HW1

Jinhong Du - 12243476

April 14, 2020

Contents

Problem A

1. Perform the following simulation study. Simulate 1000 tusks (values of x) from each of the models M_S and M_F. For each simulated tusk compute the LR for M_S vs M_F, so you have computed 2000 LRs. Now consider using the LR to classify each tusk as being from a savanna or a forest elephant. Recall that large values for LR indicate support for M_S, so a natural classification rule is "classify as savanna if $LR > c$, otherwise classify as forest" for some threshold c. Plot the misclassification rate (= number of tusks wrongly classified/2000) for this rule, as c ranges from 0.01 to 100. What value of c minimizes the misclassification rate? [Hint: the plot will look best if you do things on the log scale, so you could let $\log_{10}(c)$ vary from -2 to 2 using an equally spaced grid, and plot the misclassification rate on the y axis against $\log_{10}(c)$ on the x axis.]
2. Repeat the above simulation study using 100 tusks from M_S and 1900 tusks from M_F. What value of c minimizes the misclassification rate? Comment.

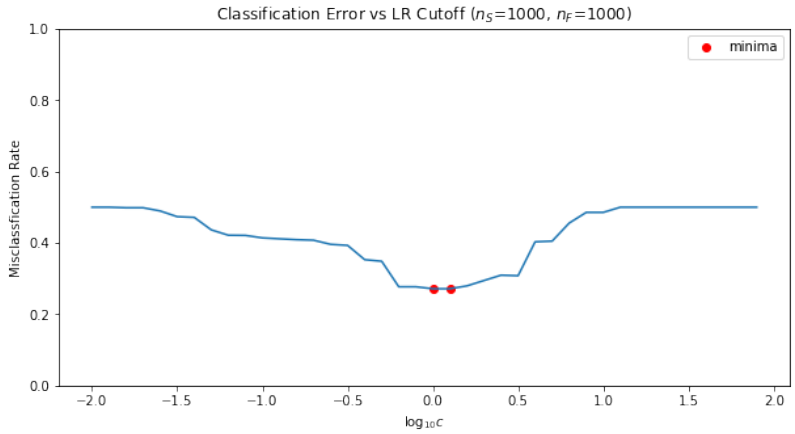
First we define the following functions:

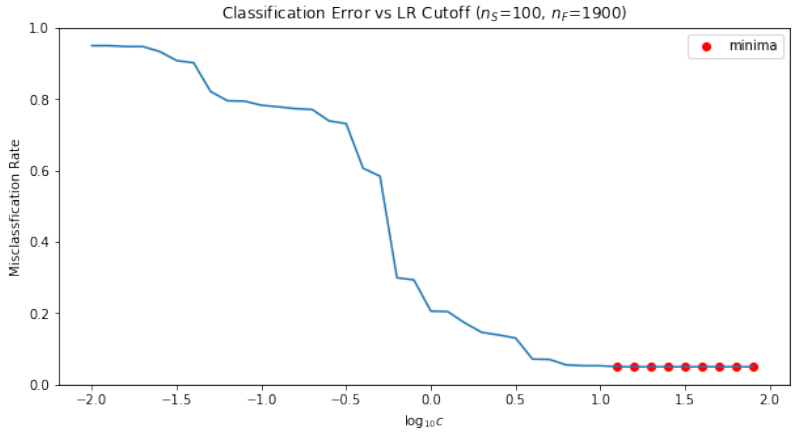
[1]:
```
# Import Packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
def generate_x(f, n, seed):
    '''
    Generate random samples from the given model.

    Params:
        f    - list of m bernoulli proportions
        n    - number of samples to be generated
        seed - random seed
    Return:
        x    - n-by-m array of samples
    '''
    np.random.seed(seed)
    x = np.random.binomial(1, f, size=(n,len(f)))
    return x
def likelihood_ratio(fS, fF, x):
    '''
    Compute the LR of two models given data x.

    Params:
        fS - list of probability for the 1st model
        fF - list of probability for the 2nd model
        x  - array of sample points
    Return:
        LR - n-by-1 array of likelihood ratios of samples
    '''
    L = lambda f,x:np.prod(f**x * (1-f)**(1-x), axis=1)
    LR = L(fS,x)/L(fF,x)
    return LR
```

