Q2:

1.

Credit to Stanford Machine Learning Lecture2 on Youtube, at 1:06:00

$L(w) = \frac{1}{2}(A^{\frac{1}{2}}(y - Xw))^2 + \frac{\lambda}{2}\|w\|_2^2$

$= \frac{1}{2}\|A^{\frac{1}{2}}y - A^{\frac{1}{2}}Xw\|_2^2 + \frac{\lambda}{2}\|w\|_2^2$

$= \frac{1}{2}(A^{\frac{1}{2}}y - A^{\frac{1}{2}}Xw)^T(A^{\frac{1}{2}}y - A^{\frac{1}{2}}Xw) + \frac{\lambda}{2}\|w\|_2^2$

$= \frac{1}{2}(y^TA^{\frac{1}{2}} - w^TX^TA^{\frac{1}{2}})(A^{\frac{1}{2}}y - A^{\frac{1}{2}}Xw) + \frac{\lambda}{2}\|w\|_2^2$

$= \frac{1}{2}(y^TAy + w^TX^TAXw - y^TAXw - w^TX^TAy) + \frac{\lambda}{2}\|w\|_2^2$

$\nabla L(w) = \frac{1}{2}\nabla_w(y^TAy + w^TX^TAXw - y^TAXw - w^TX^TAy + \frac{\lambda}{2}\|w\|_2^2)$

$= \frac{1}{2}\nabla_w(y^TAXw + w^TX^TAXw - w^TX^TAy) + \lambda Iw$

$= -\frac{1}{2}\nabla_w tr(y^TAXw) + \frac{1}{2}\nabla_w tr(w^TX^TAXw) - \frac{1}{2}\nabla_w tr(w^TX^TAy) + \lambda Iw$

$= -\frac{1}{2}X^TAy + X^TAXw - \frac{1}{2}X^TAy + \lambda Iw$

$= -X^TAy + X^TAXw + \lambda Iw = 0$

So, $(X^TAX + \lambda I)w = X^TAy$

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

2.

```
# A helper work on formula from q2.1, generate w^*
def formula_w(x_train,y_train,lam,A):

    # All formulas below are all according to assignment handout.

    # (The transpose of x_train dot product A) = temp, which is (X^Y)A in handout.
    temp = np.dot(x_train.T, A)

    # (X^Y)AX in handout
    element1 = np.dot(temp, x_train)
    # (X^Y)Ay in handout
    element2 = np.dot(temp, y_train)

    # np.eye will return a 2-D array with ones on the diagonal and zeros elsewhere.
    # I is an array where all elements are equal to zero.
    I = np.eye(element1.shape[0])

    # λI in handout
    element3 = lam * I

    # (X^Y)AX + λI in handout
    element4 = element1 + element3

    # similar to q1
    # handout formular is w^* = (((X^Y)AX + λI)^(-1))((X^Y)Ay)
    # which can be show as "w^* = (a^(-1))b" or "a(w^*) = b"
    # at here, element4 = a = (X^Y)AX + λI, element2 = b = (X^Y)Ay
    # so use np.linalg.solve to w^* as well as q1
    w = np.linalg.solve(element4, element2)

    return w
```

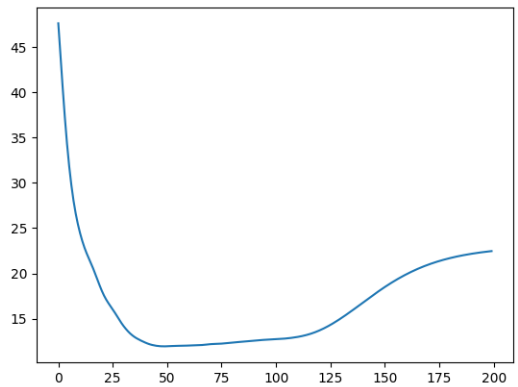
```python
# A helper work on formula from q2.2, generate A, A_ii = a^(i)
def formula_ai(test_datum,x_train,tau):

    # All formulas below are all according to assignment handout.

    # Call helper function l2 to get matrix dist with test_datum and x_train
    dist = l2(test_datum, x_train)

    # np.divide(x1, x2) can divide arguments element-wise and return x1/x2.
    # Returns a scalar if both x1 and x2 are scalars.
    # "-1 * (l2(test_datum, x_train))" is the Dividend array.
    # "2 * tau * tau" is the Divisor array.
    array = np.divide(-1 * dist, 2 * tau * tau)

    # Compute the log of the sum of exponentials of input elements.
    sum_log = mis.logsumexp(array)

    # Calculate the exponential of all elements in the input array and get a output array,
    # element-wise exponential of array.
    exp1 = np.exp(array)
    exp2 = np.exp(sum_log)

    # Compute distance-based weights for each training example as assignment handout.
    A = np.divide(exp1, exp2)

    return A
```

```python
#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

    # All formulas below are all according to assignment handout and lecture slides.

    A = formula_ai(test_datum,x_train,tau)

    # Extract a diagonal array.
    A = np.diag(A[0, :])

    w = formula_w(x_train,y_train,lam,A)

    # predict on test_datum
    y_hat = np.dot(test_datum, w)

    return y_hat
    ## TODO
```

```python
# A helper to update losses by setting testing and training sets.
# There is no return value in this function.
def update_losses(x,y,taus,k,fold_size,indices,losses):

    # loop over all folds
    for i in range(k):

        # Print data for checking and debugging
        print("Fold number: {}".format(i))

        # find test set in all indices with a range
        test_sets = indices[(i * fold_size) : ((i + 1) * fold_size)]

        # except test set, the rest is the train set
        train_set = [s for s in indices if s not in test_sets]

        # find out what data should be tested or trained
        test_X = x[test_sets, :]
        test_y = y[test_sets]
        train_X = x[train_set, :]
        train_y = y[train_set]

        # update losses by helper
        losses[i, :] = run_on_fold(test_X, test_y, train_X, train_y, taus)
```

```python
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

    data_num = x.shape[0]
    fold_size = int(np.round(data_num / k))

    # Print data for checking and debugging
    print("Total Data number: ", data_num)
    print("Fold size: ", fold_size)

    # losses is a new array of given shape, without initializing entries
    losses = np.empty([k, len(taus)])

    # randomly permute np.arange(range(data_num)) as random index array
    indices = np.random.permutation(range(data_num))

    update_losses(x, y, taus, k, fold_size, indices, losses)

    # arithmetic mean along the specified axis
    averages = np.mean(losses, axis=0)

    return averages
    ## TODO
```

3.




```
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6 /Users/bill/Desktop/CSC411/a1/q2.py
Total Data number:  506
Fold size:  101
Fold number: 0
Fold number: 1
Fold number: 2
Fold number: 3
Fold number: 4
minimunm loss = 11.935513816961372

Process finished with exit code 0
```

4. How does this algorithm behave when $\tau \to \infty$? When $\tau \to 0$?
As images above, when $\tau \to \infty$, the loss $\to$ a specific number which is greater than 20 and less than 25. By further checking, I found out it is approximate to a float number between 24 and 25.
When $\tau \to 0$, the loss $\to \infty$.