Machine Learning: Assignment #1
多项式拟合正弦函数

康瑞

1160300514@stu.hit.edu.cn

HIT — September 27, 2018

Contents

1 Details/Environments

实验目的：

掌握最小二乘法求解（无惩罚项的损失函数）、掌握加惩罚项（2 范数）的损失函数优化、梯度下降法、共轭梯度法、理解过拟合、克服过拟合的方法 (如加惩罚项、增加样本

实验要求：

(1). 生成数据，加入噪声

(2). 用高阶多项式函数拟合曲线

(3). 用解析解求解两种 loss 的最优解（无正则项和有正则项）

(4). 优化方法求解最优解（梯度下降，共轭梯度)

(5). 用你得到的实验数据，解释过拟合

(6). 用不同数据量，不同超参数，不同的多项式阶数，比较实验效果

实验环境：python + numpy

2 设计思想

2.1 算法原理

2.1.1 最小二乘法求参数矩阵

$$\nabla_W Loss = \frac{\partial Loss}{\partial W} = \frac{\partial[\frac{1}{2}(XW-Y)^T(XW-Y)-\frac{\lambda}{2}W^TW]}{\partial W} = X^TXW - X^TY + \lambda W^1 \tag{1}$$

从推导中可以见到为使 Loss 为最小值，需要使 Loss 在 W 参数矩阵空间下的投影为最小值的等价形式为

$$\nabla_W Loss = 0 \tag{2}$$

因此，由等式（1）可知，

$$W = (X^TX + \lambda IW)^{-1}X^TY \tag{3}$$

根据已导出的等式可以直接求得 W。

2.1.2 梯度下降法求参数矩阵

由 (1) 知：

$$\nabla_W Loss = X^TXW - X^TY + \lambda W \tag{4}$$

[1]Here we suppose Y, W are column vector, X is a transposition of vandermonde matrix

为 Loss 在 W 参数矩阵形成的空间中最快下降方向，可以用此 Loss 对 W 进行迭代更新

$$W = W - \alpha(X^T XW - X^T Y + \lambda W) \tag{5}$$

迭代步长

$$\alpha = 1e - 7 \tag{6}$$

2.1.3　共轭梯度下降

共轭梯度法是一种求解对称正定线性方程组 Ax=b 的迭代方法, 对于求解方程组的问题可以等价的求解其对应的二次范数为 0 的解，因此可以用迭代法。对于方程 AX=b, 可以对其变换：

$$||Ax - b||_2^2 = 0 \tag{7}$$

$$minimize_{x \in R^n} \frac{1}{2} x^T A^T Ax - b^T Ax \tag{8}$$

即变成了一个二次规划问题。对于正定矩阵 Q，如果非零向量 x,y 相对于 Q 共轭，则

$$x^T Qy = 0 \tag{9}$$

因此可以找到 n 个相互共轭、线性无关的 n 维向量,$d_i(i \in 0, 1, 2...n-1, d_i \in R^n)$,$x = \sum_{i=0}^{n-1} a_i d_i, (a_i \in R)$ 因此目标函数可推得：

$$min_{d_i \in R^n, a_i \in R} \frac{1}{2} (\sum_{i=0}^{i=n-1} a_i d_i)^T Q (\sum_{i=0}^{i=n-1} a_i d_i) - b^T (\sum_{i=0}^{i=n-1} a_i d_i) \tag{10}$$

$$= min_{d_i \in R^n, a_i \in R} \frac{1}{2} (\sum_{i=0}^{i=n-1} \sum_{j=0}^{j=n-1} a_i a_j d_i^T Q d_j) - (\sum_{i=0}^{i=n-1} a_i b^T d_i) \tag{11}$$

由 $b_i Q b_j = 0$ (b_i 为关于 Q 的共轭向量) 因此，可以化简 (11) 式得：

$$min_{d_i \in R^n, a_i \in R} (\sum_{i=0}^{i=n-1} \frac{1}{2} a_i^2 d_i^T Q d_i - a_i b^T d_i) \tag{12}$$