[2]:
```
def misclassification_rate(y, y_hat):
    '''
    Compute the misclassification rate.

    Params:
        y     - [n, ] array of true binary labels
        y_hat - [c,n] array of predictions
    Return:
        mr    - [c, ] array of misclassification rates
    '''
    mr = np.mean(np.abs(y-y_hat), axis=1)
    return mr

def simulation_study(nS, nF, fS, fF, lc, seed):
    '''
    Perform simulation.

    Params:
        nS   - number of samples generated from the 1st model
        nF   - number of samples generated from the 2nd model
        fS   - array of probability for the 1st model
        fF   - array of probability for the 2nd model
        lc   - array of log(c) values of LR cutoff to use
        seed - random seed
    '''
    x = np.r_[generate_x(fS, nS, seed), generate_x(fF, nF, seed)]
    LR = likelihood_ratio(fS, fF, x)

    y = np.zeros(nS+nF)
    y[nS:] = 1
    lc = lc.reshape(-1,1)
    y_hat = (np.tile(LR,(len(lc),1))<=10**lc).astype(int)

    mr = misclassification_rate(y, y_hat)

    plt.figure(figsize=(10,5))
    plt.plot(lc, mr)
    plt.scatter(lc[mr==mr.min()], mr[mr==mr.min()], c='r', label='minima')
    plt.legend()
    plt.xlabel('$\log_{10}c$')
    plt.ylabel('Misclassfication Rate')
    plt.title('Classification Error vs LR Cutoff ($n_S$=%d, $n_F$=%d)'%(nS,nF))
    plt.ylim([0,1])
    plt.show()
```

Now we can do the two experiments and look at the results.

[3]:
```
fS = np.array([0.40, 0.12, 0.21, 0.12, 0.02, 0.32])
fF = np.array([0.8, 0.2, 0.11, 0.17, 0.23, 0.25])
lc = np.arange(-2, 2, 0.1)

simulation_study(1000, 1000, fS, fF, lc, 1)
simulation_study(100 , 1900, fS, fF, lc, 1)
```





For the first experiment, when $\log_{10} c$ is near 0, i.e., c is near 1, the misclassfication rate is smallest. This is because the numbers of samples generated from M_S and M_F are the same. So the data cannot tell us which model is apprently better than the other.

For the second experiment, the misclassification rate is smallest when c is larger than about 10. This is because that as most samples are generated from M_F, we will need strong evidence (large value of c) to conclude that a sample in from M_S rather than M_F.

Problem B

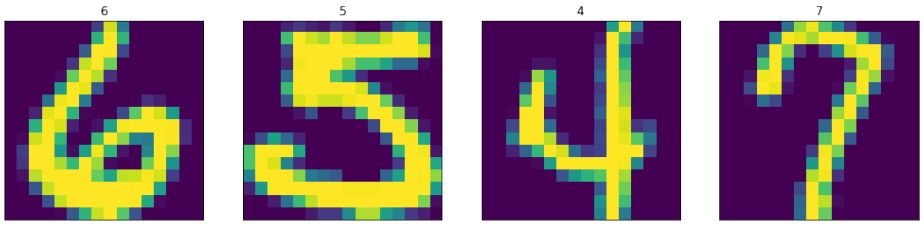
Consider the zipcode data from ESL (https://web.stanford.edu/~{}hastie/ElemStatLearn/index.html).

- Download the data and try plotting a few examples of the training data as 16 x 16 images to see if you can see the digits visually as expected.
- Consider the problem of trying to distinguish the digit 2 from the digit 3.
  - Use the training data to learn classifiers, using a) logistic regression and b) k-nn, with $k = 1, 3, 5, 7, 15$. This gives 6 classifiers in total.
  - Apply these classifiers to the test data, and plot the misclassification rates for both training data and test data. (Plot the results for k-nn with k on x axis, and misclassification rate on y axis, with two different colors for test and training sets. Then put appropriately colored horizontal lines on the same plot – one for test and one for train – indicating the results for logistic regression.)
  - (Note: This is basically Exercise 2.8 from ESL, except replacing linear regression with logistic regression.)
- Repeat the k-nn training as above, but using CV *on the training set* to tune k. That is, act like you do not have access to the test data and have to decide what k to use. How does it do?
- Suppose now that for some reason it is considered worse to misclassify a 2 as a 3 than vice versa. Specifically, suppose you lose 5 points every time you misclassify a 2 as a 3, but 1 point every time you misclassify a 3 as a 2.
  - Modify your logistic regression classifier to take account of this new loss function. Compute the new loss on the test set for both the modified classifier and the original logistic classifier.
  - As far as you can, repeat this for the k-nn classifiers (ie modify them for the new loss function and compare the loss for modified vs original classifiers). Discuss any challenges you face here.

