CSC 411/2515 Introduction To Machine Learning

Assignment #1

Shengze Gao

1002935942

Oct 5th, 2017

Part 1: Learning basics of regression in Python

1. Load the Boston housing data from the sklearn datasets module. Describe and summarize the data in terms of number of data points, dimensions, target, etc

Data set has in total of 506 data points and 506 targets. The original dimension of each data point is 13, since there is 13 different features in this data set. However, when we do linear regression, we will add bias to data points thus the dimension of each data point will become 14.

2. Visualization: present a single grid containing plots for each feature against the target. Choose the appropriate axis for dependent vs. independent variables. Hint: use pyplot.tight layout function to make your grid readable

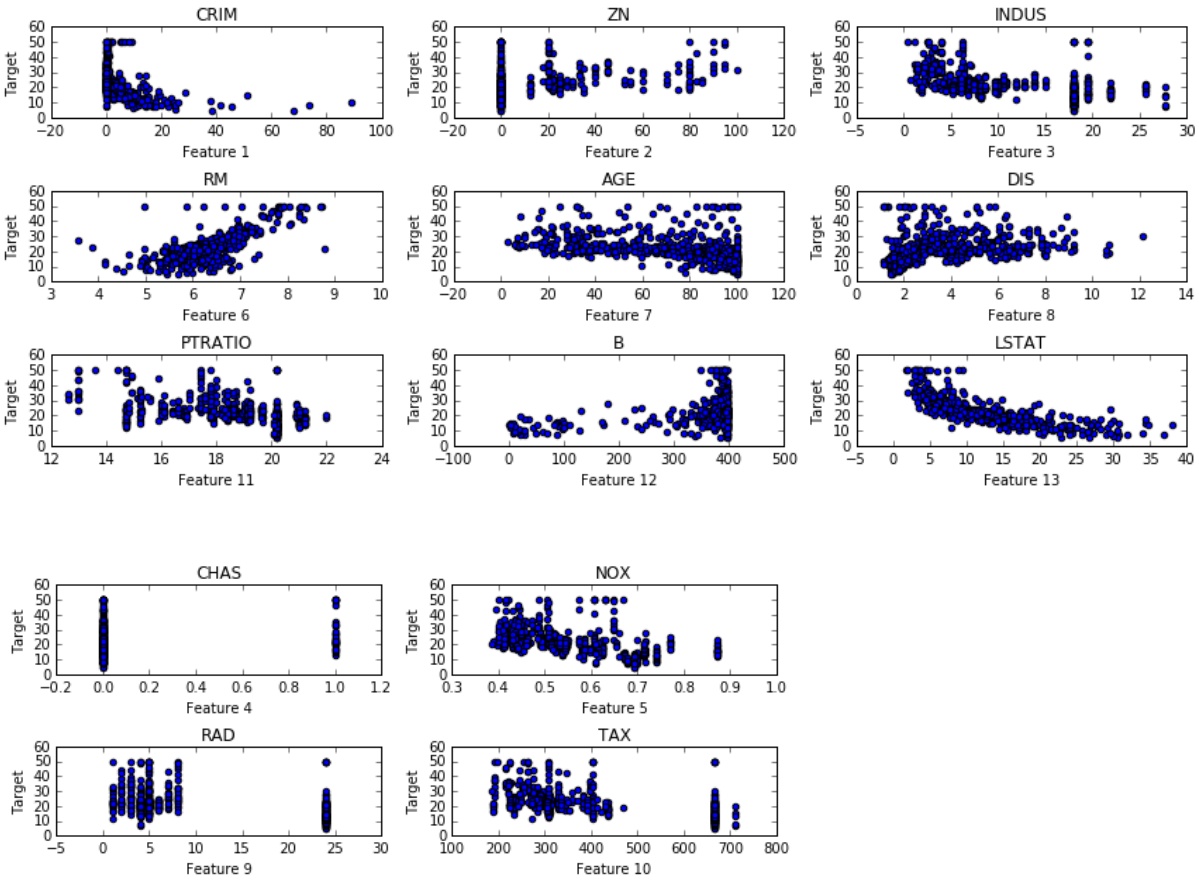

Figure 1: Each Feature(x-axis) vs. Target Value(y-axis) Plots for 13 Features.

