Stat 542 HW6

Harris Nisar (nisar2)

March 24, 2022

1 Question 1 [35 Points] Local Linear Regression

We have implemented the Nadaraya-Watson kernel estimator in HW 6. In this question, we will investigate a local linear regression:

$$\widehat{f}(x) = \widehat{\beta}_0(x) + \widehat{\beta}_1(x)\,x,$$

where x is a testing point. Local coefficients $\widehat{\beta}_r(x)$ for $r = 0, 1$ are obtained by minimizing the object function

$$\underset{\beta_0(x),\,\beta_1(x)}{\text{minimize}} \quad \sum_{i=1}^{n} K_\lambda(x, x_i)\left[y_i - \beta_0(x) - \beta_1(x)x_i\right]^2.$$

In this question, we will use the Gaussian kernel $K(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}}$.

a) [20 pts] Write a function `myLocLinear(trainX, trainY, testX, lambda)`, where `lambda` is the bandwidth and `testX` is all testing samples. This function returns predictions on `testX`. The solution of $\beta_0(x)$ and $\beta_1(x)$ can be obtained by fitting a weighted linear regression. The formula is provided on Page 25 of our lecture note.

[2]:
```
def wm(x, trainX, lam):
    # m is the No of training examples .
    m = trainX.shape[0]

    # Initialising W as an identity matrix.
    w = np.mat(np.eye(m))

    # Calculating weights for all training examples [x(i)'s].
    for i in range(m):
        xi = trainX[i]
        d = (-2 * lam * lam)
        w[i, i] = (1/math.sqrt(2*math.pi)) * np.exp(np.dot((xi-x), (xi-x).T)/d)

    return w

def myLocLinear(trainX, trainY, testX, lam):
    preds = []
    for point in testX:
        # m = number of training examples.
        m = trainX.shape[0]
```

```
        # Appending a cloumn of ones in X to add the bias term.
        X_ = np.append(trainX, np.ones(m).reshape(m,1), axis=1)

        # point is the x where we want to make the prediction.
        point_ = np.append(point, np.ones(1).reshape(1,1))

        # Calculating the weight matrix using the wm function we wrote
        w = wm(point_, X_, lam)
        # Calculating parameter theta using the formula.
        theta = np.linalg.pinv(X_.T @ w @ X_)@ X_.T @ w @ trainY

        # Calculating predictions.
        pred = np.dot(theta,point_.T)

        preds.append(pred)
    return np.array(preds)
```

b) [15 pts] Fit a local linear regression with our given training data. The testing data are generated using the code given below. Try a set of bandwidth $\lambda = 0.05, 0.1, \ldots, 0.55, 0.6$ when calculating the kernel function.

- Provide a plot of testing MSE vs λ. Does your plot show a "U" shape?
- Report the best testing MSE with the corresponding λ.
- Plot three figures of your fitted testing data curve, with $\lambda = 0.05, 0.25$, and 0.5. Add the true function curve (see the following code for generating the truth, I modified this to simply save train, testX, and testY to .csv to import in python) and the training data points onto this plot. Label each λ and your curves. Comment on the the shape of fitted curves as your λ changes.