1. Plotting Samples. We have totally 7291 training samples and 2007 testing samples. The figures are as follows.

[4]:
```
train = np.loadtxt('zip.train')
test = np.loadtxt('zip.test')
print(train.shape, test.shape)

fig, axes = plt.subplots(1,4, figsize=(16,4))
for i in range(4):
    axes[i].imshow(train[i,1:].reshape(16,16))
    axes[i].set_title('%d'%train[i,0])
    axes[i].get_xaxis().set_visible(False)
    axes[i].get_yaxis().set_visible(False)
```

```
(7291, 257) (2007, 257)
```



[3]:
```
# Prepare data: Select samples with labels 2 or 3
train = train[(train[:,0]==2)|(train[:,0]==3)]
test = test[(test[:,0]==2)|(test[:,0]==3)]
Y_train, X_train = train[:,0], train[:,1:]
Y_test, X_test = test[:,0], test[:,1:]
print(len(Y_train), len(Y_test))
```

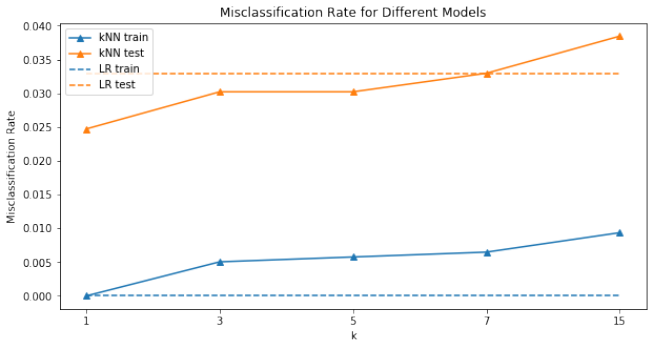
```
1389 364
```

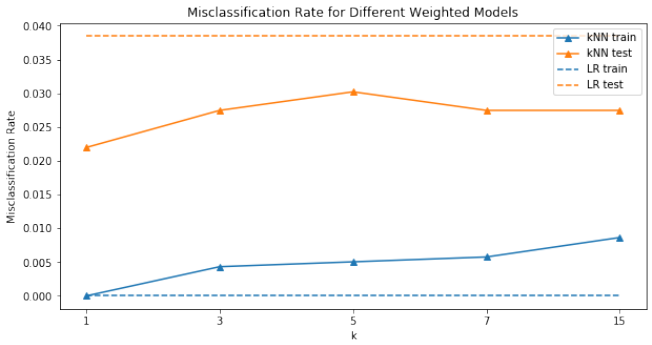
For labels 2 and 3, we have 1389 training samples and 364 testing samples.

2. Train and Test for Weighted and Unweighted Models. We first defind the following two functions to train and test, and plot the results. For the weighted Logistic Regression model, we can just specify the **class weight**, i.e., the weight for different labels. While for the weighted kNN model, we need to use a weighted metric and pass a weight vector for all samples (**sample weight**) when training.