3. **Divide your data into training and test sets, where the training set consists of 80% of the data points (chosen at random). Hint: You may find numpy.random.choice useful. Write code to perform linear regression to predict the targets using the training data. Remember to add a bias term to your model.**

Please note that the code is in the appendix section.

4. **Tabulate each feature along with its associated weight and present them in a table. Explain what the sign of the weight means in the third column ('INDUS') of this table. Does the sign match what you expected? Why?**

```
Weights for Features:
 [  3.13900155e+01  -1.48728050e-01   4.45193041e-02  -8.22618742e-04
    3.63742577e+00  -1.84649255e+01   4.42528774e+00  -1.56249289e-03
   -1.49538448e+00   3.13232190e-01  -1.11605292e-02  -8.79276712e-01
    9.36490106e-03  -4.99089047e-01]
```

Figure 2: The Weights Parameters after Linear Solving

Table: Each Feature along with its Weight

W_0(bias)	W_1(Feature 1)	W_2(Feature 2)	W_3(Feature3)	W_4(Feature4)	W_5(Feature5)	W_6(Feature6)
31.39	-0.149	0.0445	-0.000823	3.637	-18.465	4.425
W_7(Feature 7)	W_8(Feature 8)	W_9(Feature 9)	W_{10}(Feature1 0)	W_{11}(Feature1 1)	W_{12}(Feature1 2)	W_{13}(Feature1 3)
-0.00156	-1.495	0.313	-0.0112	-0.879	0.00936	-0.499

Obtained from the table above, the sign of the weight for feature "INDUS" is negative sign, which matches that we expected. By viewing the Figure1, the feature of "INDUS" is negative correlation to the target value, which make sense because the fitted weight of "INDUS" is a negative value.

5. **Test the fitted model on your test set and calculate the Mean Square Error of the result.**

The following capture is the Mean Square Error calculated with the fitted model:

```
Mean Square Error:  22.3641728803
```

Please note the corresponding code is in appendix section as well.

6. **Suggest and calculate two more error measurement metrics; justify your choice.**

By running the program two more times, we obtained three mean squared errors in total:

```
Mean Square Error:  22.3641728803

Mean Square Error:  22.1399333134

Mean Square Error:  24.0282234897
```

The second Mean Squared Error is the smallest one which means the second weight parameters fit the dataset best.

7. **Feature Selection: Based on your results, what are the most significant features that best predict the price? Justify your answer.**

NOX / Feature5

By viewing, the all the weight parameters, the weight of Feature5(“NOX”) has the largest absolute value, which means this feature may have the most significant effect on predicting target value(price).

Part 2: Locally reweighted regression

1. **Given {(x $^{(1)}$,y $^{(1)}$),..,(x $^{(N)}$,y $^{(N)}$)} and positive weights a $^{(1)}$,...,a $^{(N)}$ show that the solution to the weighted least square problem**

$$\mathbf{w}^* = \arg\min \frac{1}{2}\sum_{i=1}^{N} a^{(i)}(y^{(i)} - \mathbf{w}^T\mathbf{x}^{(i)})^2 + \frac{\lambda}{2}||\mathbf{w}||^2$$

is given by the formula

$$\mathbf{w}^* = \left(\mathbf{X}^T\mathbf{A}\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{A}\mathbf{y}$$

where X is the design matrix (defined in class) and A is a diagonal matrix where $A_{ii} = a^{(i)}$

$$L(w) = \frac{1}{2}A\|y - Xw\|^2 + \frac{\lambda}{2}\|w\|^2$$

$$= \frac{1}{2}y^TAy + \frac{1}{2}w^TX^TAXw - w^TX^TAy + \frac{\lambda}{2}w^Tw$$

$$\nabla L(w^*) = X^TAXw^* - X^TAy + \lambda w^* = 0$$

$$\Rightarrow (X^TAX + \lambda I)w^* = X^TAy$$

$$\Rightarrow w^* = (X^TAX + \lambda I)^{-1}X^TAy$$

2. **Compute w* and predicts y^ = x^Tw^*. Complete the implementation of locally reweighted least squares by providing the missing parts for q2.py.**