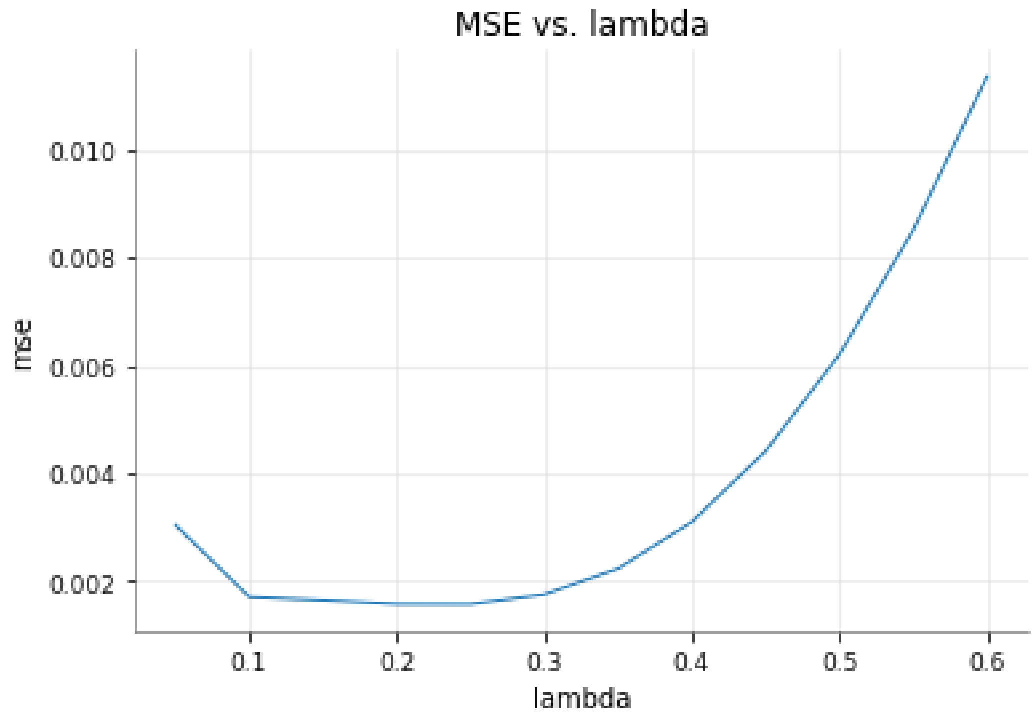
```
train = read.csv('/Users/harrisnisar/Downloads/hw7_Q1_train.csv')
testX = 2 * pi * seq(0, 1, by = 0.01)
testY = sin(testX)
write.csv(train,'/Users/harrisnisar/Documents/Stat 542/HW7/data/problem1_trainX.csv')
write.csv(testX,'/Users/harrisnisar/Documents/Stat 542/HW7/data/problem1_testX.csv')
write.csv(testY,'/Users/harrisnisar/Documents/Stat 542/HW7/data/problem1_testY.csv')
```

[3]:
```
train = np.array(pd.read_csv('./data/problem1_trainX.csv'))[:,1:]
trainX = train[:,0].reshape(train.shape[0],-1)
trainY = train[:,1]

testX = np.array(pd.read_csv('./data/problem1_testX.csv'))[:,1:]
testY = np.array(pd.read_csv('./data/problem1_testY.csv'))[:,1]
```

