in Xa:
         #      print(calcul_log(xa))
```

```
        if (-0.01< calcul_log(xa)<0.01):
            decision_boundary_x1.append(xa[0])
            decision_boundary_x2.append(xa[1])

plt.plot(decision_boundary_x1,decision_boundary_x2,color='black',linewidth=2)
```

```
Out[12]: [<matplotlib.lines.Line2D at 0x1a1bad3eb8>]
```

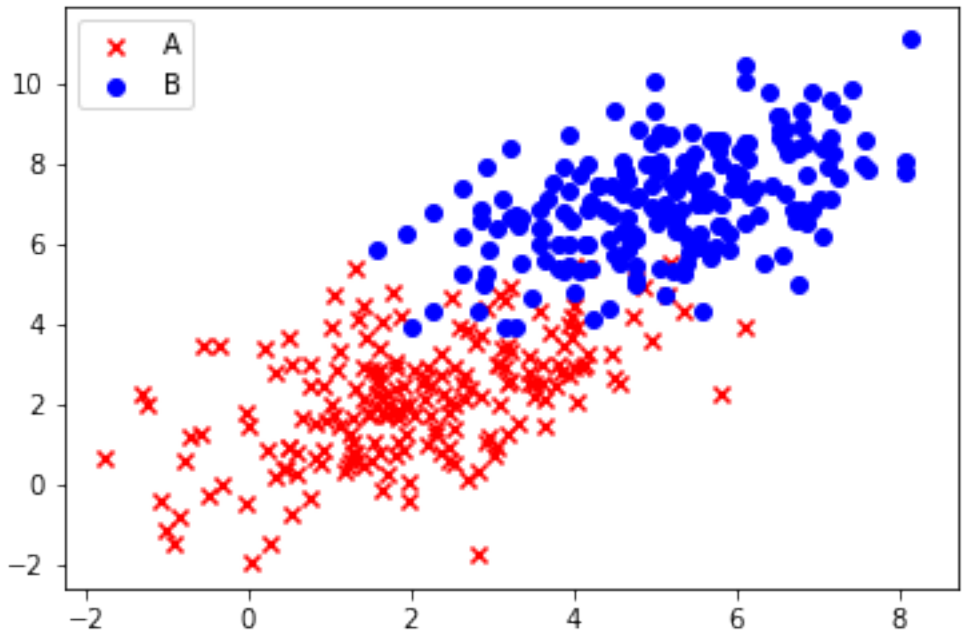


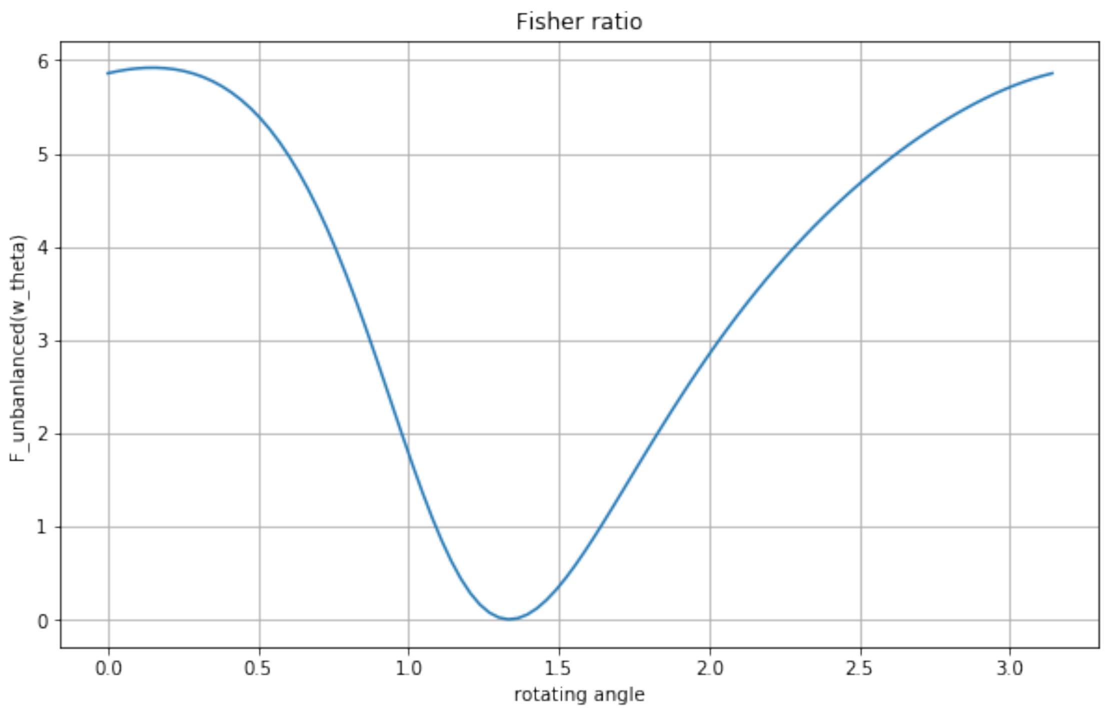
1.2.3 2. (c)

```
In [13]: def F_unbalanced_func(miu_a,miu_b,sigmasqu_a,sigmasqu_b):
             molecule = (miu_a - miu_b)**2
             denominator = sigmasqu_a + sigmasqu_b
             return molecule/denominator
```

```
In [14]: ma = np.array([2,2])
         mb = np.array([5,7])
         sa = np.array([[2,1],[1,2]])
         sb = np.array([[2,1],[1,2]])

         xa1, xa2 = np.random.multivariate_normal(ma, sa, 200).T
         xb1, xb2 = np.random.multivariate_normal(mb, sb, 200).T
         plt.scatter(xa1, xa2, marker='x', color='r', label='A')
         plt.scatter(xb1, xb2, marker='o', color='b', label='B')
         plt.legend()
```

```
w1 = np.array([1,3])
xa = np.array([xa1,xa2])
xb = np.array([xb1,xb2])

thetas = np.linspace(0,math.pi,100)

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(10,6))

F_unbanlanced = []
for theta in thetas:
    R = np.array([[math.cos(theta), -math.sin(theta)],[math.sin(theta),math.cos(theta
    w_theta = R.dot(w1)
    y_a1_theta = np.dot(w_theta,xa)
    y_a2_theta = np.dot(w_theta,xb)
    miu_a = calcul_miu(y_a1_theta)
    miu_b = calcul_miu(y_a2_theta)
    sigmasqu_a = calcul_sigmasqu(y_a1_theta,miu_a)
    sigmasqu_b = calcul_sigmasqu(y_a2_theta,miu_b)
    F_unbanlanced.append(F_unbalanced_func(miu_a,miu_b,sigmasqu_a,sigmasqu_b))

ax.plot(thetas,F_unbanlanced)
ax.set_xlabel('rotating angle')
ax.set_ylabel('F_unbanlanced(w_theta)')
ax.set_title('Fisher ratio')
plt.grid()
```





```
In [15]: # use w1 to get the solution
         thetas_sol = np.linspace(0,math.pi,1000)
         Fw_sol = []
         for theta in thetas_sol:
             R = np.array([[math.cos(theta), -math.sin(theta)],[math.sin(theta),math.cos(theta
             w_theta = R.dot(w1)
             y_a1_theta = np.dot(w_theta,xa)
             y_a2_theta = np.dot(w_theta,xb)
             miu_a = calcul_miu(y_a1_theta)
             miu_b = calcul_miu(y_a2_theta)
             sigmasqu_a = calcul_sigmasqu(y_a1_theta,miu_a)
             sigmasqu_b = calcul_sigmasqu(y_a2_theta,miu_b)
             Fw_sol.append(F_unbalanced_func(miu_a,miu_b,sigmasqu_a,sigmasqu_b))

         index = Fw_sol.index(max(Fw_sol))
         theta_star = thetas_sol[index]
         print(theta_star)
         R_star = np.array([[math.cos(theta_star), -math.sin(theta_star)],[math.sin(theta_star
         w_star = R_star.dot(w1)
         print(w_star)
```

```
0.147802657376
[ 0.54730174  3.11455628]
```

Cw_3.2

December 3, 2018

1 3.2 Iris Data

```
In [1]: %matplotlib inline
        import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from sklearn.datasets import load_iris

In [2]: # load data
        iris_dataset = load_iris()

In [3]: # check keys in dataset
        print("Keys of iris_dataset:\n{}".format(iris_dataset.keys()))
```

```
Keys of iris_dataset:
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names'])
```

```
In [4]: # check the discription
        print('DESCR of iris_dataset:\n{}'.format(iris_dataset['DESCR']))
```

```
DESCR of iris_dataset:
Iris Plants Database
====================

Notes
-----
Data Set Characteristics:
    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
```

```
              - Iris-Virginica
    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20  0.76     0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988

This is a copy of UCI ML iris datasets.
http://archive.ics.uci.edu/ml/datasets/Iris

The famous Iris database, first used by Sir R.A Fisher

This is perhaps the best known database to be found in the
pattern recognition literature.  Fisher's paper is a classic in the field and
is referenced frequently to this day.  (See Duda & Hart, for example.)  The
data set contains 3 classes of 50 instances each, where each class refers to a
type of iris plant.  One class is linearly separable from the other 2; the
latter are NOT linearly separable from each other.