Please note that the code is in the appendix section.(It may takes you up to 6 mins to run)

3. **Use k-fold cross-validation to compute the average loss for different values of τ in the range [10,1000] when performing regression on the Boston Houses dataset. Plot these loss values for each choice of τ.**

The following picture shows the plot of average loss vs. different values of τ (in range of [10, 1000]):

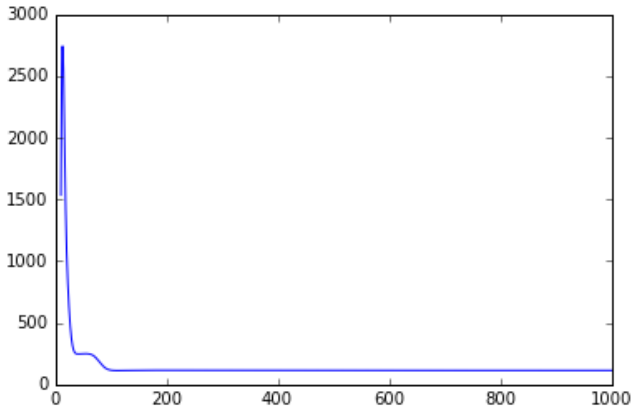

Figure 3: Plot of average loss vs. values of τ in range of 10 to 1000

4. **How does this algorithm behave when τ → ∞? When τ → 0?**

When τ approaches to infinity, viewing by Figure3, the average loss is going to approaches a constant value which is larger than 0. By analyzing the formula

$$\frac{\exp(-\|\mathbf{x}-\mathbf{x}^{(i)}\|^2/2\tau^2)}{\sum_j \exp(-\|\mathbf{x}-\mathbf{x}^{(j)}\|^2/2\tau^2)}$$

which contains τ. When τ approaches to infinity, $\exp(-||x-x^{(i)}||^2/2T^2)$ will approach to 1. Thus, the average loss will approach to a constant value larger than 0.

While when τ approaches to 0, the term, $\exp(-||x-x^{(i)}||^2/2T^2)$, will approach to 0. Thus, both the nominator and denominator of the formula above approach to 0. It will end up with the average loss approach to a constant number larger than 0, which also can be inferred from Figure3. When τ approaches to 0, the average loss also approach to a constant value larger than 0.

Part 3: Mini-batch SGD Gradient Estimator

1. **Given a set {a 1 ,...,a n } and random mini-batches I of size m, show that**

$$\mathbb{E}_{\mathcal{I}}\left[\frac{1}{m}\sum_{i\in\mathcal{I}} a_i\right] = \frac{1}{n}\sum_{i=1}^{n} a_i$$

Assume we are sampling without repetition, then we have $\frac{n}{m}$ mini-batches.

$$E_I\left[\frac{1}{m}\sum_{i\in I} a_i\right] = \frac{1}{n/m}\sum_{I=1}^{n/m}\left(\frac{1}{m}\sum_{i\in I} a_i\right)$$

$$= \frac{1}{n}\sum_{I=1}^{n/m}\sum_{i\in I} a_i$$

$$= \frac{1}{n}\sum_{i=1}^{n} a_i$$

2. **Show that** $\mathbb{E}_{\mathcal{I}}\left[\nabla L_{\mathcal{I}}(\mathbf{x}, y, \theta)\right] = \nabla L(\mathbf{x}, y, \theta)$

Assume we are sampling without repetition, then we have $\frac{n}{m}$ mini-batches.