因此，可以求每一项的最小值，线性组合后的向量为 x 的解。可以对每一项关于 a_i 求导，得到 a_i 与 d_i 的关系，得：

$$a_i d_i^T Q d_i - b^T d_i = 0 \tag{13}$$

$$a_i = \frac{b^T d_i}{d_i^T Q d_i} \tag{14}$$

$$x = \sum_{i=0}^{i=n-1} \frac{b^T d_i}{d_i^T Q d_i} d_i \tag{15}$$

利用施密特正交化方法: 线性无关向量组未必是正交向量组, 但正交向量组又是重要的, 因此: 从一个线性无关向量组 $\alpha_1, \alpha_2 ... \alpha_m$ 出发, 构造出一个标准正交向量组 $e_1, e_2 ... e_m$, 并且使向量组 $\alpha_i, i \in 1, ...m$ 与向量组 $e_i, i \in 1, ...m$ 等价。可以用于本题求解，首先取两个初始线性无关的向量，之后迭代生成解。

2.2 算法实现

2.2.1 Gradient Descent

Algorithm 1: `GD norm`

Input: (X, Y), X is a trans of vandermonde matrix of train data x; Y is a column vector, y_train

Result: W, such that loss is (almostly) minimized

$cnt \leftarrow 1$

while $loss \geq 0.01$ *or* $cnt \leq 10^5$ **do**

  $grad \leftarrow X^T(XW - Y) + \lambda W$

  $W \leftarrow W - \alpha grad$

end

return W

2.2.2 Conjugate GD

Algorithm 2: `ConjugateGD`

Input: $jumpout, X, Y$ (X, Y are illustrated before)

Result: W

$initialize : x \leftarrow xavier\_initializer()$

$Q = X^TX + \lambda I$

$b = X^TY$

$r = b - Qx$

$d = r$

$k \leftarrow 0$

while $k < 10^6$ **do**

  $\alpha \leftarrow \frac{r^Tr}{d^TQd}$

  $x \leftarrow x + \alpha d$

  $r_{next} = r - \alpha Qd$

  if $\frac{r_{next}^T r_{next}}{size(r_{next})} \leq jumpout$ **then**

    break

  end

  $\beta \leftarrow \frac{r_{next}^T r_{next}}{r^Tr}$

  $d \leftarrow r_{next} + \beta d$

  $r \leftarrow r_{next}$

  $k \leftarrow k + 1$

end

return W

3 实验结果与分析

The dataset size [2], using dimision, loss, λ and learning rate are evaluated and illustrated in the result.