[5]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier

def train_and_test(weight, X_train, Y_train, X_test, Y_test):
    '''
    Plot misclassification rate in the training and testing set.
    Params:
        weight          - boolean if weighted or not
        X_train, X_test - features of training and testing set
        Y_train, Y_test - labels of training and testing set
    Returns:
        ms              - [6,2] array of misclassication rate/loss
        Y_hat           - prediction on testing set
    '''
    k_list = [1,3,5,7,15]
    if weight:
        w = np.ones(len(Y_train))
        w[Y_train==2] = 5
        classifiers = [LogisticRegression(random_state=0,
            class_weight={2:5,3:1}, solver='newton-cg')] + \
            [KNeighborsClassifier(n_neighbors=k, algorithm='brute',
            metric='wminkowski', p=1, metric_params={'w': w}) for k in k_list]
    else:
        classifiers = [LogisticRegression(random_state=0, solver='newton-cg')] + \
              [KNeighborsClassifier(n_neighbors=k, algorithm='brute') for k in k_list]
    ms = np.zeros((6,2))
    Y_hat = np.zeros((6, len(Y_test)))
    for i,clf in enumerate(classifiers):
        clf.fit(X_train, Y_train)
        ms[i,:] = [clf.score(X_train, Y_train), clf.score(X_test, Y_test)]
        Y_hat[i,:] = clf.predict(X_test)
    ms = 1 - ms
    return ms, Y_hat
```

[6]:
```
def plot_train_test_results(ms, ylabel='Misclassification Rate',
                            title='Misclassification Rate for Different Models',
                            item_names=['train','test']):
    '''
    Plot misclassification rate in the training and testing set.
    Params:
        ms        - [6,2] array of misclassication rate/loss
        ylabel    - str of ylabel
        title     - str of title
        item_name - list of str of 2 items (train/test or unweighted/weighted)
    '''
    plt.figure(figsize=(10,5))
    plt.plot(ms[1:, 0], '-^', label='kNN '+item_names[0])
    plt.plot(ms[1:, 1], '-^', label='kNN '+item_names[1])
    plt.hlines(ms[0,0], 0, 4, linestyles='dashed', colors='#1f77b4',
               label='LR '+item_names[0])
    plt.hlines(ms[0,1], 0, 4, linestyles='dashed', colors='#ff7f0e',
               label='LR '+item_names[1])
    plt.xlabel('k')
    plt.xticks(np.arange(5), [1,3,5,7,15])
    plt.ylabel(ylabel)
    plt.title(title)
    plt.legend()
    plt.show()

ms, Y_hat = train_and_test(False, X_train, Y_train, X_test, Y_test)
plot_train_test_results(ms)
ms, Y_hat_w = train_and_test(True, X_train, Y_train, X_test, Y_test)
plot_train_test_results(ms, title='Misclassification Rate for Different Weighted␣
↪Models')
```





The above two plots show the results for unweighted and weighted models. On both the training and testing set, (weighted and unweighted) kNN with $k = 1$ achieve smallest misclassfication rate. Specifically, the 1NN and Logistic Regression model perfectly separate the samples in the training set. For kNN, as k increases, the training error decrease sicne smaller k increases complexity of the model, which can be seen from the decision boundaries. And the testing errors also decrease as k increases. This may be because that the testing set is similar to the training so the 1NN can also perform well in the testing set.

3. Cross-Validation for Weighted and Unweighted Models. As the distributions of the numbers of two classes in training and testing set are similar, we can use K-fold cross validation. I choose $K = 10$. The procedure is as follows.

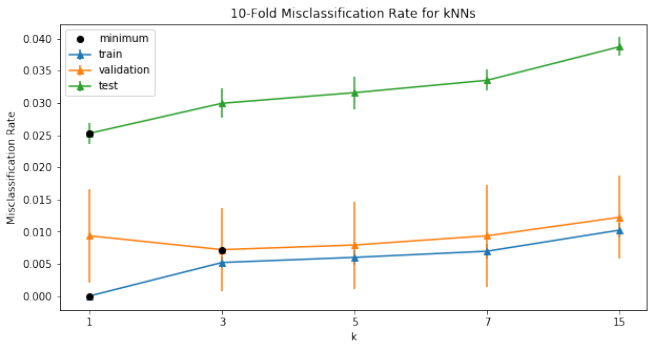
1. We randomly split the training dataset into 10 parts with (near) equal sizes.
2. For kNN models, we iterate over different values of k. For a given k, we treat the each part of the data as the validation set, use other parts as training set to train a new model, and evaluate the model on the corresponding validation set.
3. Finally we combine the results on 10 folds and choose a k that achieves smallest average error.

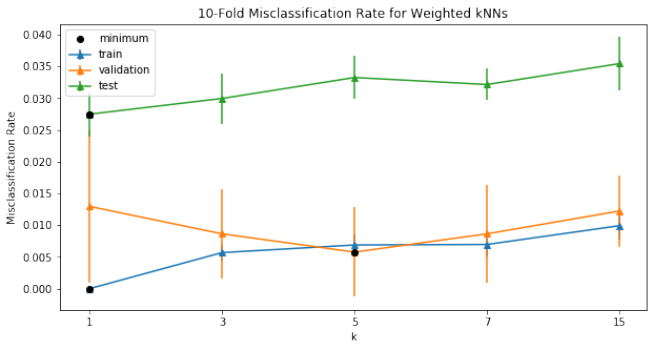
[7]:
```
print(np.sum(Y_train==2), np.sum(Y_train==3))
print(np.sum(Y_test==2), np.sum(Y_test==3))
```

