Homework 1

Christopher Marais & Songzi Wu

2022-09-20

Q1

1.1

We imported the data and split the training, validation and test data.

```
data_df <- read.csv("SML.NN.data.csv")
train_df = data_df[data_df$set == 'train',]
valid_df = data_df[data_df$set == 'valid',]
test_df = data_df[data_df$set == 'test',]
```

Thereafter we constructed the function to get the class probability for a single point estimated as the training data points within a radius from the point.

```
# function to get proportion of class 1 in radius to point x (single)
# output is a float
getClass1Prop <- function(x, r, data=train_df) {
  "Import the data this function is
  based on before using it. The data should be named data_df
  and contain columns Y, X1, X2, set all in a dataframe.
  x is a vector of length 2
  r is the selected radius
  "
  # calculate the distance between the point and all the data
  x1 = (data$X1-as.numeric(x[1]))^2
  x2 = (data$X2-as.numeric(x[2]))^2
  data$euc_dist = sqrt(rowSums(data.frame(x1,x2)))
  #select points inside radius
  in_r_df = data[data$euc_dist <= r,]
  if(nrow(in_r_df)==0){
    # return NA if no points within radius
    return(NA)
  } else {
    # calculate proportion of class 1 values in data
    class_1_prop = sum(in_r_df$Y)/nrow(data)
    return(class_1_prop)
  }
}
```

1.2

To create a function that calculates the miss clasification rates we first created a function that creates a classification based on the class 1 probability estimate produced from the previous function. The class 1 probability estimate can be done with the radius function given in 1.1 or with the K nearest neighbor function (KNN) given in 1.6. The method of choice is chosen based on the *func_type* parameter ("radius"=the radius based technique, "knn"= the nearest neighbors technique) . The *rk* parameter is used to allocate either radius size or number of neighbors based on the class 1 probability estimate technique used. The t parameter is used to specify the threshold at which the class probability estimate is seen as a positive class 1 classification. The *data* parameter is used to indicate which data set to use as training data.

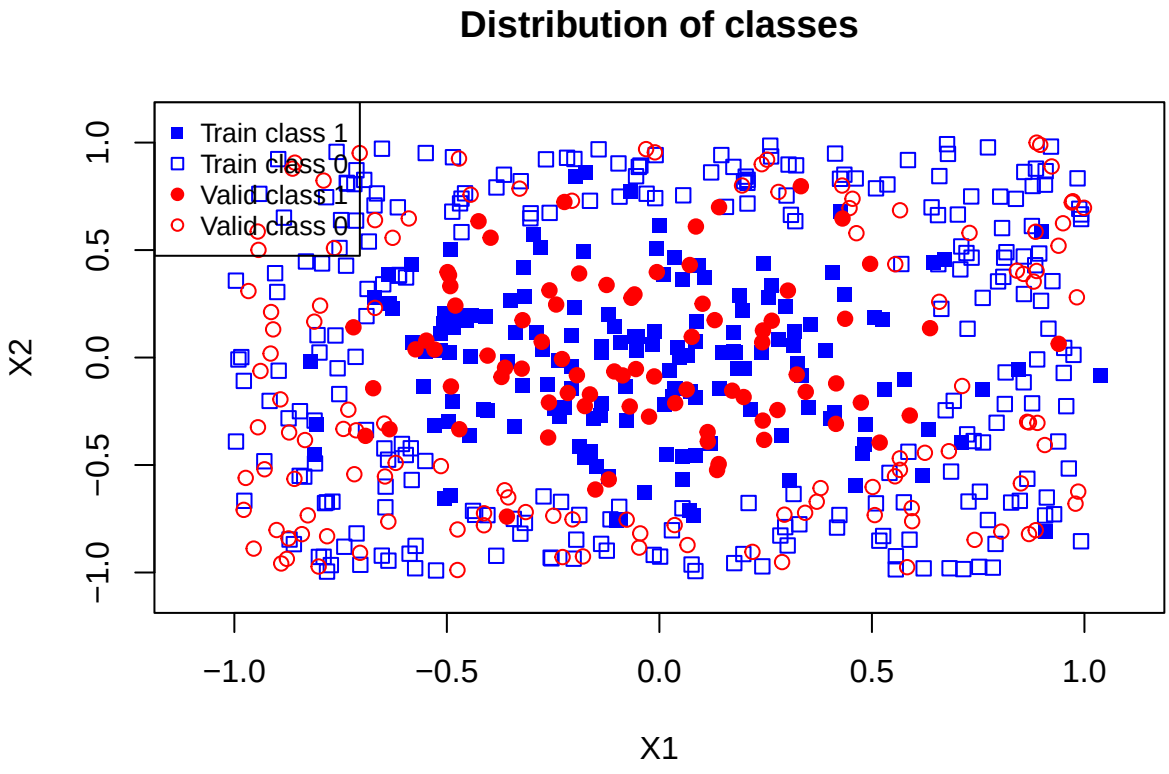
```
# get class 1 prediction for points in x (multiple)
# output is a vector of binary predictions
getClass1Prediction <- function(x, rk, t=0.5, data=train_df, func_type="radius"){
  "t is the threshold and should be between 0 and 1."
  pred_vec = c()
  # loop through all points in x
  for(i in 1:nrow(x)){
    x_i = x[c('X1','X2')][i,] # get coordinates
    if(func_type=="radius"){
      class_1_prop = getClass1Prop(x=x_i, r=rk, data=data)
    }else if(func_type=="knn") {
      class_1_prop = getClass1PropKNN(x=x_i, k=rk, data=data)
    }

    if(is.na(class_1_prop)){
      pred=NA
    }
    else if(class_1_prop >= t){
      pred=1
    }else{
      pred=0
    }
    pred_vec=c(pred_vec, pred)
  }
  return(pred_vec)
}
```