以下仅添加部分导出的结果。

3.1 Conjugate Gradient Descent

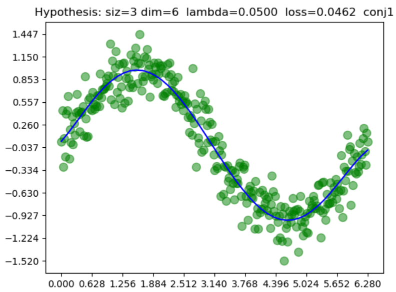

Figure 1: conjugate1

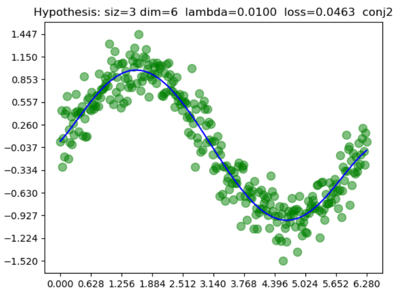

Figure 2: conjugate2

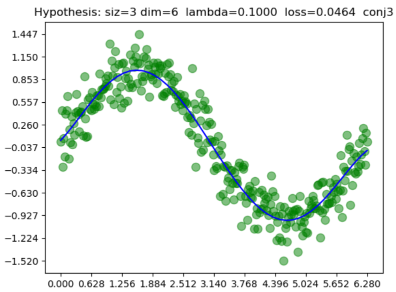

Figure 3: conjugate3

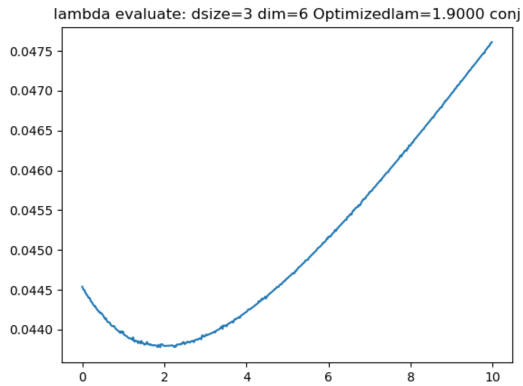

Figure 4: evaluate lambda

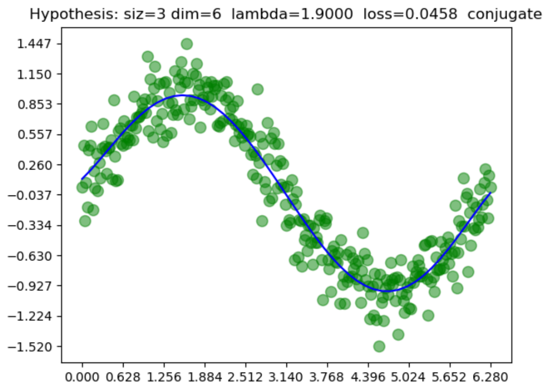

Figure 5: conjugate with optimized lambda

经过多次实验得 λ 的值与数据集的状况密切相关，λ 的值经常性的改变，但均值都在 0.5 附近，因此，在今后的实验中，数据量在 300 附近的数据集下通常使用均值为 0.5，方差为 0.3 的正态分布。由实验结果 (大量实验取频率较大的状况) 对 λ 进行观测取值，通常可以得到一个最低点，并且此 λ 适用于此次生成的数据集，使用此 λ 进行训练时通常可以得到 loss 较低的解。

3.2 Direct solving: least square method

结果可见图 6，7，8 为最小二乘法直接解出参数矩阵，图 9 为数据集规模较小且参数矩阵的维度较大时出现的过拟合情况，模型训练结果尽量保证所有的训练集数据在曲线上，但没法很好的预测其他点的情况，导致 test 过程有较高的 loss，在这种情况下，通常可以选择将 λ 调制一个较高的值，并且降低模型拟合维度。图 10 为不加正则项的求解。

[2]size is the evaluate of dataset, calculated by len(X_samp)/100

3.3 Direct Gradient Descent

结果可见图 11，由于直接梯度下降收敛较慢，所以训练到一个较适合的解便停止迭代。可以使用 tensorflow 等框架的 Adam Optimizer 进行训练，结果较好，收敛较快。