```
731 658
198 166
```

[8]:
```
from sklearn.model_selection import KFold
def cv(weight, X_train, Y_train, X_test, Y_test):
    '''
    Plot misclassification rate in the training and testing set.
    Params:
        weight          - boolean if weighted or not
        X_train, X_test - features of training and testing set
        Y_train, Y_test - labels of training and testing set
    Returns:
        ms              - [5,2] array of misclassication rate/loss
        Y_hat           - prediction on testing set
    '''
    k_list = [1,3,5,7,15]
    kf = KFold(n_splits=10, shuffle=True, random_state=0)
    Y_hat = np.zeros((10,5,len(Y_test)))
    ms = np.zeros((10,5,3))
    for fold, (train_index,test_index) in enumerate(kf.split(X_train)):
        _X_train, _X_val = X_train[train_index], X_train[test_index]
        _Y_train, _Y_val = Y_train[train_index], Y_train[test_index]
        for i,k in enumerate(k_list):
            if weight:
                w = np.ones(len(_Y_train))
                w[_Y_train==2] = 5
                clf = KNeighborsClassifier(n_neighbors=k, algorithm='brute',
                                           metric='wminkowski', p=1, metric_params={'w': w})
            else:
                clf = KNeighborsClassifier(n_neighbors=k, algorithm='brute')
            clf.fit(_X_train, _Y_train)
            ms[fold,i,:] = [clf.score(_X_train, _Y_train), clf.score(_X_val, _Y_val),
                            clf.score(X_test, Y_test)]
            Y_hat[fold,i,:] = clf.predict(X_test)
    ms = 1 - ms
    return ms, Y_hat
```

[9]:
```
def plot_cv_results(ms, ylabel='Misclassification Rate', title='10-Fold␣
↪Misclassification Rate for kNNs',
                    item_names=['train','validation','test']):
    '''
    Plot misclassification rate in the training and testing set.
    Params:
        ms        - [10, 5, 2/3] array of misclassication rate/loss
        ylabel    - str of ylabel
        title     - str of title
        item_name - list of str of 2/3 items (train/val/test or unweighted/weighted)
    '''
    plt.figure(figsize=(10,5))
    for i in range(ms.shape[-1]):
        plt.errorbar(np.arange(5),
                     np.mean(ms[:,:,i], axis=0), np.std(ms[:,:,i], axis=0),
                     linestyle='-', marker='^', label=item_names[i])
        arg_min = np.argmin(np.mean(ms[:,:,i], axis=0))
        plt.scatter(arg_min, np.mean(ms[:,arg_min,i]), c='k',
                    zorder=6)
    plt.scatter(arg_min, np.mean(ms[:,arg_min,i]), c='k', label='minimum')
    plt.xlabel('k')
    plt.xticks(np.arange(5), [1,3,5,7,15])
    plt.ylabel(ylabel)
    plt.title(title)
    plt.legend()
    plt.show()