The function to calculate the miss classification was calculated as 1 - the proportion of correct classifications. This means that any classification that resulted in an NA or unknown was labelled as a miss classification.

```
# function to get the miss classification rate
getMissClassRate <- function(true, pred){
  # make dataframe of predictions and true values
  df = data.frame(true, pred)
  # calculate equality of data
  df$equal = (df$true == df$pred)
  mis_class_rate = 1 - sum(df$equal, na.rm = TRUE)/nrow(df)
  return(mis_class_rate)
}
```

1.3

```
# make plot of coordinates density distribution
plot(train_df[train_df$Y==1,]$X1,
     train_df[train_df$Y==1,]$X2,
     main = "Distribution of classes",
     xlab = "X1",
     xlim=c(-1.1, 1.1),
     ylab = "X2",
     ylim=c(-1.1, 1.1),
     pch = 15,
     col = "blue")
points(train_df[train_df$Y==0,]$X1, train_df[train_df$Y==0,]$X2, pch = 0, col = "blue")
points(valid_df[valid_df$Y==1,]$X1, valid_df[valid_df$Y==1,]$X2, pch = 19, col = "red")
points(valid_df[valid_df$Y==0,]$X1, valid_df[valid_df$Y==0,]$X2, pch = 1, col = "red")
legend("topleft",
       legend=c("Train class 1", "Train class 0", "Valid class 1", "Valid class 0"),
       col=c("blue","blue","red","red"),
       pch=c(15,0,19,1),
       cex=0.8)
```