$$E_I[\nabla L_I(x,y,\theta)] = \frac{1}{n/m}\sum_{I=1}^{n/m}\frac{\partial L_I(x,y,\theta)}{\partial\theta}$$

$$= \frac{1}{n/m}\sum_{I=1}^{n/m}\left(\frac{1}{m}\sum_{i=1}^{m}\frac{\partial L_i(x,y,\theta)}{\partial\theta}\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n}\frac{\partial L_i(x,y,\theta)}{\partial\theta}$$

$$= \nabla L(x,y,\theta)$$

3. Write, in a sentence, the importance of this result.

The result shows the correctness of stochastic gradient descent algorithm, which means we can separate a batch to several mini-batches and manipulate mini-batches with the same way of manipulating the batch to get the same result.

4. **(a) Write down the gradient, ∇L above, for a linear regression model with cost function l(x,y,θ) = (y − w T x) ² .**

$$l(x,y,\theta) = (y - w^T x)^2$$

$$L(w) = \|y - Xw\|^2 = (y-Xw)^T(y-Xw)$$

$$= y^Ty + w^TX^TXw - 2w^TX^Ty$$

$$\nabla L = 2X^TXw - 2Xy$$

(b) Write code to compute this gradient.

The capture of computed gradient of loss function is shown as follows:

```
Computed Gradient of Loss Fuction:
 [  1.34162263e+06   6.19358097e+06   1.27216974e+07   1.64445015e+07
   9.03982217e+04   7.67172065e+05   8.35968562e+06   9.62373069e+07
   4.75243363e+06   1.52651113e+07   6.00440771e+08   2.50993997e+07
   4.75603158e+08   1.81402515e+07]
```

Please note that the code is in the appendix section.

5. **Using your code from the previous section, for m = 50 and K = 500 compute**

$$\frac{1}{K}\sum_{k=1}^{K}\nabla L_{\mathcal{I}_k}(\mathbf{x}, y, \theta)$$

, where I k is the mini-batch sampled for the kth time.

Randomly initialize the weight parameters for your model from a N(0,I) distribution. Compare the value you have computed to the true gradient, ∇L, using both the squared distance metric and cosine similarity. Which is a more meaningful measure in this case and why?

Cosine similarity is the more meaningful measure in this case.

The following capture showing the result of comparing using both squared distance and cosine similarity:

```
squared distance:  4.85435696815689e+17
cosine similarity:  0.99999960033
```

Since the squared distance is not only comparing the direction of two vectors, but also comparing the absolute value of two vectors. There is impossible for the two gradient vectors have similar magnitude in this case. Thus, using cosine similarity to comparing the directions of these two vectors makes more sense. If they got similar directions, they have similar ratio among different weight parameters.

6. **For a single parameter, w_j , compare the sample variance, σ_j , of the mini-batch gradient estimate for values of m in the range [0,400] (using K = 500 again). Plot log σ_j against log m.**