References
----------
   - Fisher,R.A. "The use of multiple measurements in taxonomic problems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
     Mathematical Statistics" (John Wiley, NY, 1950).
   - Duda,R.O., & Hart,P.E. (1973) Pattern Classification and Scene Analysis.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
     Structure and Classification Rule for Recognition in Partially Exposed
     Environments".  IEEE Transactions on Pattern Analysis and Machine
     Intelligence, Vol. PAMI-2, No. 1, 67-71.
   - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions
     on Information Theory, May 1972, 431-433.
   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II
     conceptual clustering system finds 3 classes in the data.
   - Many, many more ...
```

```
In [5]: print('Feature names of iris_dataset:\n{}'.format(iris_dataset['feature_names']))

Feature names of iris_dataset:
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']


In [6]: print('data of iris_dataset:\n{}'.format(iris_dataset['data'][:5]))
        print('shape of iris_dataset:\n{}'.format(iris_dataset['data'].shape))

data of iris_dataset:
[[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]
 [ 4.7  3.2  1.3  0.2]
 [ 4.6  3.1  1.5  0.2]
 [ 5.   3.6  1.4  0.2]]
shape of iris_dataset:
(150, 4)


In [7]: print('target_names of iris_dataset:\n{}'.format(iris_dataset['target_names']))

target_names of iris_dataset:
['setosa' 'versicolor' 'virginica']


In [8]: #
        iris = iris_dataset.data
        #
        target_names = iris_dataset.target_names

In [9]: #
        target = iris_dataset.target
        print(target)

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]


In [10]: #
         mean_vectors = []
         for cl in range(0,3):
             mean_vectors.append(np.mean(iris[target==cl], axis=0))
             print('Mean Vector of class %s: %s\n' %(cl, mean_vectors[cl]))

Mean Vector of class 0: [ 5.006  3.418  1.464  0.244]
Mean Vector of class 1: [ 5.936  2.77   4.26   1.326]

Mean Vector of class 2: [ 6.588  2.974  5.552  2.026]
```

```
In [11]: # Computing the covariance matrices

         # within-class covariance matrices
         sigma_w = np.zeros((4,4))
         for cl,mv in zip(range(0,3), mean_vectors):
             class_cov_mat = np.zeros((4,4))                  # scatter matrix for every class
             for row in iris[target == cl]:
                 row, mv = row.reshape(4,1), mv.reshape(4,1)  # make column vectors
                 class_cov_mat += (row-mv).dot((row-mv).T)
             sigma_w += class_cov_mat                         # sum class scatter matrices
         print('within-class covariance Matrix:\n', sigma_w)
```

```
within-class covariance Matrix:
 [[ 38.9562  13.683   24.614    5.6556]
 [ 13.683   17.035    8.12     4.9132]
 [ 24.614    8.12    27.22     6.2536]
 [  5.6556   4.9132   6.2536   6.1756]]
```

```
In [41]: # between-class covariance matrices
         overall_mean = np.mean(iris, axis=0)
         print('overall mean:\n',overall_mean)

         sigma_b = np.zeros((4,4))
         for mv in mean_vectors:
             mv = mv.reshape(4,1) # make column vector
             overall_mean = overall_mean.reshape(4,1) # make column vector
             sigma_b +=  1/3 * (mv - overall_mean).dot((mv - overall_mean).T)

         print('between-class Scatter Matrix:\n', sigma_b)
```

```
overall mean:
 [ 5.84333333  3.054       3.75866667  1.19866667]
between-class Scatter Matrix:
 [[ 0.42141422 -0.13022667  1.10109778  0.47575378]
 [-0.13022667  0.073184   -0.37370133 -0.14994933]
 [ 1.10109778 -0.37370133  2.91095822  1.24605422]
 [ 0.47575378 -0.14994933  1.24605422  0.53736089]]
```

1.0.1 1. Find the optimal direction

```
In [42]: # calculate eigenvectors,eigenvalues for inverse of sigma_w dot sigma_b
         eig_values, eig_vectors = np.linalg.eigh(np.linalg.inv(sigma_w).dot(sigma_b))
```

```
        for i in range(len(eig_values)):
            eigvec_sc = eig_vectors[:,i].reshape(4,1)
            print('\nEigenvector {}: \n{}'.format(i+1, eigvec_sc.real))
            print('Eigenvalue {:}: {:.2e}'.format(i+1, eig_values[i].real))
```

```
Eigenvector 1:
[[ 0.43122748]
 [ 0.08952333]
 [ 0.50890581]
 [-0.73962376]]
Eigenvalue 1: -8.67e-02

Eigenvector 2:
[[ 0.77595698]
 [ 0.43033621]
 [-0.40027793]
 [ 0.22908315]]
Eigenvalue 2: -4.76e-02

Eigenvector 3:
[[-0.38420452]
 [ 0.89010784]
 [ 0.24019528]
 [ 0.04900146]]
Eigenvalue 3: 2.41e-02

Eigenvector 4:
[[ 0.25361489]
 [-0.12043398]
 [ 0.72325561]
 [ 0.63093301]]
Eigenvalue 4: 3.27e-01
```

```
In [43]: for i in range(len(eig_values)):
             eigenvector = eig_vectors[:,i].reshape(4,1)
             eigenvalue = eig_values[i]
             left = (np.linalg.inv(sigma_w).dot(sigma_b)).dot(eigenvector)
             right = eigenvalue * eigenvector
             print('\n',left-right)
```

```
 [[  1.90209617e-02]
 [ -2.68951782e-02]
 [  9.31535889e-02]
 [ -5.55111512e-17]]
```

```
[[  4.04358520e-02]
 [  2.81356808e-02]
 [ -2.88523956e-02]
 [  1.38777878e-17]]

[[  9.27616446e-03]
 [ -2.05747387e-02]
 [ -6.17159934e-03]
 [  5.16080234e-17]]

[[ -1.42006246e-01]
 [ -7.15220049e-02]
 [ -7.94642863e-02]
 [ -2.77555756e-17]]
```

```
In [44]: # Make a list of (eigenvalue, eigenvector) tuples
         eig_pairs = [(np.abs(eig_values[i]), eig_vectors[:,i]) for i in range(len(eig_values)

         # Sort the (eigenvalue, eigenvector) tuples from high to low
         eig_pairs = sorted(eig_pairs, key=lambda k: k[0], reverse=True)

         # Visually confirm that the list is correctly sorted by decreasing eigenvalues

         print('Eigenvalues in decreasing order:\n')
         for i in eig_pairs:
             print(i[0])
```

```
Eigenvalues in decreasing order:

0.327191967911
0.0867488177535
0.0475633100106
0.0241169909437
```

```
In [45]: # express the explained variance as percentage
         print('Variance explained:\n')
         eigv_sum = sum(np.abs(eig_values))
         for i,j in enumerate(eig_pairs):
             print('eigenvalue {0:}: {1:.2%}'.format(i+1, (j[0]/eigv_sum).real))
```

```
Variance explained:

eigenvalue 1: 67.38%
eigenvalue 2: 17.86%
eigenvalue 3: 9.79%
eigenvalue 4: 4.97%
```

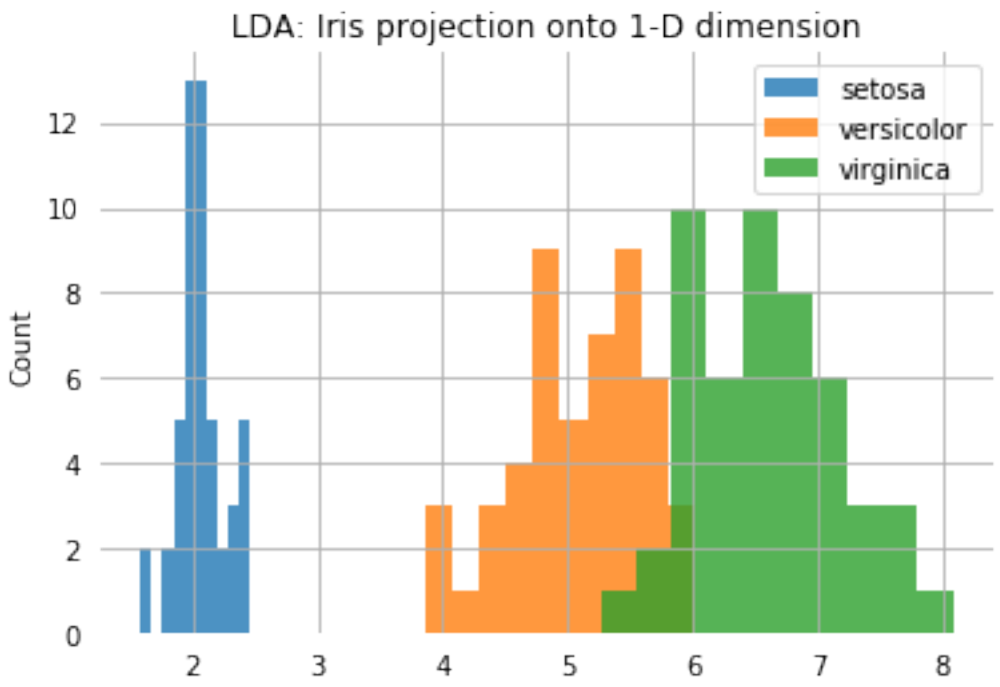
```
In [47]: W = eig_pairs[0][1].reshape(4,1)
         print('The optimal w*:\n',W)
```