ms, Y_hat_cv = cv(False, X_train, Y_train, X_test, Y_test)
plot_cv_results(ms)
ms, Y_hat_cv_w = cv(True, X_train, Y_train, X_test, Y_test)
plot_cv_results(ms, title='10-Fold Misclassification Rate for Weighted kNNs')
```

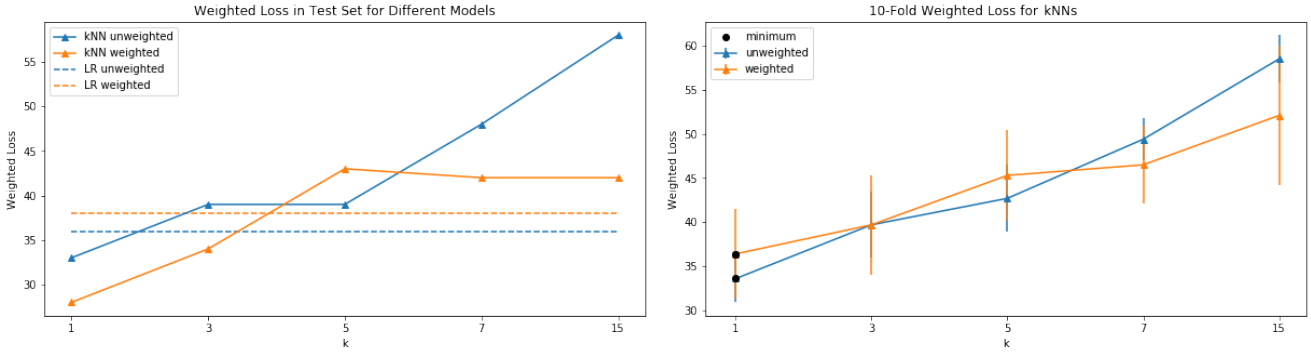




As we can see, the best value of k selected by 10-fold cross validation is 3 for unweighted kNN and 5 for weighted kNN. The testing errors for cross-validation models are similar to the testing errors in part 2. For unweighted models, although in the testing set 1NN performs slightly better than 3NN, in general we would prefer to choose a simpler model, i.e., 3NN, to avoid overfitting.

4. Weighted Loss in Testing Set. Now we are able to combine results from the previous part, to compare the weighted loss in the testing set for different models.

[10]:
```
losses = np.zeros((6,2))
losses[:,0] = np.sum((Y_test==3)&(Y_hat==2), axis=1) + \
              5*np.sum((Y_test==2)&(Y_hat==3), axis=1)
losses[:,1] = np.sum((Y_test==3)&(Y_hat_w==2), axis=1) + \
              5*np.sum((Y_test==2)&(Y_hat_w==3), axis=1)
plot_train_test_results(losses, 'Weighted Loss', 'Weighted Loss in Test Set for␣
 ↪Different Models', ['unweighted', 'weighted'])

losses = np.zeros((10,5,2))
losses[:,:,0] = np.sum((Y_test==3)&(Y_hat_cv==2), axis=-1) + 5*np.
 ↪sum((Y_test==2)&(Y_hat_cv==3), axis=-1)
losses[:,:,1] = np.sum((Y_test==3)&(Y_hat_cv_w==2), axis=-1) + 5*np.
 ↪sum((Y_test==2)&(Y_hat_cv_w==3), axis=-1)
plot_cv_results(losses, 'Weighted Loss', '10-Fold Weighted Loss for kNNs',
    ['unweighted', 'weighted'])
```



For models without cross-validation, as expected, the weighted models perform better than their corresponding unweighted version on the weighted loss on the testing set in general. However, the weighted LR model perform worse than the unweighted LR model. We can look at the predictions of the unweighted and weighted LR models.

[11]:
```
from sklearn.metrics import confusion_matrix
print('TN,  FP,  FN,  TP')
print(confusion_matrix(Y_test, Y_hat[0,:]).ravel())
print(confusion_matrix(Y_test, Y_hat_w[0,:]).ravel())
```

```
TN,  FP,  FN,  TP
[192   6   6 160]
[192   6   8 158]
```

Actually, the training set is perfectly separable for both of these two models. The weighted model would tend to classify a sample as class 2 since class 2 has more weight. As we can see, the weighted LR model misclassify 2 more test samples of label 3, which causes larger loss in the testing set.

For kNN with cross-validation, the weighted losses on the testing set for weighted and unweighted models are similar. For large k, the weighted models achieve lower weighted losses on the testing set.

Problem C

Clone the repository https://github.com/stephens999/stat302/.

- Run the R code in `exercises/seeb/train_test.R` (i.e. https://github.com/stephens999/stat302/blob/master/exercises/seeb/train_test.R) which loads and processes a dataset on 544 fish (salmon) at 12 genetic markers/loci, from (Seeb et., 2007).
- Complete the exercise in the commented text at the end of this file. (note: see https://github.com/stephens999/stat302/tree/master/data/seeb for a description of the data, and particular explanation of the idea that fish are "diploid").

  The idea is that you want to see if you can correctly classify the individuals in the test set based on the information in the training set.

  1. At each locus, use the training set to estimate the allele frequencies (i.e. proportions) in each of the four subpopulations. Assume for the remainder of this exercise that these allele frequencies from the training set are the "true" frequencies in each population.
  2. For each individual in the test data set, compute the posterior probability that it arose from each of the four populations, assuming that all four populations are equally likely a priori. You can assume that the 12 loci contribute independently to the likelihood. That is, the likelihood is defined by multiplying the likelihood across loci.
  3. If you "assign" each individual in the test set to the population that maximizes its posterior probability, what is the error rate? (i.e. how many individuals are misassigned vs correctly assigned?)
  4. Comment on any problems you came across as you did this exercise, and how you solved them. Your answer should include all your R code in a format that can be run to reproduce your results (I recommend using RStudio and the knitr package to produce your report).

[12]:
```
orig_data = read.table("four_salmon_pops.csv", header=TRUE, colClasses="character",␣
 ↪sep=",")
set.seed(100)