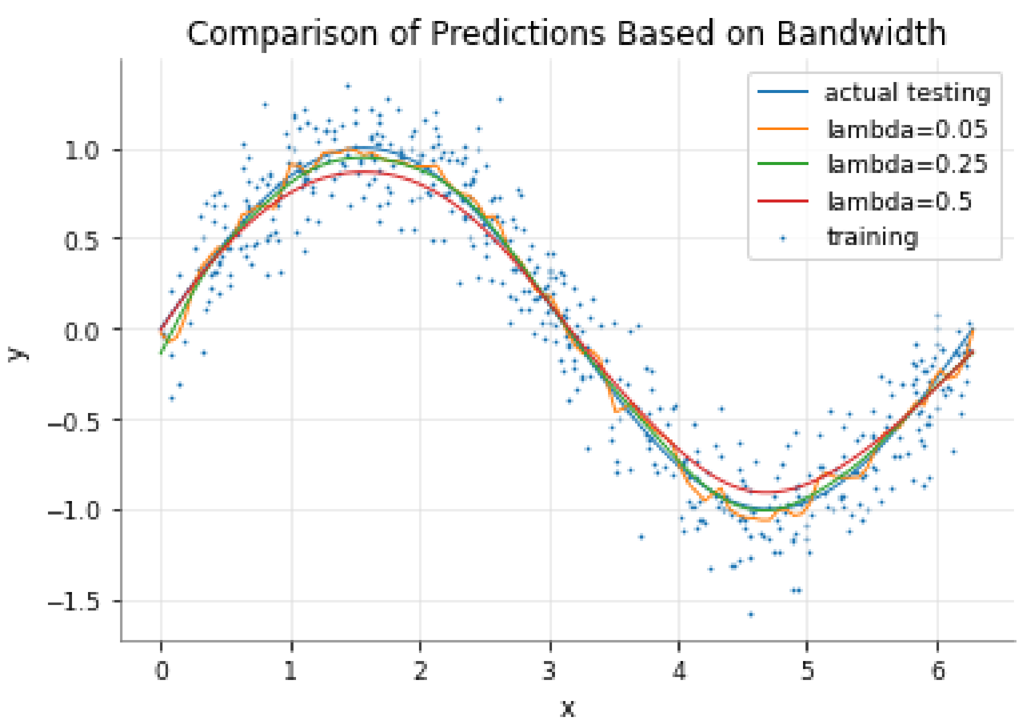
[4]:
```
lams = np.arange(0.05,0.65,0.05)
preds_for_lams = []
for lam in lams:
    preds = myLocLinear(trainX, trainY, testX, lam)
    preds_for_lams.append(preds)
```

```
preds_for_lams = np.array(preds_for_lams).reshape(12,101)
```

[5]:
```
test_mses = []
def mse(a,b):
    return np.mean(np.power(a-b,2))

for pred in preds_for_lams:
    test_mses.append(mse(pred,testY))

plt.plot(lams,test_mses)
plt.title('MSE vs. lambda')
plt.xlabel('lambda')
plt.ylabel('mse')
plt.show()
print(f'Best lambda: {lams[np.argmin(test_mses)]} with MSE: {test_mses[np.
 ↪argmin(test_mses)]}')
plt.scatter(trainX, trainY,s=0.5, label='training')
plt.plot(testX, testY, label='actual testing')
plt.plot(testX, preds_for_lams[0,:], label='lambda=0.05')
plt.plot(testX, preds_for_lams[4,:], label='lambda=0.25')
plt.plot(testX, preds_for_lams[9,:], label='lambda=0.5')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Comparison of Predictions Based on Bandwidth')
plt.legend()
plt.show()
```




```
Best lambda: 0.25 with MSE: 0.0015753600683333387
```



2 Question 2 [35 Points] Linear Discriminant Analysis

For both question 2 and 3, you need to write your own code. We will use the handwritten digit recognition data from the `ElemStatLearn` package. We only consider the train-test split, with the pre-defined `zip.train` and `zip.test`. Simply use `zip.train` as the training data, and `zip.test` as the testing data for all evaluations and tuning. No cross-validation is needed in the training process.

- The data consists of 10 classes: digits 0 to 9 and 256 features (16×16 grayscale image).
- More information can be attained by code `help(zip.train)`.

a. [10 pts] Estimate the mean, covariance matrix of each class and pooled covariance matrix. Basic built-in R functions such as `cov` are allowed. Do NOT print your results.

[6]:
```
# data loading
train = np.array(pd.read_csv('./data/problem2_train.csv'))[:,1:]
test = np.array(pd.read_csv('./data/problem2_test.csv'))[:,1:]

train_x = train[:,1:]
train_y = train[:,0]
test_x = test[:,1:]
test_y = test[:,0]

possible_labels = [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0]
n = np.shape(train_x)[0]
k = len(possible_labels)
```