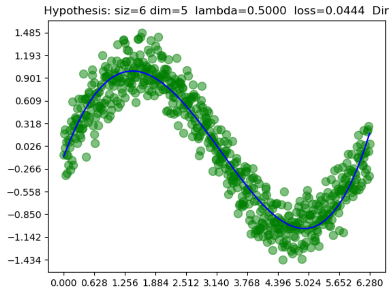

Figure 6: evaluate lambda

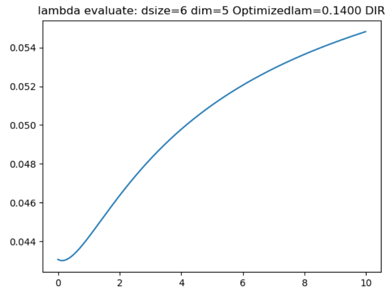

Figure 7: Direct with optimized lambda

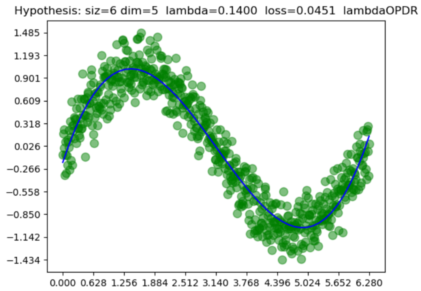

Figure 8: Direct with optimized lambda

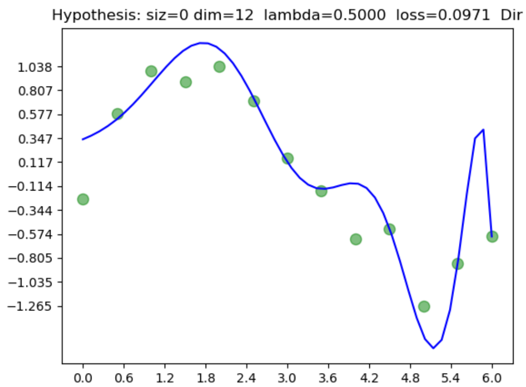

Figure 9: overfit

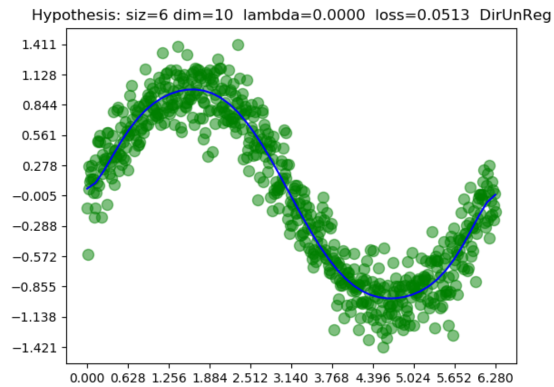

Figure 10: Direct Solve：No Regularization

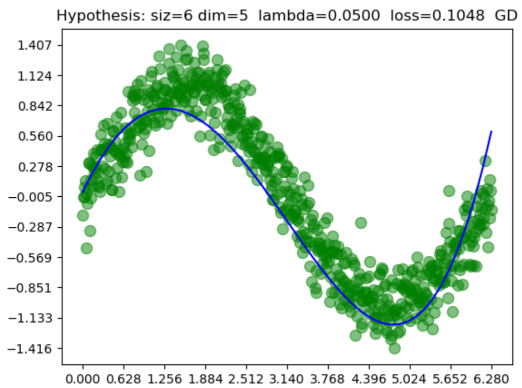

Figure 11: Gradient Descent

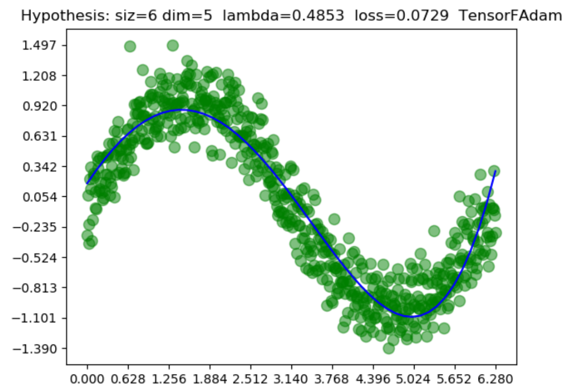

Figure 12: Gradient Descent:Tensorflow

3.4 Training with tensorflow:Gradient Descent

收敛较快，结果可见图 12，使用 Adam Optimizer 进行迭代。

4 结论

(1). 直接的梯度下降效果较差，学习率只可以维护在较低的值，随机梯度下降与 Batch 的梯度下降军没有得到很好的结果，如果选择梯度下降求解参数可以参考使用牛顿法与共轭梯度法，或实现自适应的梯度下降，例如当训练神经网络等没有解析解的情况，应用 Tensorflow 的 Optimizer 可以很好的拟合特征函数，收敛

(2). 正则项的使用对于较小数据规模的作用明显，样本变多时，λ 的取值向零点靠近，λ 的作用可以防止模型对较小的数据规模进行过度学习，当使用较大样本时，可以适当降低 λ 值。且对于多数情况，λ 值对应的 loss 随 λ 的增加而先减小后增大，此过程为模型的过拟合到最佳到欠拟合的过程。

(3). 模型拟合能力随拟合多项式的阶数升高而提升，但应对对应的数据集较好的确定阶数大小，否则会产生过拟合与欠拟合的情况。

(4). 学习率等超参数对于模型拟合的过程较为重要，可以使用 validation 集对超参数进行先行试验，已确定最佳的超参数范围。

5 参考文献

References

[1] Christopher M.Bishop. *Pattern Recognition and Machine Learning*. Springer, 2007.

[2] 周志华. 机器学习. 清华大学出版社, 2016.