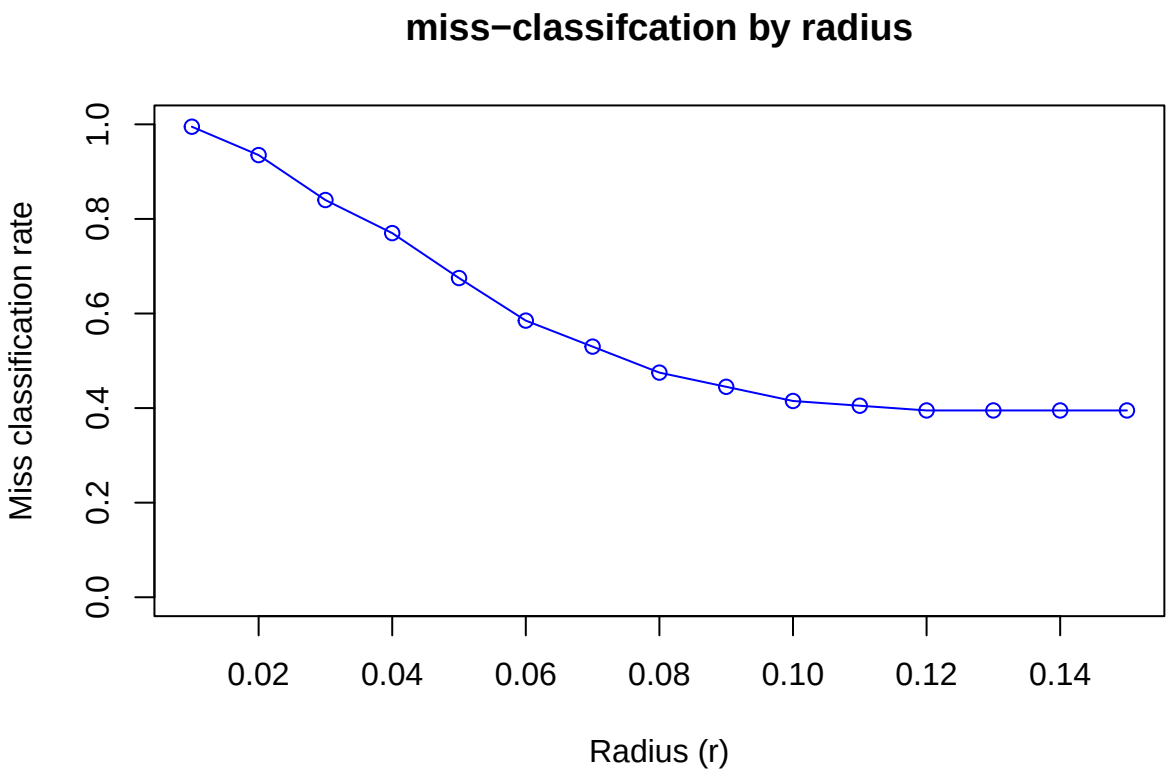


From the plot showing the distribution of training and validation data. we can see that both data sets have points that are uniformly distributed and that the class one points are surrounded by class 0 points for both data sets. The class 1 points seem to be placed within a even distribution around the (0,0) midpoint with a radius of approximately 0.25. A circular radius is an inconvenient shape to classify the edge cases correctly

considering the distribution of the data for class one to also be circular. With a uniform distribution the radius is more likely to include more class 0 points than class 1 points on the border of the two classes. I will refer to this effect as the eclipsing effect. A smaller radius will reduce the eclipsing effect. The radius should also be large enough to reduce then umber of NA classifications. So my guess would be to have the radius be the same as the the distance to the closest neighbor of the most isolated point in the distribution. This is a computationally heavy calculation to perform so I will just guess from a glance at the data that it should be around 0.1.

1.4

```
# select best r for range of r values
# get class predictions
r_grid = seq(0.01, 0.15, 0.01)
class_pred_lst = lapply(r_grid,
                        getClass1Prediction,
                        x=valid_df,
                        t=0.5,
                        data=train_df,
                        func_type="radius")
# get miss classification rate
miss_class_lst = lapply(class_pred_lst, getMissClassRate, true = valid_df$Y)
# plot results
plot(x=r_grid,
      y=miss_class_lst,
      main="miss-classifcation by radius",
     xlab="Radius (r)",
     ylab="Miss classification rate",
     ylim=c(0,1),
     col="blue"
      )
lines(x=r_grid,
      y=miss_class_lst,
      col="blue")
```