[7]:
```
mean_list = []
#pooled_cov = np.cov(train_x.T)
pooled_cov = []
class_cov_list = []
class_prop_list = []
nks = []
for i in possible_labels:
    nk = np.sum(train_y==i)
    nks.append(nk)
    train_x_subset = train_x[train_y == i,:]
    mean_list.append(np.mean(train_x_subset,axis=0))
    class_cov_list.append(np.cov(train_x_subset.T))
    class_prop_list.append(train_x_subset.shape[0]/train_x.shape[0])
    pooled_cov.append((np.cov(train_x_subset.T) * (nk-1)))
mean_list = np.array(mean_list)
class_cov_list = np.array(class_cov_list)
class_prop_list = np.array(class_prop_list)
pooled_cov = np.array(pooled_cov)
# pooled_cov = np.mean(class_cov_list, axis=0)
```

```python
pooled_cov = np.sum(pooled_cov,axis=0) / (n-k)
```

b. [15 pts] Write your own linear discriminate analysis (LDA) code following our lecture note. To perform this, you should calculate μ_k, π_k, and Σ from the data. You may consider saving μ_k's and π_k's as a list (with 10 elements in each list).

You are not required to write a single function to perform LDA, but you could consider defining a function as `myLDA(testX, mu_list, sigma_pool)`, where `mu_list` is the estimated mean vector for each class, and `sigma_pool` is the pooled variance estimation. This function should return the predicted class based on comparing **discriminant functions** $\delta_k(x) = w_k^T x + b_k$ given on page 32 of the lecture note.

[8]:
```python
def myLDA(testX, mu_list, sigma_pool, class_probs):
    sigma_pool_inv = np.linalg.inv(sigma_pool)
    wks = []
    bks = []
    for i in range(len(class_probs)):
        class_mean = mu_list[i,:]
        class_prop = class_probs[i]
        wk = sigma_pool_inv @ class_mean
        bk = (-1/2) * class_mean.T @ sigma_pool_inv @ class_mean + np.
↪log(class_prop)
        wks.append(wk)
        bks.append(bk)
    wks = np.array(wks)
    bks = np.array([np.array(bks)]).T
    preds = wks@testX.T + bks
    # print(preds.shape)
    return (wks, bks, np.argmax(preds,axis=0))
```

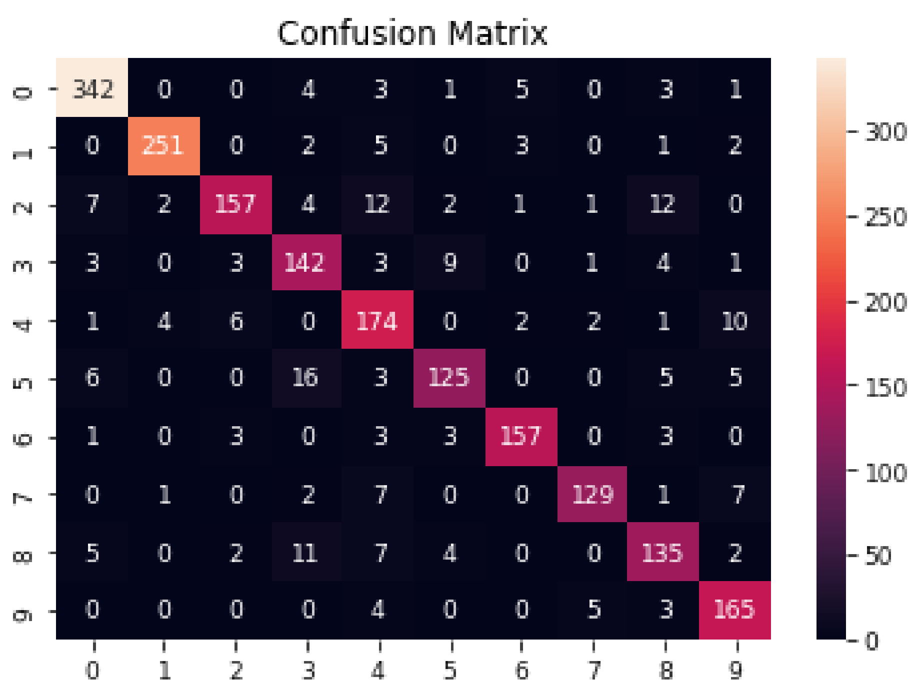
c. [10 pts] Fit LDA model on the training data and predict with the testing data.