```
The optimal w*:
 [[ 0.25361489]
 [-0.12043398]
 [ 0.72325561]
 [ 0.63093301]]
```

1.0.2 2. Plot the histogram

```
In [48]: iris_lda = iris.dot(W)
         # print(iris_lda)
```

```
In [49]: # plot the histogram
         def plot_1d_iris():
             ax = plt.subplot(111)
             for label,color in zip(range(0,3), ('blue','red','green')):
         #         print(iris_lda[target==cl])
                 plt.hist(x=iris_lda[target==label],label=target_names[label],alpha=0.8)

             plt.ylabel('Count')

             legend = plt.legend(loc='upper right', fancybox=True)
             plt.title('LDA: Iris projection onto 1-D dimension')
             # hide axis ticks
             plt.tick_params(axis="both", which="both", bottom="off", top="off",
                     labelbottom="on", left="off", right="off", labelleft="on")

             # remove axis spines
             ax.spines["top"].set_visible(False)
             ax.spines["right"].set_visible(False)
             ax.spines["bottom"].set_visible(False)
             ax.spines["left"].set_visible(False)

             plt.grid()
             plt.tight_layout
             plt.show()

         plot_1d_iris()
```



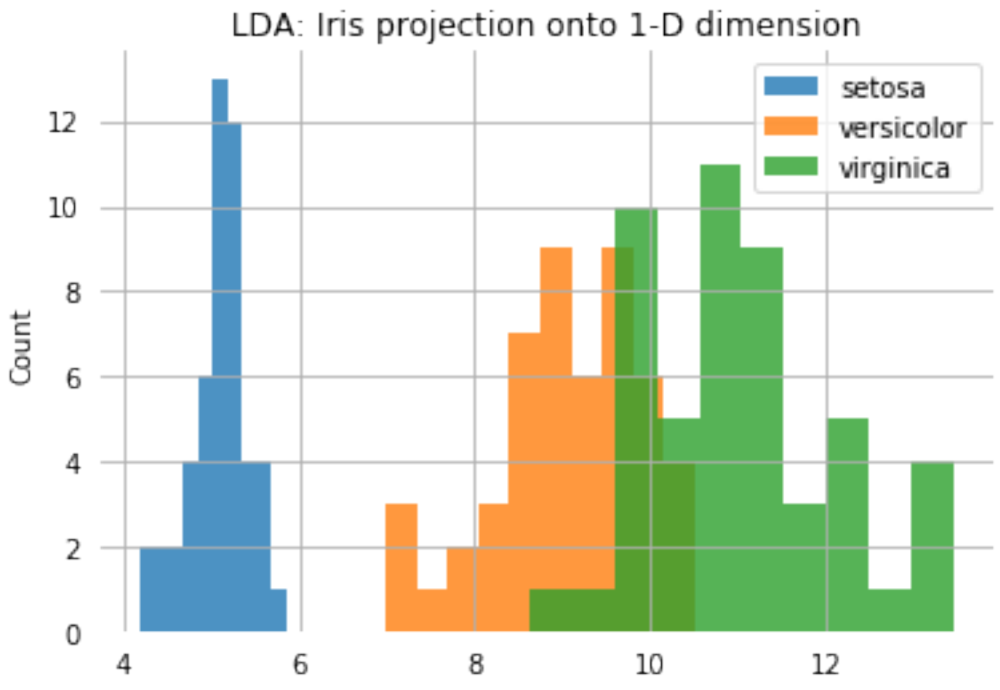
1.0.3 3. W = w* + a ?

```
In [50]: a = eig_pairs[1][1].reshape(4,1)
         W2 = W + a
         print(W2)
```

```
[[ 0.68484237]
 [-0.03091065]
 [ 1.23216142]
 [-0.10869075]]
```

```
In [51]: iris_lda2 = iris.dot(W2)
         # plot the histogram
         def plot_1d_iris2():
             ax = plt.subplot(111)
             for label,color in zip(range(0,3), ('blue','red','green')):
         #         print(iris_lda[target==cl])
                 plt.hist(x=iris_lda2[target==label],label=target_names[label],alpha=0.8)

             plt.ylabel('Count')

             legend = plt.legend(loc='upper right', fancybox=True)
             plt.title('LDA: Iris projection onto 1-D dimension')
             # hide axis ticks
```

```
    plt.tick_params(axis="both", which="both", bottom="off", top="off",
            labelbottom="on", left="off", right="off", labelleft="on")

    # remove axis spines
    ax.spines["top"].set_visible(False)
    ax.spines["right"].set_visible(False)
    ax.spines["bottom"].set_visible(False)
    ax.spines["left"].set_visible(False)

    plt.grid()
    plt.tight_layout
    plt.show()

plot_1d_iris2()
```



4_Linear Regression with non-linear functions

December 3, 2018

0.1 4.1 Performing linear regression

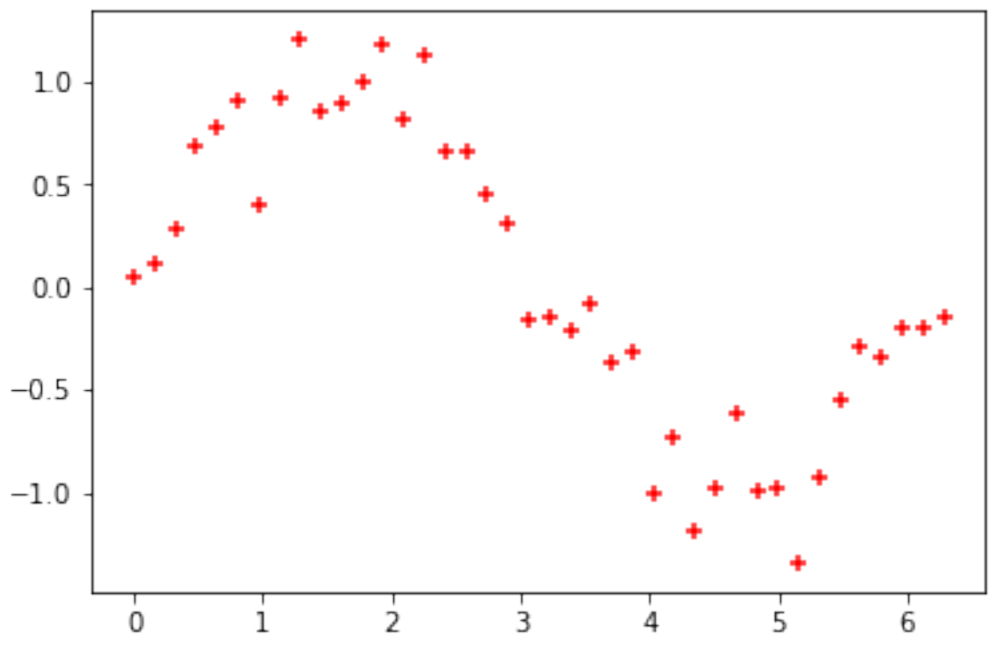
0.1.1 1.Learn the weights w

```
In [31]: %matplotlib inline
         import math
         import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt
```

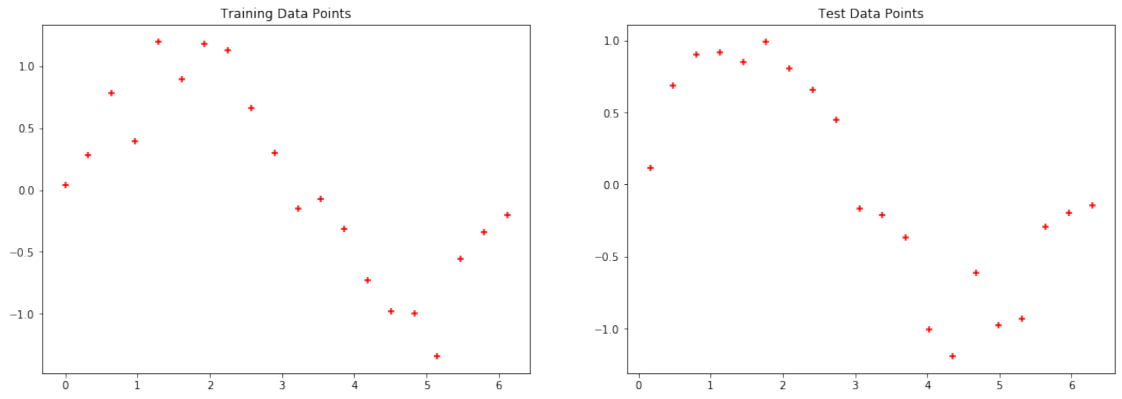
```
In [32]: #
         def generate_data_points(N):
             x = np.linspace(0, 2 * math.pi, N, dtype=float)
             y = []
             for i in range(0,len(x)):
                 fx = np.sin(x[i],dtype=float) + np.random.normal(0,scale=0.2)
                 y.append(fx)
             return x, y
```