[2] [1]

6 Codes

hyperparams.py

```

import numpy as np
import matplotlib.pyplot as plt


class hyperparams(object):
    def __init__(self, dim, X_samp, Y_samp, alpha, lambdas=None):
        """
        Params, Datas

        :param dim: generate dimsion to fitting: f(x) = \sum wi*x^i, i \in {0,1.. dim}
        :param X_samp:
        :param Y_samp:
        :param alpha:learning rate,
        :param lambdas:regularization hyperparam if is none, the program will either
                        choose one from valid range or just test/ignore regular
        """
        self.lambdas = lambdas
        self.dim = dim
        self.X_samp = X_samp
        self.Y_samp = Y_samp
        (self.X_train, self.Y_train), (self.X_valid, self.Y_valid), (self.X_test, self.Y_test) =
             self.data_split()
        self.X = self.matrix_X_gener()
        self.alpha = alpha
```

```python
        self.cnt = 0
        self.Y_train = np.transpose(self.Y_train)

    def data_split(self):
        """
        Split the whole dataset into 3 parts: train, valid, test = 9:4:1
        :return:
        """
        return (self.X_samp[:np.int32(9*len(self.X_samp)/14)], self.Y_samp[:np.int32(9*len(self.X_samp)/14)]), (self.X_samp[np.int32(9*len(self.X_samp)/14): np.int32(13*len(self.X_samp)/14)], self.Y_samp[np.int32(9*len(self.X_samp)/14): np.int32(13*len(self.X_samp)/14)]), (self.X_samp[np.int32(13*len(self.X_samp)/14):], self.Y_samp[np.int32(13*len(self.X_samp)/14):])

    def matrix_X_gener(self, ori=None):
        """
        Generate the vandmonde matrix. from X_train or other origin
        :param ori: X
        :return: vand-mat
        """
        if ori is None:
            ori = self.X_train
        tmp = [1 for i in range(0, len(ori))]
        ans = [tmp]
        for i in range(1, self.dim):
            tmp = [i*j for (i, j) in zip(tmp, ori)]
            ans.append(tmp)
        X = np.matrix(ans).T
        return X

    def solve_W_direct(self, regular=True):
        XT = np.transpose(self.X)
        XTX = np.dot(XT, self.X)
        XTY = np.dot(XT, np.matrix(self.Y_train).T)
        if not regular or self.lambdas is None:
            W = np.dot(np.matrix(XTX).I, np.matrix(XTY))
            return W
        else:
            W = np.dot(np.matrix(XTX+self.lambdas*np.eye(self.dim)).I, np.matrix(XTY))
            return W

    def pr_plot(self, w, X_smp=None, Y_smp=None, attached_flags=""):
        """
        Making graphics, w is params and X,Y_smp is always the whole dataset: used to plot the points.
        :param w:
        :param X_smp:
        :param Y_smp:
        :param attached_flags: labels to classify the method used to generate the W.
        :return:
        """
        self.cnt += 1
        if X_smp == None or Y_smp == None:
            X_smp = self.X_samp
            Y_smp = self.Y_samp
        plt.scatter(X_smp, Y_smp, s=75, c='green', alpha=.5)
        X_map = np.linspace(np.min(self.X_samp), np.max(self.X_samp))
        XY_exm = self.hypothesis(w, X_map)
        plt.plot(X_map, XY_exm, color='blue')
        plt.xticks(np.linspace(np.min(X_smp), np.max(X_smp), 11, endpoint=True))
        plt.yticks(np.linspace(np.min(Y_smp), np.max(Y_smp), 11, endpoint=True))
        plt.title("Hypothesis: siz=%(size)d dim=%(dim)d  lambda=%(lam).4f  loss=%(los).4f  %(fla)s" %
                  {'size': len(self.X_samp)/100, 'dim': self.dim, 'lam': self.lambdas, 'los': self.evaluate(w), 'fla': attached_flags})
        plt.savefig("Figure_Hyp dim%(dim)d lam%(lam).4f %(fla)s .png" % {'dim': self.dim, 'lam': self.lambdas, 'fla': attached_flags})
        plt.show()