- Report the first 5 entries of the w coefficient vector and b for digit `0`.
- Report a 10×10 confusion matrix, where each **column** is true digit and each **row** is your predicted digit. You can use the `table()` function in R.
- Report a table of misclassification rate of each (true) digit. Hence, this is the $1-$ sensitivity of each digit in a multi-class problem. Only keep the first three digits after the decimal point for the rate. Also report the overall mis-classification rate.

[9]:
```python
wks, bks, preds = myLDA(test_x, mean_list, pooled_cov, class_prop_list)
print(f'First 5 entries of w0: {wks[0,:5]}')
print(f'b0: {bks[0,:5][0]}')
accuracy = np.sum(preds == test_y)/test_y.shape[0]
cm = confusion_matrix(test_y, preds)
f = sns.heatmap(cm, annot=True, fmt='d')
f.set_title('Confusion Matrix')

# to-do: MISCLASSIFICATION RATE PER DIGIT
TP = np.diag(cm)
```

```
FN = cm.sum(axis=1) - np.diag(cm)
sensitivity = (TP / (TP + FN))
misclassification_rate = 1 - sensitivity
misclassification_rate_formatted = [format(mcr, ".3f") for mcr in␣
 ↪misclassification_rate]

dict = {
        'Misclassification Rate' : misclassification_rate_formatted,
       }

df = pd.DataFrame(dict)

display(df.T)
print(f'Overall Misclassification Rate: {(1-accuracy):0.3f}')
```

```
First 5 entries of w0: [-549.55302059   68.57504727  -39.11263175   -3.09565875
 -9.65674984]
b0: -1156.4428919058962

                            0      1      2      3      4      5      6  \
Misclassification Rate  0.047  0.049  0.207  0.145  0.130  0.219  0.076

                            7      8      9
Misclassification Rate  0.122  0.187  0.068

Overall Misclassification Rate: 0.115
```



3 Question 3 [30 points] Regularized quadratic discriminate analysis

QDA uses a quadratic discriminant function. However, QDA does not work directly in this example because we do not have enough samples to provide an invertible sample covariance matrix for each digit. An alternative idea to fix this issue is to consider a regularized QDA method, which uses

$$\widehat{\Sigma}_k(\alpha) = \alpha\widehat{\Sigma}_k + (1-\alpha)\widehat{\Sigma}$$

instead of Σ_k. Then, they are used in the decision rules given in page 36 of lecture notes. Complete the following questions

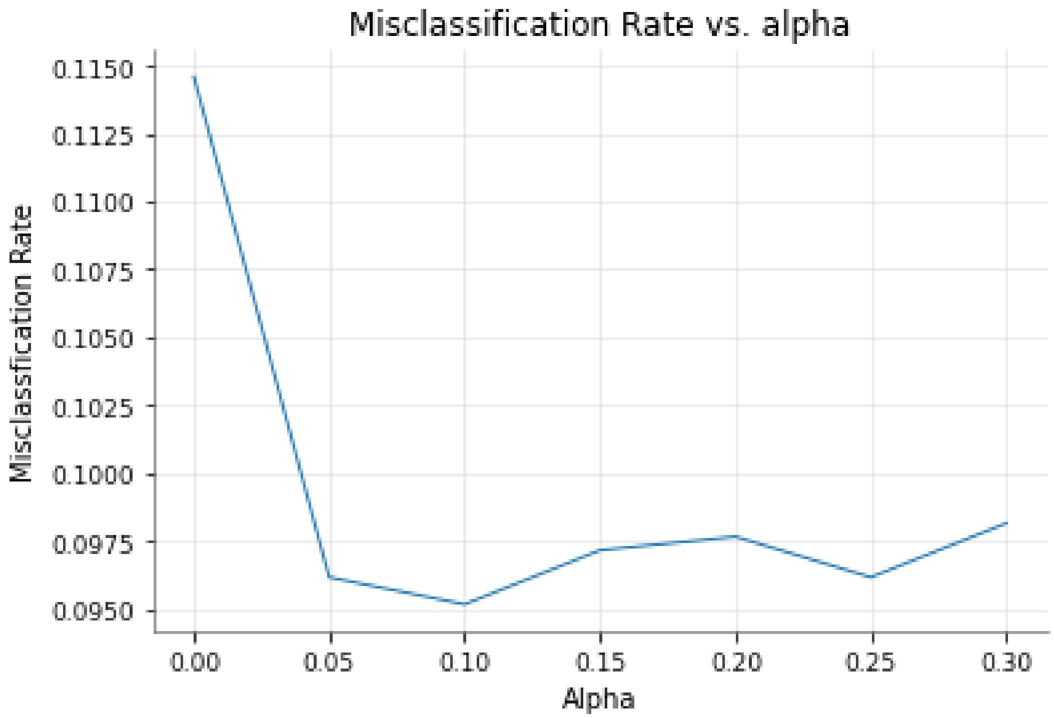
a. [20 pts] Write your own function `myRQDA(testX, mu_list, sigma_list, sigma_pool, alpha)`, where `allpha` is a scaler `alpha` and `testX` is your testing covariate matrix. And you may need a new `sigma_list` for all the Σ_k. This function should return a vector of predicted digits.