```
In [33]: #
         x, y = generate_data_points(40)
         plt.scatter(x, y, marker='+', color='red')
```

```
Out[33]: <matplotlib.collections.PathCollection at 0x1a1786e2e8>
```



```
In [34]: # Split up the dataset into 2 parts
         x_tr = x[:40:2]
         x_ts = x[1:40:2]
         y_tr = y[:40:2]
         y_ts = y[1:40:2]
         X_tr = np.atleast_2d(x_tr).T
         X_ts = np.atleast_2d(x_ts).T
         fig, ax = plt.subplots(nrows=1,ncols=2,figsize=(18,6))
         ax[0].scatter(x_tr,y_tr,marker='+', color='red')
         ax[0].set_title('Training Data Points')
         ax[1].scatter(x_ts,y_ts,marker='+', color='red')
         ax[1].set_title('Test Data Points')

Out[34]: Text(0.5,1,'Test Data Points')
```



```
In [35]: # A
         def fn(X,t):
             return np.power(X, t, dtype=float)

         #
         def C(w,lin):
             penalty = lin * np.sum(np.square(w, dtype=float), axis=0,dtype=float)
             return penalty
```

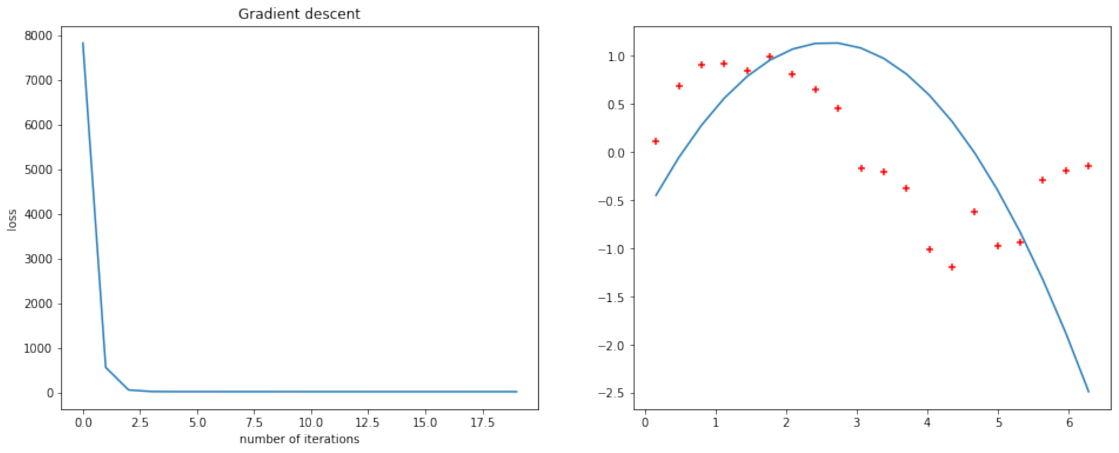
```
In [223]: #
          def loss_func(y,A,w,lin):
              N, p = A.shape
              loss = np.sum(np.square(y - A.dot(w), dtype=float), axis=0, dtype=float) + C(w,li
              return loss

          # define gradient_descent
          def gradient_descent(A, y, winit, rate, lin, numiter):
              N, p = A.shape
              whistory = []
              losshistory = []
              w = winit
              for i in range(numiter):
                  loss = loss_func(y, A, w, lin)
                  whistory.append(w)
                  losshistory.append(loss)
                  grad = (-2) * A.T.dot((y-A.dot(w))) + 2 * w.T
                  w = w - rate * grad
              return w, np.asarray(whistory), np.asarray(losshistory)
```

```
In [230]: # assume p=2
          # Design Matrix
          def design_matrix(X):
              X = X[:,0]
              col1 = np.ones(X.shape)
              col2 = X
              col3 = fn(X,2)
              return np.stack((col1, col2, col3)).T

          # Train
          A2_tr = design_matrix(X_tr)
          N, p = A2_tr.shape
          winit = np.random.randn(p)
          rate = 0.0001
          lin = 0.5
          numiter = 20
```

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(16,6))
ax[0].plot(gradient_descent(A2_tr, y_tr, winit, rate, lin, numiter)[2])
optimal_w = gradient_descent(A2_tr, y_tr, winit, rate, lin, numiter)[0]
print('optimal w = ',optimal_w)
ax[0].set_title("Gradient descent")
ax[0].set_xlabel("number of iterations")
ax[0].set_ylabel("loss")

A2_ts = design_matrix(X_ts)
y_2 = A2_ts.dot(optimal_w)
ax[1].scatter(x_ts, y_ts, marker='+', color='red')
ax[1].plot(x_ts, y_2)
```

```
optimal w =  [-0.6648862   1.38910887 -0.26728129]
```

```
Out[230]: [<matplotlib.lines.Line2D at 0x1a3012f1d0>]
```

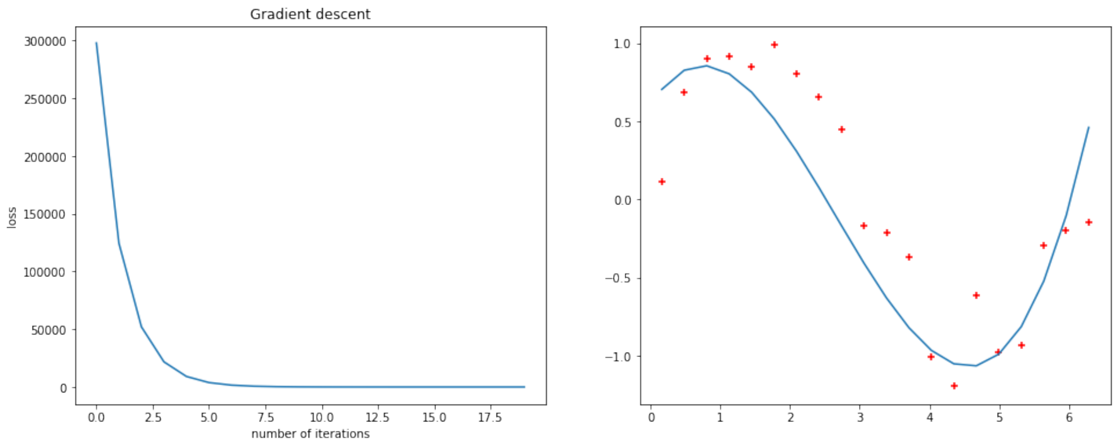


```
In [463]: # assume p=3

          # Design Matrix
          def design_matrix(X):
              X = X[:,0]
              col1 = np.ones(X.shape)
              col2 = X
              col3 = fn(X,2)
              col4 = fn(X,3)
              return np.stack((col1, col2, col3,col4)).T

          # Train
          A3_tr = design_matrix(X_tr)
          N, p = A3_tr.shape
```

```
            winit = np.random.randn(p)
            rate = 0.000001
            lin = 0.5
            numiter = 20
            fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(16,6))
            ax[0].plot(gradient_descent(A3_tr, y_tr, winit, rate, lin, numiter)[2])
            optimal_w = gradient_descent(A3_tr, y_tr, winit, rate, lin, numiter)[0]
            print('optimal w = ',optimal_w)
            ax[0].set_title("Gradient descent")
            ax[0].set_xlabel("number of iterations")
            ax[0].set_ylabel("loss")

            # Test the learning result
            A3_ts = design_matrix(X_ts)
            y_3 = A3_ts.dot(optimal_w)
            ax[1].scatter(x_ts, y_ts, marker='+', color='red')
            ax[1].plot(x_ts, y_3)
```

```
optimal w =  [ 0.60406742  0.71468463 -0.55432166  0.06954432]
```