    def evaluate(self, W, testX=None, testY=None):
        """
        Generate test for W, maybe use the dataset from valid or test,
        if testX,Y are not attached with one value, then use the test data,
        else use the valid(or attached): used in evaluating the lambda.
        :param W:
        :param testX:
        :param testY:
        :return:
        """
```

```
        if testX is None or testY is None:
            testX = self.X_test
            testY = self.Y_test
        Y_exm = self.hypothesis(W, testX)
        loss = np.sum([(y-y_)**2 for (y, y_) in zip(Y_exm, testY)])
        return loss/len(testX)

    def evaluate_lambda(self, max_lambda, user_method=solve_W_direct, flag=""):
        """
        Test lambda to find a better fitting,
        :param max_lambda: max range of lambda
        :param user_method: choice of method to hypothesis
        :param flag: used for graph making
        :return:
        """
        Loss = []
        Lambda = []
        evalu = 0x3fffffff
        store_lambda = 0
        for fl in range(0, int(max_lambda*100), 2):
            fl *= 0.01
            self.lambdas = fl
            tmp = self.evaluate(user_method(self), testX=self.X_valid, testY=self.Y_valid)
            Lambda.append(fl)
            Loss.append(tmp)
            if tmp == min(tmp, evalu):
                store_lambda = fl
                evalu = min(tmp, evalu)
        self.lambdas = store_lambda
        plt.plot(Lambda, Loss)
        plt.title("lambda evaluate: dsize=%(size)d dim=%(dim)d Optimizedlam=%(Opt).4f %(flags)s"
             % {'size': len(self.X_samp)/100, 'dim': self.dim, 'Opt': self.lambdas, 'flags':
            flag})
        plt.savefig("lambda evaluate dsize%(size)d dim%(dim)d Opt%(Opt).4f %(flags)s .png" % {'
            size': len(self.X_samp)/100, 'dim': self.dim, 'Opt': self.lambdas, 'flags': flag})
        plt.show()
        return store_lambda, evalu

    def hypothesis(self, W, x):
        XX = self.matrix_X_gener(ori=x)
        return np.dot(XX, W)

    def xavier_initializer(self):
        val = np.sqrt(3/len(self.X_train))
        return np.matrix([np.random.uniform(low=-val, high=val) for i in range(self.dim)]).T

    def solve_W_GD(self, regular=True, W=None):
        """
        Gradient Descent: loss = (0.5*(XW-Y)T(XW-Y) + lambda*0.5*WTW)/m
        \frac{\partial loss}{\partial W} = gradW = (XTXW - XTY + lambda*I*W)/m
        self.alpha: learning rate
        Weight initializer = xavier_initializer
        :param regular:
        :return:
        """
        if W is None:
            W = np.zeros([self.dim, 1])

        if regular is True and self.lambdas is None:
            self.lambdas = np.random.normal(loc=0.5, scale=0.4)
            print("select lambda as:" + str(self.lambdas))

        cntt = 0
        while True:
            bias = (np.dot(self.X, W) - np.matrix(self.Y_train).T)/len(self.X_train) # size*1
            grad = np.dot(self.X.T, bias) + self.lambdas * W
            W = W - self.alpha * grad
            cntt += 1
            loss = self.evaluate(W)
            if cntt == 1000 or loss <= 1e-4:
                break
            if cntt % 1000 == 0:
                print(str(cntt/100) + "data:")
                print(loss)
                print(W)
        return W

    def solve_W_tf(self, regular=True):
        """
```

```
        Tensorflow: 1.2.0 py3.5
        GD using Adam Optim from TF
        :param regular:
        :return:
        """
        if not regular:
            return
        import tensorflow as tf
        if self.lambdas is None:
            self.lambdas = np.random.normal(0.5, 0.3)