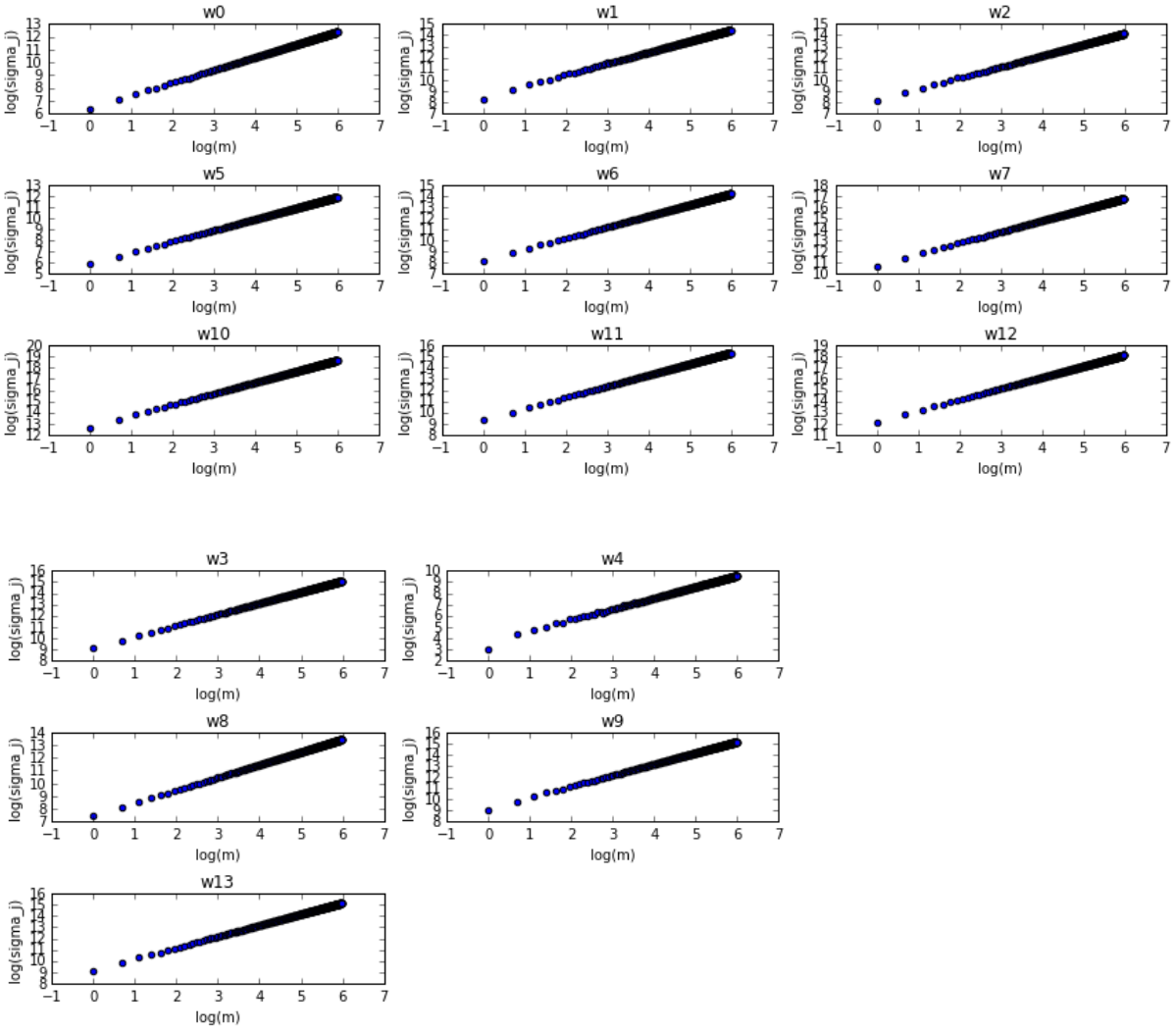

Figure 4: log m (x-axis) vs. log σ_j (y-axis) where σ_j is the variance of gradient corresponding to each parameter w_j

Appendix: (Please note all the code is submitted in .py files also)