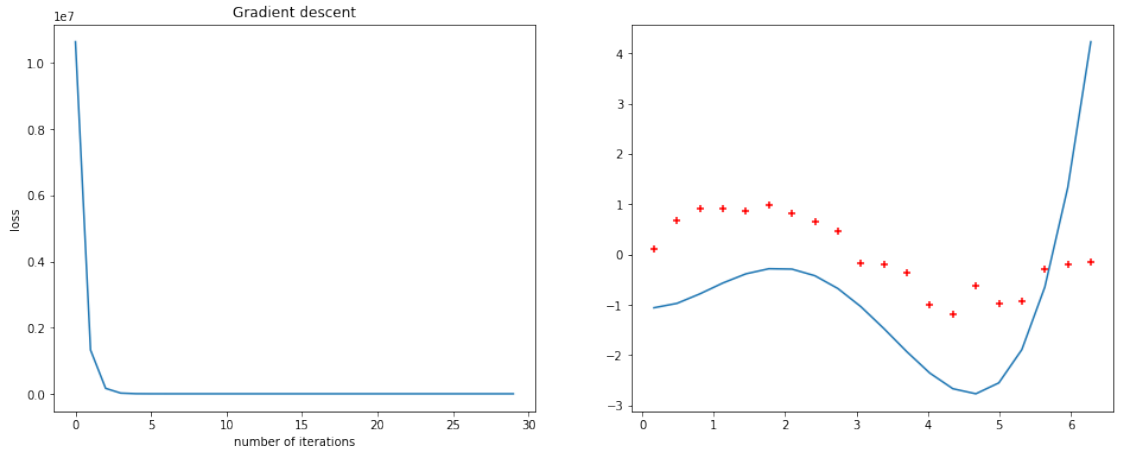
```
Out[463]: [<matplotlib.lines.Line2D at 0x1a520e0240>]
```



```
In [613]: # assume p=4

            # Design Matrix
            def design_matrix(X):
                X = X[:,0]
                col1 = np.ones(X.shape)
                col2 = X
                col3 = fn(X,2)
                col4 = fn(X,3)
                col5 = fn(X,4)
                return np.stack((col1, col2, col3,col4,col5)).T

            # Train
            A4_tr = design_matrix(X_tr)
            N, p = A4_tr.shape
            winit = np.random.randn(p)
            rate = 0.00000006
            lin = 0.5
            numiter = 30
            fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(16,6))
            ax[0].plot(gradient_descent(A4_tr, y_tr, winit, rate, lin, numiter)[2])
            optimal_w = gradient_descent(A4_tr, y_tr, winit, rate, lin, numiter)[0]
            print('optimal w = ',optimal_w)
            ax[0].set_title("Gradient descent")
            ax[0].set_xlabel("number of iterations")
            ax[0].set_ylabel("loss")

            # Test
            A4_ts = design_matrix(X_ts)
            y_4 = A4_ts.dot(optimal_w)
            ax[1].scatter(x_ts, y_ts, marker='+', color='red')
            ax[1].plot(x_ts, y_4)
```

```
optimal w =  [-1.03783969 -0.31815213  1.17784157 -0.53394669  0.05980694]
```

```
Out[613]: [<matplotlib.lines.Line2D at 0x1a544a1860>]
```

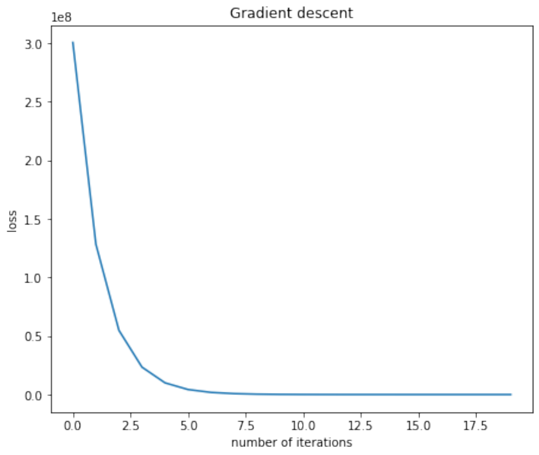


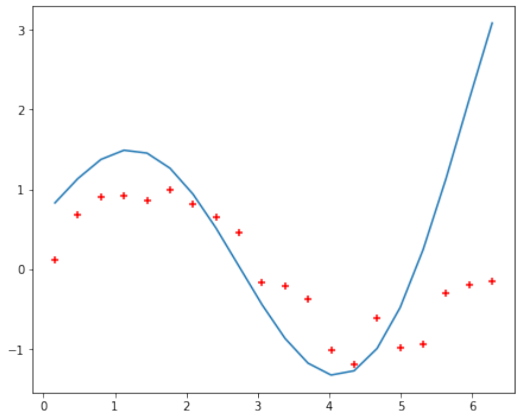
```
In [385]: # assume p=5

          # Design Matrix
```

```
def design_matrix(X):
    X = X[:,0]
    col1 = np.ones(X.shape)
    col2 = X
    col3 = fn(X,2)
    col4 = fn(X,3)
    col5 = fn(X,4)
    col6 = fn(X,5)
    cols = [col1,col2,col3,col4,col5,col6]
    return np.stack(cols).T

# Trian
A5_tr = design_matrix(X_tr)
N, p = A5_tr.shape
winit = np.random.randn(p)
rate = 0.000000001
lin = 0.5
numiter = 20
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(16,6))
ax[0].plot(gradient_descent(A5_tr, y_tr, winit, rate, lin, numiter)[2])
optimal_w = gradient_descent(A5_tr, y_tr, winit, rate, lin, numiter)[0]
print('optimal w = ',optimal_w)
ax[0].set_title("Gradient descent")
ax[0].set_xlabel("number of iterations")
ax[0].set_ylabel("loss")

# Test
A5_ts = design_matrix(X_ts)
y_5 = A5_ts.dot(optimal_w)
ax[1].scatter(x_ts, y_ts, marker='+', color='red')
ax[1].plot(x_ts, y_5)
```

```
optimal w =  [ 0.67936116  0.88756441  0.38988102 -0.64255124  0.15706285 -0.01061723]
```

```
Out[385]: [<matplotlib.lines.Line2D at 0x1a476412e8>]
```



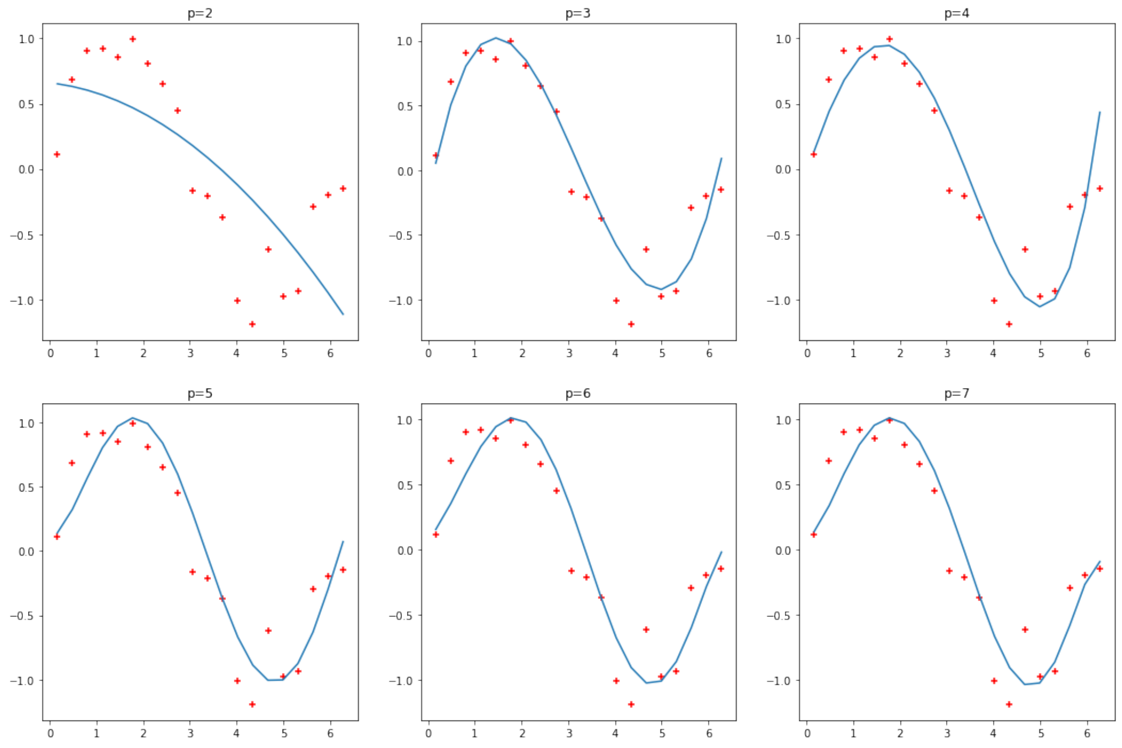


0.1.2 2. Obtain the weights from the analytical expression

```
In [450]: # obtain the weights w
          def calcul_optw(A, lin, I, y):
              w = np.linalg.inv((A.T.dot(A) + lin*I)).dot(A.T).dot(y)
              return w
          # Design matrix X
          def comb_designmat(X,p):
              X = X[:,0]
              cols = []
              col0 = np.ones(X.shape)
              cols.append(col0)
              for i in range(1,p+1):
                  cols.append(fn(X,i))
              A = np.stack(cols).T
              return A
          # Identity matrix I
          def Imat(p):
              return np.ones((p+1,p+1))
```