From the plot we can see that the lowest r value is **0.12** with a miss classification rate of **0.395** for the validation data. This is not too far off from my guess of the radius of **0.1**.

```
# get miss classification for test data with r*
getMissClassRate(true = test_df$Y,
                 pred = getClass1Prediction(
                   x=valid_df,
                   t=0.5,
                   rk=0.12,
                   data=train_df,
                   func_type="radius"
                 ))
```

```
## [1] 0.33
```

```
# get miss classification for test data with guessed r
getMissClassRate(true = test_df$Y,
                 pred = getClass1Prediction(
                   x=valid_df,
                   t=0.5,
                   rk=0.1,
                   data=train_df,
                   func_type="radius"
                 ))
```

```
## [1] 0.365
```

We can see on the test data that the r* based on the training data and validation data has a lower miss classification rate (**0.33**) than my guessed r value's miss classification rate (**0.365**). However, these two are still quite close.

1.5

We reduced the number of for loops to one. Which can be found in the *getClass1Prediction* function.

1.6

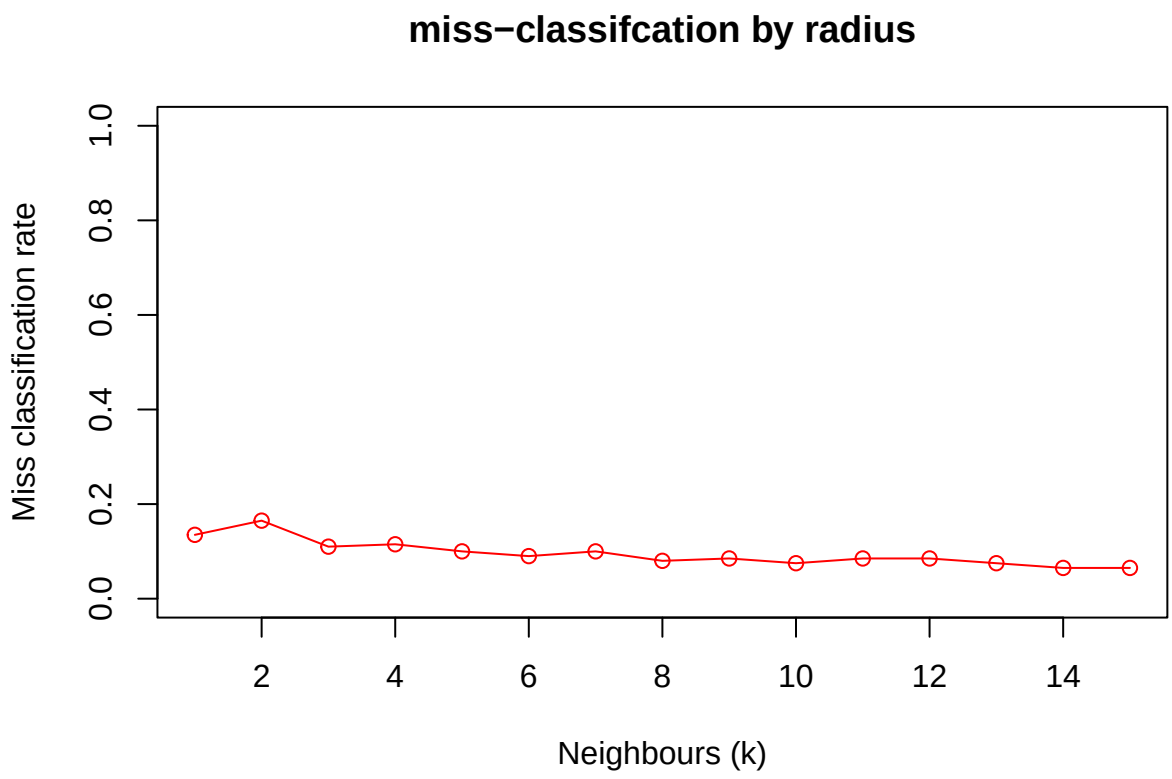
We created a similar function to *getClass1Prop* but this one is based on k nearest neighbors (KNN).

```
# KNN
getClass1PropKNN <- function(x, k, data=train_df) {
  "Import the data this function is
  based on before using it. The data should be named data_df
  and contain columns Y, X1, X2, set all in a dataframe.
  x is a vector of length 2
  k is the number of neigbours
  "
  # calculate the distance between the point and all the data
  x1 = (data$X1-as.numeric(x[1]))^2
  x2 = (data$X2-as.numeric(x[2]))^2
  data$euc_dist = sqrt(rowSums(data.frame(x1,x2)))
  data = data[order(data$euc_dist),]
  class_1_prop = sum(data$Y[1:k])/k
  return(class_1_prop)
}
```