Problem1:

```
from sklearn import datasets
import matplotlib.pyplot as plt
import numpy as np

def load_data():
    boston = datasets.load_boston()
    X = boston.data
    y = boston.target
    features = boston.feature_names
    data_num = y.shape[0]
    data_dim = X.shape[1]
    return X,y,features,data_num,data_dim



def visualize(X, y, features):
    plt.figure(figsize=(20, 5))
    feature_count = X.shape[1]

    # i: index
    for i in range(feature_count):
        #TODO: Plot feature i against y
        ax = plt.subplot(3, 5, i + 1)
        ax.set_title(features[i])
        ax.scatter(X[:,i],y)

    plt.tight_layout()
    plt.show()
```

```
def fit_regression(X,Y):
    #TODO: implement linear regression
    # Remember to use np.linalg.solve instead of inverting!
    #raise NotImplementedError()
    X = np.concatenate((np.ones((X.shape[0],1)),X),axis=1)
    w, residual, rank, s = np.linalg.lstsq(X, Y)
    return w

def MeanSquareError(X, Y, w):
    X = np.concatenate((np.ones((X.shape[0],1)),X),axis=1)
    y_predict = np.matmul(X, w)
    mse = sum(np.square(Y - y_predict)) / (X.shape[0])
    return mse

def main():
    # Load the data
    X, y, features, num, dim = load_data()
    print("Features: {}".format(features))

    print(str(num) + "  " + str(dim))

    # Visualize the features
    visualize(X, y, features)

    #TODO: Split data into train and test
    train_data = []
    train_target = []
    test_data = []
    test_target = []
    for i in range(int(num*0.8)):
        readyToRemove = np.random.choice(len(X))
        train_data.append(X[readyToRemove])
```

```
        X = np.delete(X, (readyToRemove), axis = 0)
        train_target.append(y[readyToRemove])
        y = np.delete(y, (readyToRemove), axis = 0)
    test_data = X
    test_target = y
    train_data = np.array(train_data)
    test_data = np.array(test_data)
    train_target = np.array(train_target)
    test_target = np.array(test_target)

    # Fit regression model
    w = fit_regression(train_data, train_target)
    print("Weights for Features: \n",w)

    # Compute fitted values, MSE, etc.
    mse = MeanSquareError(test_data, test_target, w)
    print("Mean Square Error: ", mse)


if __name__ == "__main__":
    main()
```