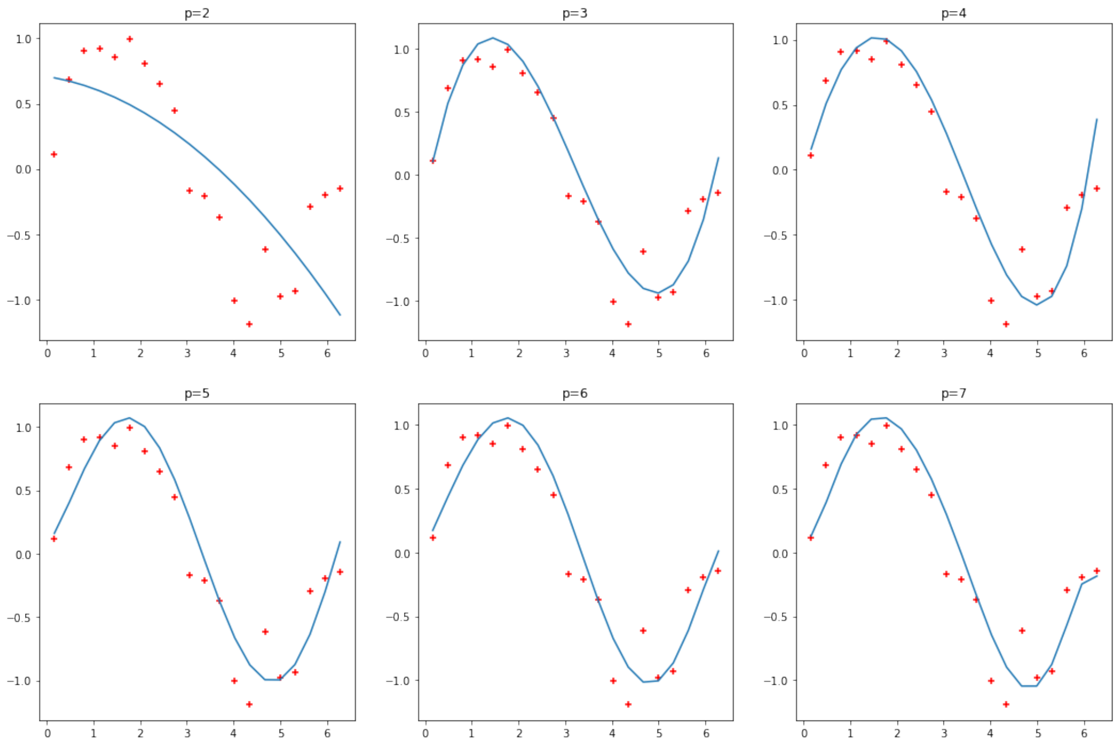
```
In [451]: # for diifferent p, lambda=0.5
          ps = [2,3,4,5,6,7]
          lin = 0.5
          optws = []
          optys = []
          for p in ps:
              I = Imat(p)
              X = comb_designmat(X_tr,p)
              optw = calcul_optw(X,lin,I,y_tr)
              A_ts = comb_designmat(X_ts,p)
              opty = A_ts.dot(optw)
              optws.append(optw)
```

```
    optys.append(opty)

fig, ax = plt.subplots(nrows=2,ncols=3,figsize=(18,12))
for i in range(2):
    for j in range(3):
            ax[i][j].scatter(x_ts, y_ts, marker='+', color='red')
            if (i == 0):
                ax[i][j].plot(x_ts,optys[j])
                ax[i][j].set_title('p={:.0f}'.format(ps[j]))
            else:
                ax[i][j].plot(x_ts,optys[j+3])
                ax[i][j].set_title('p={:.0f}'.format(ps[j+3]))
```



```
In [452]: # for diifferent p , no penalty
          ps = [2,3,4,5,6,7]
          lin = 0
          optws = []
          optys = []
          for p in ps:
              I = Imat(p)
              X = comb_designmat(X_tr,p)
              optw = calcul_optw(X,lin,I,y_tr)
              A_ts = comb_designmat(X_ts,p)
              opty = A_ts.dot(optw)
```

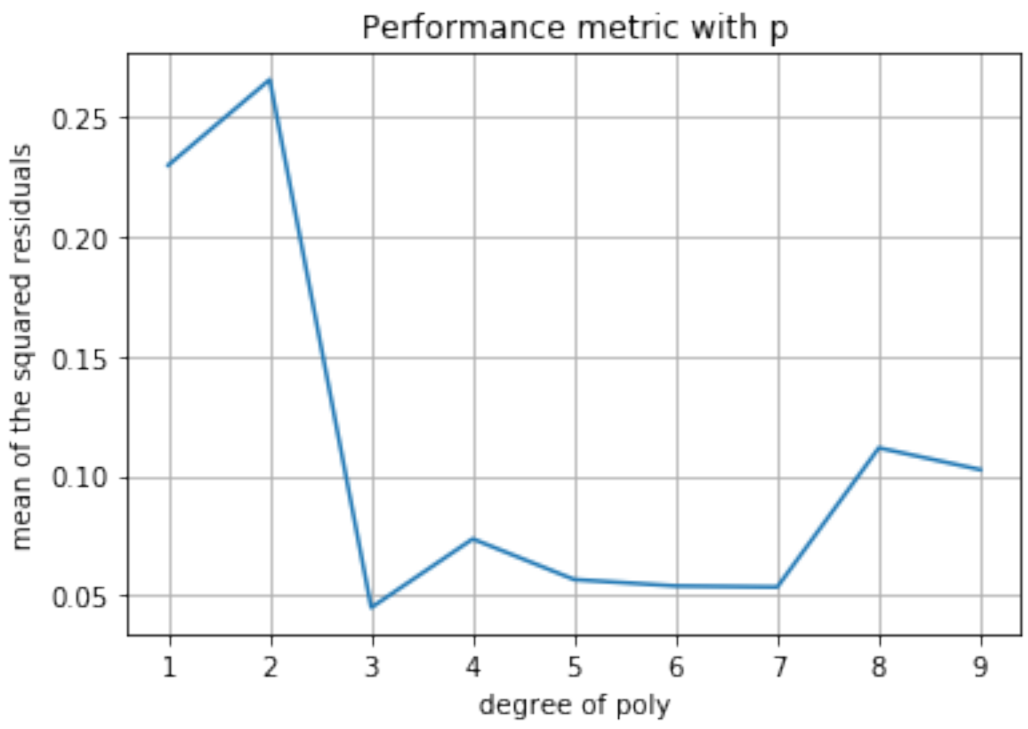
```
        optws.append(optw)
        optys.append(opty)

    fig, ax = plt.subplots(nrows=2,ncols=3,figsize=(18,12))
    for i in range(2):
        for j in range(3):
                ax[i][j].scatter(x_ts, y_ts, marker='+', color='red')
                if (i == 0):
                    ax[i][j].plot(x_ts,optys[j])
                    ax[i][j].set_title('p={:.0f}'.format(ps[j]))
                else:
                    ax[i][j].plot(x_ts,optys[j+3])
                    ax[i][j].set_title('p={:.0f}'.format(ps[j+3]))
```