#Convert the data at each locus to a factor
#Note that we have to be careful to include all the levels from *both* columns
#for each locus
mylevels= function(locus){levels(factor(c(orig_data[,(1+2*locus)],␣
 ↪orig_data[,(2+2*locus)])))}

#now set up four_salmon_pops
four_salmon_pops = orig_data for(locus in 1:12){
four_salmon_pops[,1+2*locus] = factor(four_salmon_pops[,1+2*locus],␣
 ↪levels=mylevels(locus))
four_salmon_pops[,2+2*locus] = factor(four_salmon_pops[,2+2*locus],␣
 ↪levels=mylevels(locus))
}

#Randomly divide the data into a training set and a test set
```

```
nsamp = nrow(four_salmon_pops)
subset = (rbinom(nsamp,1,0.5)==1) #include each fish in subset with probability 0.5
train = four_salmon_pops[subset,]
test = four_salmon_pops[!subset,]

#this function computes a table of the alleles and their counts at a given locus␣
↪(locus= 1...12)
#in a given data frame (data)
compute_counts = function(data,locus){
return(table(data[,1+2*locus]) + table(data[,2+2*locus]))
}

#Here's an example of how this can be used to compute the allele frequencies
# (both the counts and the proportions) in the training set
trainc = list()
for(i in 1:locus){trainc[[i]] = compute_counts(train,i)}
normalize= function(x){x/sum(x)}
trainf = lapply(trainc, normalize) #compute the proportions from counts
```

Let p_{ijk} denote the proportion of the kth allele in population i at locus j for $i = 1, 2, 3, 4$ (`EelR`, `FeatherHfa`, `FeatherHsp`, `KlamathRfa`), $j = 1, \ldots, 12$ and $k = 1, \ldots, n_j$. As at each locus in each population, an individual will have two alleles, this relationship can be characterized by the multinomial distribution with parameters n and $p_{ij1}, \ldots, p_{ijn_j}$. So the likelihood for sample x in population i is given by

$$L_i(x) = \prod_{j=1}^{12} 2 \left(\prod_{k=1}^{n_j} p_{ijk}^{x_{jk}} \right)$$

where $x_{jk} = 1$ if it has the kth allele at locus j and $x_{jk} = 0$ otherwise.

[13]:
```
compute_p <- function(){ # Use the training set to estimate the allele frequencies
p <- list()
population = unique(train$Population)
for(ip in 1:length(population)){
    p[[ip]] <- list()
    for(i in 1:locus){
        p[[ip]][[i]] <- compute_counts(train[train$Population==population[ip],],i)
    }
    p[[ip]] = lapply(p[[ip]], normalize)
}
return(p)
}
```

Since some values in the testing set are NA. Let's first check if there is any single allele missing.

[14]:
```
for(i in 1:dim(test)[1]){
for(j in 1:12){if(sum(is.na(test[i,(1+2*j):(2+2*j)]))==1){cat(i,j,'\n')}}}
```

As no output displayed, each two alleles measured at the same marker should both be missing if one of them is missing. As we are comparing in the individual level, to see the individual likelihood related to

which population is largest, we can ignore the missing alleles and just compare the likelihood computed with known alleles.

For any given sample x of X, let Z be the random variable of indicator of which population x belongs to. The posterior can be computed by

$$\mathbb{P}(Z=i|X=x)=\frac{\mathbb{P}(X=x|Z=i)\mathbb{P}(Z=i)}{\mathbb{P}(X=x)}.$$

Now since the priors for four populations are the same, i.e. $\mathbb{P}(Z=i)=\frac{1}{4}$ for all i, and $\mathbb{P}(X=x)$ is just a constant, we have

$$\arg\max_i \mathbb{P}(Z=i|X=x)=\arg\max_i \mathbb{P}(X=x|Z=i)=\arg\max_i L_i(x).$$

[15]:
```
p <- compute_p()
L <- matrix(1, dim(test)[1], 4)
posterior <- matrix(0, dim(test)[1], 4)
y_pred <- c()
y_true <- sapply(test$Population, function(x)which(x==population), USE.NAMES = FALSE)
for(i in 1:dim(test)[1]){
    for(ip in 1:4){
        for(j in 1:12){
            if(!is.na(test[i,1+2*j])){
                L[i,ip] <- 2 * L[i,ip] * (
                    p[[ip]][[j]][test[i,1+2*j]] * p[[ip]][[j]][test[i,2+2*j]])
            }
        }
    }
    posterior[i,] <- L[i,]/sum(L[i,])
    y_pred <- c(y_pred, which.max(L[i,]))
}
y_pred <- as.vector(y_pred)