Problem2:

```
from __future__ import print_function
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_boston
np.random.seed(0)

# load boston housing prices dataset
boston = load_boston()
```

```
x = boston['data']
N = x.shape[0]
x = np.concatenate((np.ones((506,1)),x),axis=1) #add constant one feature - no bias needed
d = x.shape[1]
y = boston['target']

idx = np.random.permutation(range(N))

#helper function
def l2(A,B):
    '''
    Input: A is a Nxd matrix
           B is a Mxd matirx
    Output: dist is a NxM matrix where dist[i,j] is the square norm between A[i,:] and B[j,:]
    i.e. dist[i,j] = ||A[i,:]-B[j,:]||^2
    '''
    #A = np.expand_dims(A, axis=0)
    #B = np.expand_dims(B, axis=0)
    A_norm = (A**2).sum(axis=1).reshape(A.shape[0],1)
    B_norm = (B**2).sum(axis=1).reshape(1,B.shape[0])
    dist = A_norm+B_norm-2*A.dot(B.transpose())
    return dist

#helper function
def run_on_fold(x_test, y_test, x_train, y_train, taus):
    '''
    Input: x_test is the N_test x d design matrix
           y_test is the N_test x 1 targets vector
           x_train is the N_train x d design matrix
           y_train is the N_train x 1 targets vector
           taus is a vector of tau values to evaluate
    output: losses a vector of average losses one for each tau value
```

```
    '''
    N_test = x_test.shape[0]
    losses = np.zeros(taus.shape)
    for j,tau in enumerate(taus):
        print("        ",j)
        predictions =  np.array([LRLS(x_test[i,:].reshape(1,d),x_train,y_train, tau) \
                    for i in range(N_test)])
        losses[j] = ((predictions-y_test)**2).mean()
    return losses



#to implement
def LRLS(test_datum,x_train,y_train, tau,lam=1e-5):
    '''
    Input: test_datum is a dx1 test vector
           x_train is the N_train x d design matrix
           y_train is the N_train x 1 targets vector
           tau is the local reweighting parameter
           lam is the regularization parameter
    output is y_hat the prediction on test_datum
    '''
    ## TODO
    x_mean = np.expand_dims(x_train.mean(axis = 0), 0)
    #print("xmean", x_mean.shape)
    #exp_sum = []
    #for i in range(x_train.shape[0]):
    #    exp_sum.append(np.squeeze(l2(x_mean, x_train[i,:])))
    exp_sum = np.squeeze(l2(x_mean, x_train))
    #print(exp_sum.shape)
    exp_sum = exp_sum * (-1/(2*tau*tau))
    B_max = np.amax(exp_sum)
    exp_sum = np.exp(exp_sum - B_max)
```

```
    summation = sum(exp_sum)

    a = exp_sum / summation
    #for i in range(exp_sum.shape[0]):
    #    a.append(exp_sum[i] / summation)
    #a = np.array(a)

    la = np.zeros((14, 14), float)
    np.fill_diagonal(la, lam)

    A = np.diag(a)
    One = np.matmul(np.matmul(np.transpose(x_train), A), x_train) + la
    Two = np.matmul(np.matmul(np.transpose(x_train), A), y_train)

    w, residual, rank, s = np.linalg.lstsq(One, Two)

    y_hat = np.matmul(w, np.transpose(test_datum))
    return y_hat



def run_k_fold(x,y,taus,k):
    '''
    Input: x is the N x d design matrix
           y is the N x 1 targets vector
           taus is a vector of tau values to evaluate
           K in the number of folds
    output is losses a vector of k-fold cross validation losses one for each tau value
    '''
    ## TODO
    x_train = []
    y_train = []
    x_test = []
```

```
    y_test = []
    losses = np.zeros(taus.shape[0])
    step = int(x.shape[0]/k)
    for subsample_i in range(k):
        print(subsample_i)
        if subsample_i < 4:
            x_test = x[subsample_i*step:(subsample_i+1)*step]
            x_train = np.concatenate([x[:subsample_i*step], x[(subsample_i+1)*step:]])
            y_test = y[subsample_i*step:(subsample_i+1)*step]
            y_train = np.concatenate([y[:subsample_i*step], y[(subsample_i+1)*step:]])
        else:
            x_test = x[subsample_i*step:]
            x_train = x[:subsample_i*step]
            y_test = y[subsample_i*step:]
            y_train = y[:subsample_i*step]
        lo = run_on_fold(x_test, y_test, x_train, y_train, taus)
        losses = losses + lo


    losses = losses/k
    return losses



if __name__ == "__main__":
    # In this excersice we fixed lambda (hard coded to 1e-5) and only set tau value. Feel free to play
with lambda as well if you wish
    taus = np.logspace(1.0,3,200)
    losses = run_k_fold(x,y,taus,k=5)
    plt.plot(taus, losses)
    print("min loss = {}".format(losses.min()))
```