0.1.3 3. Plot the measure

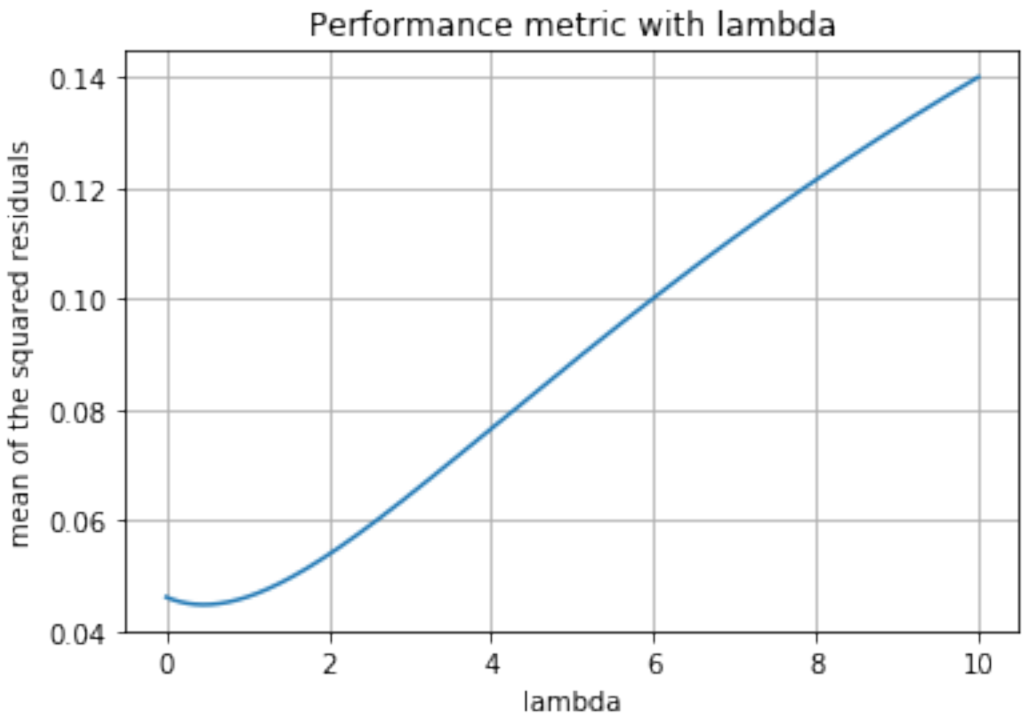
```
In [596]: def msr(y_ts,y_hat):
              N = len(y_ts)
              msr = 1/N * (np.sum(np.square(y_ts - y_hat)))
              return msr

In [598]: # different p, same lambda
          ps = range(1,10)
          lin = 0.5
```

```
optmsrs = []
for p in ps:
    I = Imat(p)
    X = comb_designmat(X_tr,p)
    optw = calcul_optw(X,lin,I,y_tr)
    A_ts = comb_designmat(X_ts,p)
    opty = A_ts.dot(optw)
    optmsr = msr(y_ts, opty)
    optmsrs.append(optmsr)

plt.plot(ps, optmsrs)
plt.xlabel('degree of poly')
plt.ylabel('mean of the squared residuals')
plt.title('Performance metric with p')
plt.grid()
```



```
In [614]: # same p, different lambda
          p = 3
          lins = np.linspace(0,1,2000)
          optmsrs = []
          for lin in lins:
              I = Imat(p)
              X = comb_designmat(X_tr,p)
              optw = calcul_optw(X,lin,I,y_tr)
```

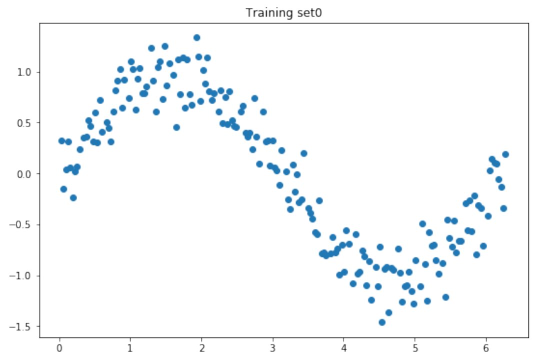
```
        A_ts = comb_designmat(X_ts,p)
        opty = A_ts.dot(optw)
        optmsr = msr(y_ts, opty)
        optmsrs.append(optmsr)

    plt.plot(lins, optmsrs)
    plt.xlabel('lambda')
    plt.ylabel('mean of the squared residuals')
    plt.title('Performance metric with lambda')
    plt.grid()
```

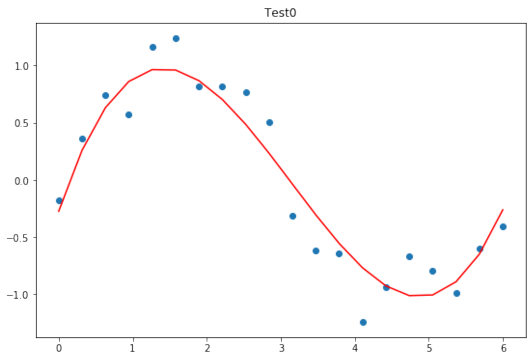


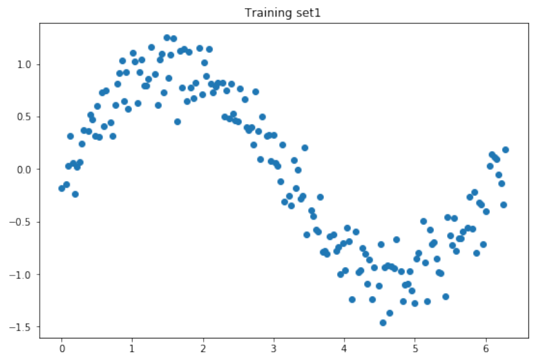
0.2 4.2 How does linear regression generalise?(10-folds cross-validation)

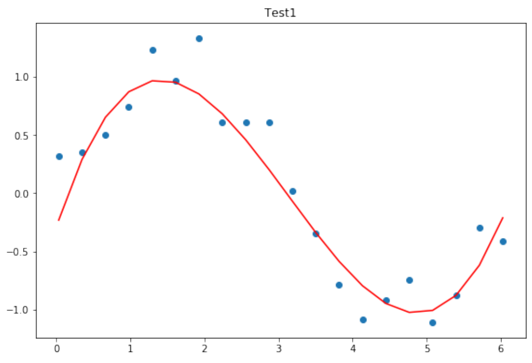
```
In [533]: # Generate the N data point
          x_new, y_new = generate_data_points(200)
          p = 3
          lin = 0.5
          I = Imat(p)
          # Split the N=200 data points
          x_folds = []
          y_folds = []
          for i in range(10):
              x_folds.append(x_new[i:200:10])
              y_folds.append(y_new[i:200:10])
```

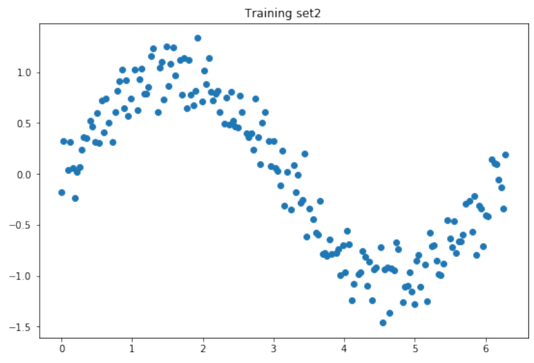
```
w_folds = []
msr_folds = []
for i in range(len(x_folds)):
    X_test = x_folds[i]
    Y_test = y_folds[i]
    X_train = []
    Y_train = []
    for j in range(len(x_folds)):
        if (j != i ):
            X_train.append(x_folds[j])
            Y_train.append(y_folds[j])

    X_trains = np.concatenate(X_train)
    Y_trains = np.concatenate(Y_train)
    X_train_col = np.atleast_2d(X_trains).T
    X_test_col = np.atleast_2d(X_test).T
#     print(X_test_col)
    # Train
    # Calculate the optimal w
    X = comb_designmat(X_train_col,p)
#     print(X,'\n')
    optw = calcul_optw(X,lin,I,Y_trains)
    w_folds.append(optw)
#     print(optw)
    # Get the fit y
    A_test = comb_designmat(X_test_col,p)
    opty = A_test.dot(optw)
    optmsr = msr(Y_test, opty)
    msr_folds.append(optmsr)
#     print(X_test,'\n',opty,'\n')
    # Plot
    fig, ax = plt.subplots(nrows=1,ncols=2,figsize=(20,6))
    ax[0].scatter(X_trains,Y_trains)
    ax[0].set_title('Training set{:.0f}'.format(i))
    ax[1].scatter(X_test,Y_test)
    ax[1].plot(X_test,opty,color='red')
    ax[1].set_title('Test{:.0f}'.format(i))

# Mean of 10 w*
w_sum = 0
msr_sum = 0
for w,msr in zip(w_folds,msr_folds):
    w_sum += w
    msr_sum += msr
w_mean = w_sum / 10
msr_mean = msr_sum / 10
print('After 10-folds cross-validation,\nThe optimal w = ', w_mean,'\nThe mean error
After 10-folds cross-validation,
The optimal w =  [-0.26474915  1.96656939 -0.90255197  0.09582414]
The mean error is:  0.0454045984953
```