We used this function to similarly get the most optimal value for K in a range of values with the lowest miss classification rate.

```
# get class predictions
k_grid = seq(1, 15, 1)
class_pred_lstKNN = lapply(k_grid,
                        getClass1Prediction,
                        x=valid_df,
                        t=0.5,
                        data=train_df,
                        func_type="knn")
# get miss classification rate
miss_class_lstKNN = lapply(class_pred_lstKNN, getMissClassRate, true = valid_df$Y)
# plot results
plot(x=k_grid,
     y=miss_class_lstKNN,
     main="miss-classifcation by radius",
     xlab="Neighbours (k)",
     ylim = c(0,1),
     ylab="Miss classification rate",
     col="red"
)
lines(x=k_grid,
```

```
      y=miss_class_lstKNN,
      col="red")
```



From the plot we can see that 14 and 15 have the lowest MCR values. We opted to go with 14 as the optimal value as this is less complex (Occam's razor). It is also worth noting that overall the KNN approach has much lower MCR values than any of the radius values tested gave. even the worst value of k has a lower MCR value than the best value of r when tested on the validation data. This is likely because the radius approach creates NA classifications that are registered as miss classifications when we calculate MCR.

```
# get miss classification for test data with best k
getMissClassRate(true = test_df$Y,
                 pred = getClass1Prediction(
                   x=valid_df,
                   t=0.5,
                   rk=14,
                   data=train_df,
                   func_type="knn"
                 ))
```

```
## [1] 0.465
```

However, when we use the optimal value of k on the test data we get a MCR of **0.465** which is higher than what we got for the optimal value of r (r*). This shows how these two methods differently describe the neighborhood they use to classify points.

In [1]:
```
import numpy as np
import pandas as pd
from scipy.stats import norm,t
import matplotlib.pyplot as plt
```

Q1

Question 1 can be found in our Rmarkdown sheet.

Q2

2.1

In [2]:
```
from IPython.display import Image
Image('2_1.jpg')
```

Out[2]:

2.1 $f_Y(y) = \int_{-\infty}^{\infty} f_{XY}(x,y)\,dx$

$= \frac{1}{2\pi\sigma_X\sigma_Y\sqrt{1-\rho^2}} \int_{-\infty}^{\infty} e^{-\frac{1}{2}Q(x,y)}\,dy$

where $Q(x,y) = \frac{(\frac{x-\mu_X}{\sigma_X})^2 - 2\rho(\frac{x-\mu_X}{\sigma_X})(\frac{y-\mu_Y}{\sigma_Y}) + (\frac{y-\mu_Y}{\sigma_Y})^2}{1-\rho^2}$

$= (\frac{x-a}{b})^2 + c$

$a = \mu_X + \rho\frac{\sigma_X}{\sigma_Y}(y-\mu_Y)$

$b = \sigma_X\sqrt{1-\rho^2}$

$c = (\frac{y-\mu_Y}{\sigma_Y})^2$

$\Rightarrow f_Y(y) = \frac{1}{2\pi\sigma_X\sigma_Y\sqrt{1-\rho^2}} \int_{-\infty}^{\infty} e^{-\frac{1}{2}[(\frac{x-a}{b})^2+c]^2}\,dx$