[10]:
```python
def myRQDA(testX, mu_list, sigma_list, sigma_pool, alpha, class_probs):
    sigma_pool_inv = np.linalg.inv(sigma_pool)
    Wks = []
    wks = []
    bks = []
    for i in range(len(class_probs)):
        class_mean = mu_list[i,:]
        class_cov = sigma_list[i,:,:]
        class_cov_reg = alpha * class_cov + (1-alpha) * sigma_pool
        class_cov_reg_inv = np.linalg.inv(class_cov_reg)
        class_prop = class_probs[i]
        Wk = (-1/2)*class_cov_reg_inv
        wk = class_cov_reg_inv @ class_mean
        bk = (-1/2) * class_mean.T @ class_cov_reg_inv @ class_mean - (1/2)*(np.
↪linalg.det(class_cov_reg))+ np.log(class_prop)
        Wks.append(Wk)
        wks.append(wk)
        bks.append(bk)
    Wks = np.array(Wks)
    wks = np.array(wks)
    bks = np.array(bks)
    preds = []
    for x in testX:
        pred = []
        x = x.reshape(-1,1)
        for k in range(len(class_probs)):
            Wk = Wks[k,:,:]
            wk = wks[k,:]
```

```
            bk = bks[k]
            p = x.T @ Wk @ x + wk.T @ x + bk
            pred.append(p[0][0])
        preds.append(np.array(pred))
    preds = np.array(preds).T
    return (wks, bks, np.argmax(preds,axis=0))
```

b. [10 pts] Perform regularized QDA with the following sequence of α values. Plot the testing error (misclassification rate) against alpha. Report the minimum testing error and the corresponding α.

[11]:
```
alphas = np.arange(0,.35,0.05)
alphas
mc_rates = []
for alpha in alphas:
    wks, bks, preds = myRQDA(test_x, mean_list, class_cov_list, pooled_cov,␣
 ↪alpha, class_prop_list)
    cm = confusion_matrix(test_y, preds)
    accuracy = np.sum(preds == test_y)/test_y.shape[0]
    mc_rates.append(1-accuracy)
plt.plot(alphas,mc_rates)
plt.title("Misclassification Rate vs. alpha")
plt.ylabel("Misclassfication Rate")
plt.xlabel("Alpha")
plt.show()
print(f'Minimum Misclassification Rate was {mc_rates[np.argmin(mc_rates)]:0.4f}␣
 ↪with alpha={alphas[np.argmin(mc_rates)]}')
```




```
Minimum Misclassification Rate was 0.0952 with alpha=0.1
```