Problem3:

```
import numpy as np
from sklearn.datasets import load_boston
import matplotlib.pyplot as plt

K = 500
BATCHES = 50
boston = load_boston()
X = boston['data']
X = np.concatenate((np.ones((506,1)),X),axis=1) #add constant one feature - no bias needed
y = boston['target']

class BatchSampler(object):
    '''
    A (very) simple wrapper to randomly sample batches without replacement.

    You shouldn't need to touch this.
    '''

    def __init__(self, data, targets, batch_size):
        self.num_points = data.shape[0]
        self.features = data.shape[1]
        self.batch_size = batch_size

        self.indices = np.arange(self.num_points)

    def random_batch_indices(self, m=None):
        '''
        Get random batch indices without replacement from the dataset.

        If m is given the batch will be of size m. Otherwise will default to the class initialized value.
        '''
        if m is None:
```

```
            indices = np.random.choice(self.indices, self.batch_size, replace=False)
        else:
            indices = np.random.choice(self.indices, m, replace=False)
        return indices

    def get_batch(self, m=None):
        '''
        Get a random batch without replacement from the dataset.

        If m is given the batch will be of size m. Otherwise will default to the class initialized value.
        '''
        indices = self.random_batch_indices(m)
        X_batch = np.take(X, indices, 0)
        y_batch = y[indices]
        return X_batch, y_batch



def load_data_and_init_params():
    '''
    Load the Boston houses dataset and randomly initialise linear regression weights.
    '''
    print('------ Loading Boston Houses Dataset ------')
    #X, y = load_boston(True)
    features = X.shape[1]

    # Initialize w
    w = np.random.randn(features)

    print("Loaded...")
    print("Total data points: {0}\nFeature count: {1}".format(X.shape[0], X.shape[1]))
    print("Random parameters, w: {0}".format(w))
    print('-----------------------------------------\n\n\n')
```

```
    return X, y, w


def l2(A,B):
    '''
    Input: A is a Nxd matrix
           B is a Mxd matirx
    Output: dist is a NxM matrix where dist[i,j] is the square norm between A[i,:] and B[j,:]
    i.e. dist[i,j] = ||A[i,:]-B[j,:]||^2
    '''
    A_norm = (A**2).sum(axis=1).reshape(A.shape[0],1)
    B_norm = (B**2).sum(axis=1).reshape(1,B.shape[0])
    dist = A_norm+B_norm-2*A.dot(B.transpose())
    return dist


def cosine_similarity(vec1, vec2):
    '''
    Compute the cosine similarity (cos theta) between two vectors.
    '''
    dot = np.dot(vec1, vec2)
    #print(dot)
    sum1 = np.sqrt(np.dot(vec1, vec1))
    sum2 = np.sqrt(np.dot(vec2, vec2))

    return dot / (sum1 * sum2)


#TODO: implement linear regression gradient
def lin_reg_gradient(X, y, w):
    '''
    Compute gradient of linear regression model parameterized by w
    '''
    #raise NotImplementedError()
```

```
    grad = 2 * np.matmul(w, np.matmul(np.transpose(X), X)) - 2 * np.matmul(y, X)
    return grad


def main():
    # Load data and randomly initialise weights
    X, y, w = load_data_and_init_params()
    # Create a batch sampler to generate random batches from data
    batch_sampler = BatchSampler(X, y, BATCHES)

    #w_ = fit_regression(X, y)
    computed_grad = lin_reg_gradient(X, y, w)

    print("Computed Gradient of Loss Fuction: \n", computed_grad)

    #sigma = np.zeros(K, float)
    # Example usage
    batchGradSum = []
    for i in range(K):
        X_b, y_b = batch_sampler.get_batch()
        #w_l = fit_regression(X_b, y_b)
        batch_grad = lin_reg_gradient(X_b, y_b, w)
        batchGradSum.append(batch_grad)

    #sigma = np.var(batchGradSum, 0)
    batch_grad_sum = np.sum(batchGradSum, axis=0)/K
    print("Gradient of Loss Fuction Applying Mini-Batch: \n", batch_grad_sum)

    squared_dist = np.squeeze(l2(np.expand_dims(computed_grad, axis=0),
np.expand_dims(batch_grad_sum, axis=0)))
    cosine_sim = cosine_similarity(computed_grad, batch_grad_sum)

    print("squared distance: ", squared_dist)
    print("cosine similarity: ", cosine_sim)
```

```
    sigma = []
    for m in np.arange(1, 401):
        batchGradSum_ = []
        for i in range(K):
            X_bb, y_bb = batch_sampler.get_batch(m)
            batch_grad_ = lin_reg_gradient(X_bb, y_bb, w)
            batchGradSum_.append(batch_grad_)
        batch_grad_sum_ = np.sum(batchGradSum_, axis = 0)/K
        sigma.append(batch_grad_sum_)
    sigma = np.log(np.array(sigma))

    mm = np.log(np.arange(1,401))
    plt.figure(figsize=(20, 5))
    # i: index
    for i in range(14):
        #TODO: Plot feature i against y
        ax = plt.subplot(3, 5, i + 1)
        ax.set_title("w"+str(i))
        ax.scatter(mm,sigma[:,i])

    plt.tight_layout()
    plt.show()

if __name__ == '__main__':
    main()
```