$= \frac{e^{-\frac{c}{2}}}{2\pi\sigma_X\sigma_Y\sqrt{1-\rho^2}} \int_{-\infty}^{\infty} e^{-\frac{1}{2}(\frac{x-a}{b})^2}\,dx$

$= \frac{e^{-\frac{c}{2}}}{\sqrt{2\pi}\sigma_X\sigma_Y\sqrt{1-\rho^2}}\,\sigma_X\sqrt{1-\rho^2}$

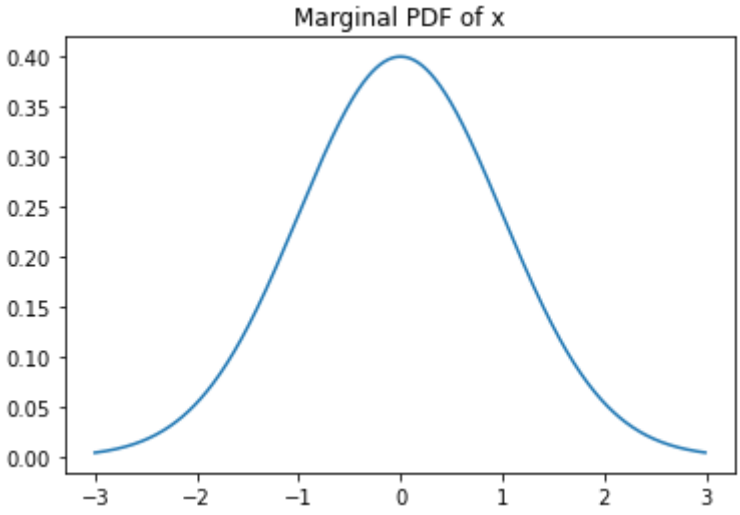
$= \frac{1}{\sqrt{2\pi}\sigma_Y} e^{-\frac{1}{2}(\frac{y-\mu_Y}{\sigma_Y})^2}$

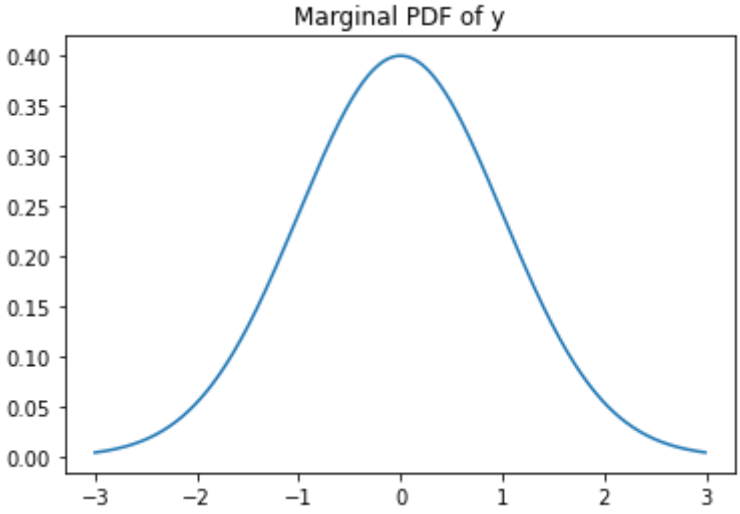
Similarly,

$f_X(x) = \int_{-\infty}^{\infty} f_{XY}(x,y)\,dy$

$= \frac{1}{\sqrt{2\pi}\sigma_X} e^{-\frac{1}{2}(\frac{x-\mu_X}{\sigma_X})^2}$

In [3]:
```
x = np.arange(-3, 3, 0.01)
y = np.arange(-3, 3, 0.01)

# Calculating mean and standard deviation
mu_x,mu_y,sig_x,sig_y = 0,0,1,1

plt.figure()
plt.plot(x, norm.pdf(x, mu_x, sig_x))
plt.title('Marginal PDF of x')
plt.show()
plt.figure()
plt.plot(y, norm.pdf(y, mu_y, sig_y))
plt.title('Marginal PDF of y')
plt.show()
```