n_correct <- length(which(y_pred-y_true==0))
n_wrong <- length(y_true) - n_correct
cat(n_correct, n_wrong, n_wrong/n_correct)
```

```
155 113 0.7290323
```

The result is bad. This is because some likelihoods of individual in four populations we compute are 0. There may be two reasons. First, some individuals in the testing set have alleles that do not appear in the training set. Second, even some alleles appear in the training set, they may be missing in one specific population.

[16]:
```
cat(sum(rowSums(L)==0))
```

```
86
```

Let's see how many individual have alleles that do not appear in the training set.

[17]:
```
idx_not_in_train = c()
for(i in 1:dim(test)[1]){
    for(j in 1:12){
        if(!is.na(test[i,1+2*j]) & (trainc[[j]][test[i,1+2*j]]==0 |␣
↪trainc[[j]][test[i,2+2*j]]==0)){
            idx_not_in_train <- c(idx_not_in_train, i)
            break
        }
    }
}
cat(length(idx_not_in_train))
```

```
17
```

So there is totally 17 individuals in the testing set that have alleles that do not appear in the training set. Then the other 86-17=69 individuals have all zero likelihoods for four populations due to alleles missing in all populations (the level of alleles missing in different populations can be different).

- For the former case, we can do nothing since we assume that the allele frequencies from the training set are the true frequencies in each population. When new unkown allele appears, it should have zero likelihood. So we cannot compute the posterior probability for them. As for the prediction, we can just randomly sample from the four kinds of populations uniformly.
- For the latter case, we can just add 1 when computing the counts of the training set, to avoid zero frequencies. So some alleles appeared only in some populations can still have very small frequencies for populations in which they dose not appear.

[18]:
```
compute_counts = function(data,locus){
    c <- table(data[,1+2*locus]) + table(data[,2+2*locus])+1
    c[names(trainc[[locus]])[trainc[[locus]]==0]] <-␣
↪c[names(trainc[[locus]])[trainc[[locus]]==0]] - 1
  return(c)
}

set.seed(0)
p <- compute_p()
L <- matrix(1, dim(test)[1], 4)
posterior <- matrix(0, dim(test)[1], 4)
y_pred <- c()
y_true <- sapply(test$Population, function(x)which(x==population), USE.NAMES = FALSE)
for(i in 1:dim(test)[1]){
    if(i %in% idx_not_in_train){
        y_pred <- c(y_pred, sample(1:4, 1))
        next
    }
    for(ip in 1:4){
        for(j in 1:12){
            if(!is.na(test[i,1+2*j])){
                L[i,ip] <- 2 * L[i,ip] * (
                    p[[ip]][[j]][test[i,1+2*j]] * p[[ip]][[j]][test[i,2+2*j]])
```

```
            }
        }
    }
    posterior[i,] <- L[i,]/sum(L[i,])
    y_pred <- c(y_pred, which.max(L[i,]))

}
y_pred <- as.vector(y_pred)

n_correct <- length(which(y_pred-y_true==0))
n_wrong <- length(y_true) - n_correct
cat(n_correct, n_wrong, n_wrong/n_correct)
```

```
202 66 0.3267327
```

As we can see, the error rate is about 0.3267327, which is improved a lot compared to the previous result.