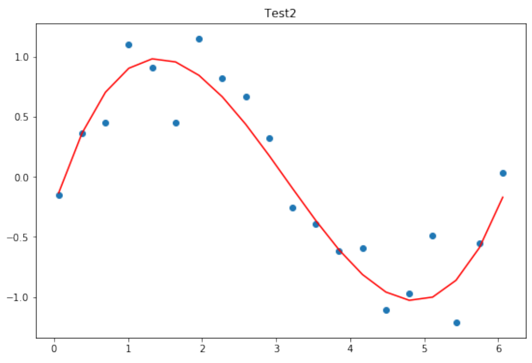


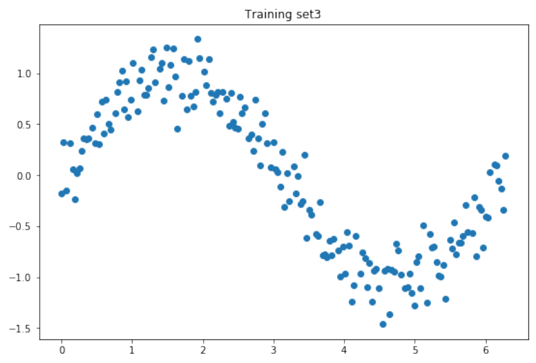


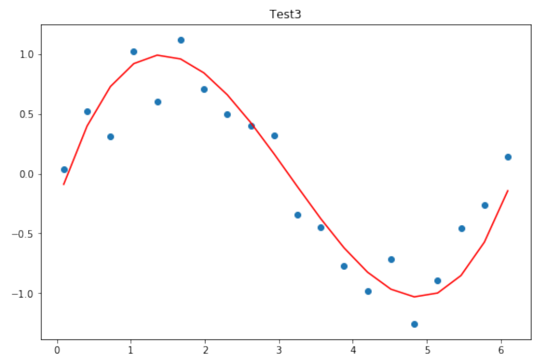


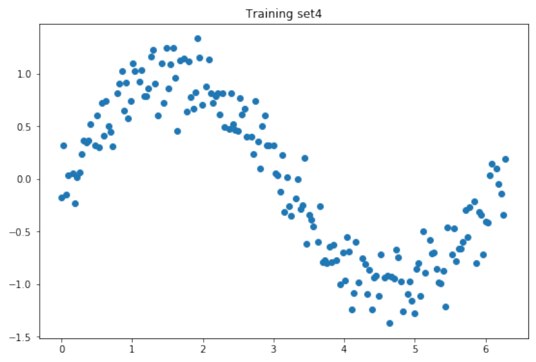


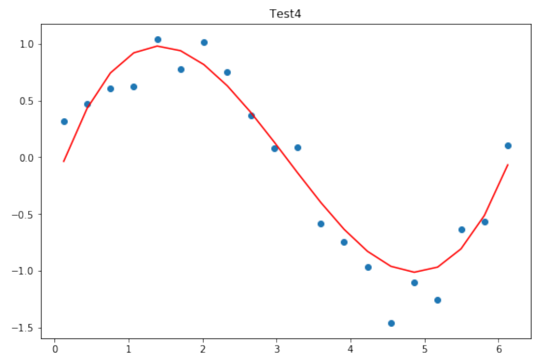


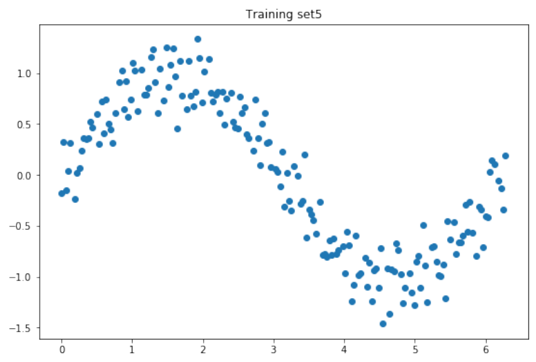


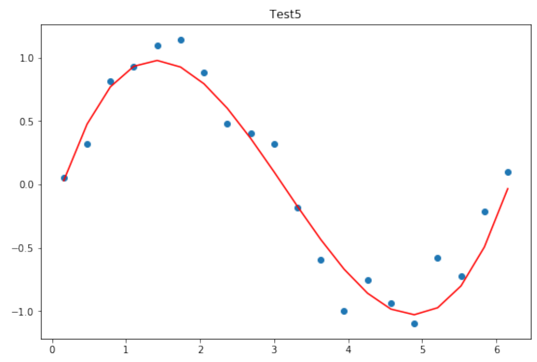

Training set3
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
1
2
3
4
5
6

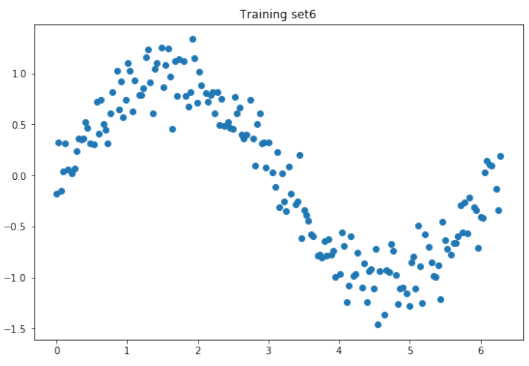

Test3
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6

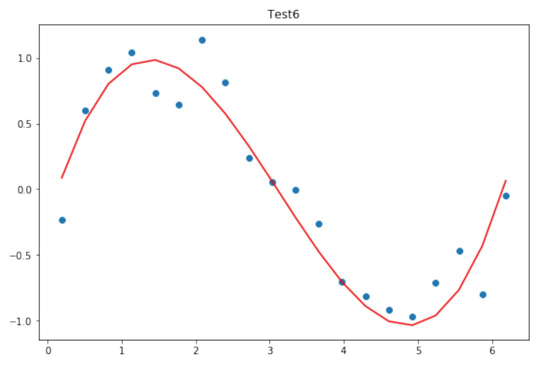

Training set4
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
1
2
3
4
5
6

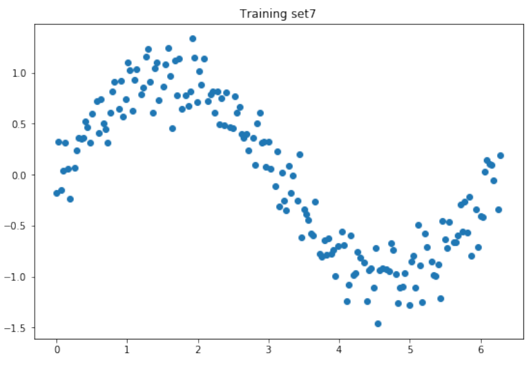

Test4
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
1
2
3
4
5
6

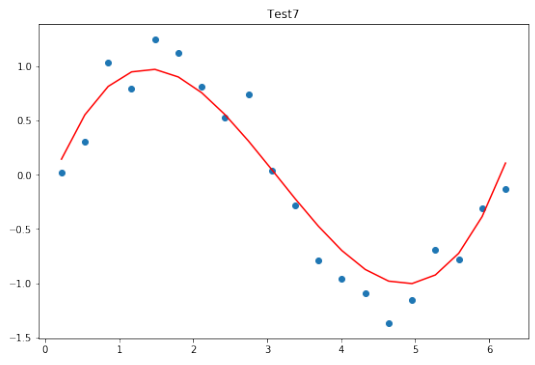

Training set5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
1
2
3
4
5
6

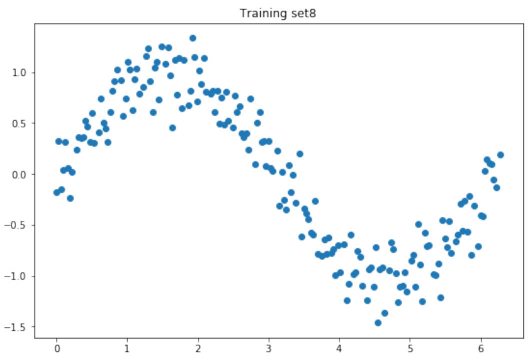

Test5
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6

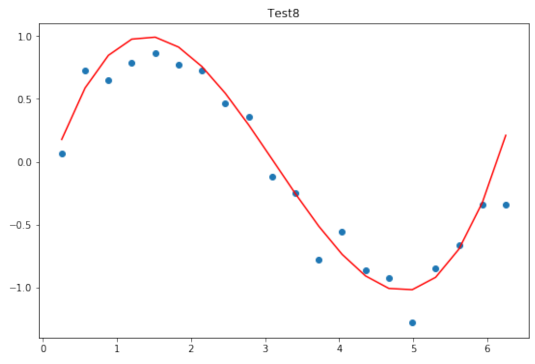

Training set6

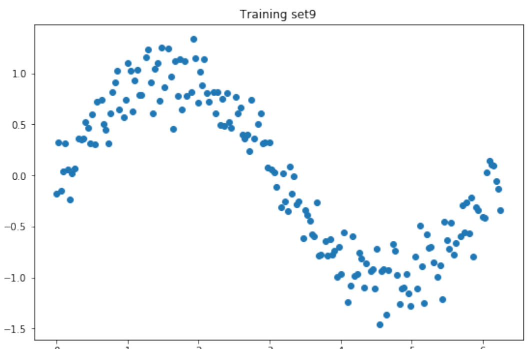

Test6

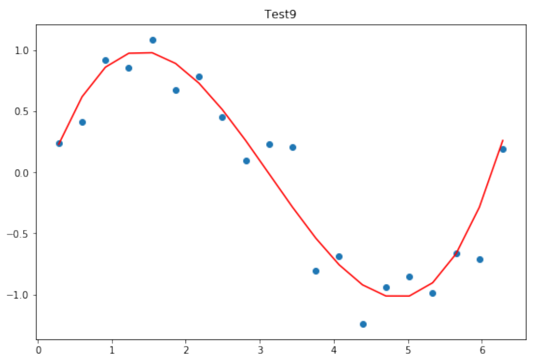

Training set7


Test7


Training set8


Test8


Training set9


Test9