        x = tf.placeholder(tf.float32, shape=(len(self.X_train), self.dim))
        y = tf.placeholder(tf.float32, shape=(len(self.Y_train), 1))
        W = tf.Variable(tf.zeros([self.dim, 1]))
        pred = tf.matmul(x, W)
        cost = tf.reduce_mean(tf.reduce_sum(tf.matmul(tf.transpose(pred-y), pred-y))+tf.multiply
            (self.lambdas, tf.matmul(tf.transpose(W), W)))
        train = tf.train.AdamOptimizer().minimize(cost)
        sess = tf.Session()
        sess.run(tf.global_variables_initializer())
        for i in range(10000):
            sess.run([train, W], feed_dict={x: self.X, y: np.matrix(self.Y_train).T})
            if i % 100 == 0:
                print(sess.run(W))
            end = sess.run(W)
        plt.scatter(self.X_samp, self.Y_samp, alpha=.5, color='green', s=75)
        self.pr_plot(end, attached_flags="tf")
        return end

    def solve_W_conjuct(self, regular=True, jumpout=0.01):
        """
        Conjugate GD

        :param regular:
        :param jumpout: lower bound of gener_los
        :return:
        """
        if not regular:
            return
        Q = np.matrix(np.dot(self.X.T, self.X) + self.lambdas*np.eye(self.dim))
        x = self.xavier_initializer()
        b = np.dot(self.X.T, np.matrix(self.Y_train).T)
        r = b - np.dot(Q, x)
        d = r
        k = 0
        while k < 5000000:
            alpha = np.sum(np.dot(r.T, r)/np.dot(np.dot(d.T, Q), d))
            x += alpha*d
            r_tmp = r - alpha*np.dot(Q, d)
            if np.sum(np.dot(r_tmp.T, r_tmp)/np.shape(r_tmp)) <= jumpout:
                break
            beta = np.sum(np.dot(r_tmp.T, r_tmp)/np.dot(r.T, r))
            d = r_tmp + beta*d
            r = r_tmp
            k += 1
            if k % 100 == 0:
                print(self.evaluate(x))
                print(x)
        return x


def data_generator(bias=0.01, random='guss', min=0.0, max=2*np.pi):
    X = np.arange(min, max, bias)
    np.random.shuffle(X)
    X = list(X)
    Y = np.sin(X)
    for i in range(len(X)):
        if random == 'uniform':
            Y[i] += np.random.uniform(low=0.0, high=1.0)
        else:
            Y[i] += np.random.normal(loc=0.0, scale=0.2)
    return X, Y


if __name__ == '__main__':
    X_tmp, Y_tmp = data_generator(bias=0.02)
    hy = hyperparams(int(6), X_tmp, Y_tmp, 0.02, 0.05)
    w = hy.solve_W_conjuct()
    hy.pr_plot(w, attached_flags="conj1")
    hy = hyperparams(int(6), X_tmp, Y_tmp, 0.02, 0.01)
```

```
    w = hy.solve_W_conjuct()
    hy.pr_plot(w, attached_flags="conj2")
    hy = hyperparams(int(6), X_tmp, Y_tmp, 0.02, 0.1)
    w = hy.solve_W_conjuct()
    hy.pr_plot(w, attached_flags="conj3")
    hy.evaluate_lambda(10.0, user_method=hy.solve_W_conjuct, flag="conj")
    w = hy.solve_W_conjuct()
    hy.pr_plot(w, attached_flags="conjugate")
    print(hy.evaluate(w))

    X_tmp, Y_tmp = data_generator(bias=0.01)
    hy = hyperparams(int(10), X_tmp, Y_tmp, 0.02, 0.0)
    W = hy.solve_W_direct()
    hy.pr_plot(W, attached_flags="DirUnReg")

    print(hy.evaluate_lambda(10, flag="DIR"))
    W = hy.solve_W_direct()
    hy.pr_plot(W, attached_flags="lambdaOPDR")

    X_tmp, Y_tmp = data_generator()
    hy = hyperparams(5, X_tmp, Y_tmp, 0.0000002, 0.05)
    W = hy.solve_W_GD(W=np.matrix([0.03785995, 1.26832123, -0.55128356,0.01908278,0.01176263]).T
        )
    hy.pr_plot(W, attached_flags="GD")
    print(W)

    X_tmp, Y_tmp = data_generator()
    hy = hyperparams(5, X_tmp, Y_tmp, 0.01)
    W = hy.solve_W_tf()
    hy.pr_plot(W, attached_flags="TensorFAdam")
    print(W)
```

7 End

Machine Learning: fitting of sine function using polynomial