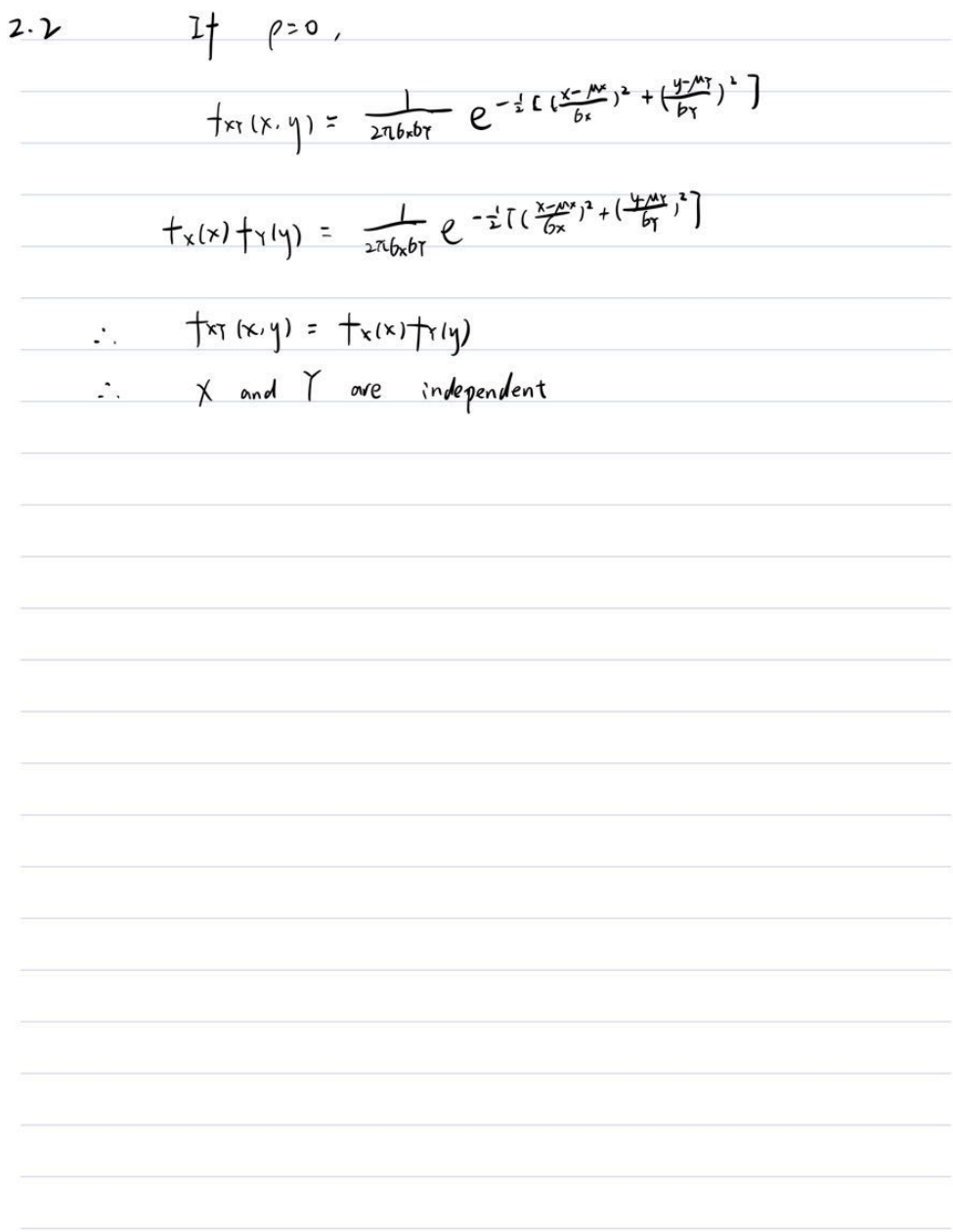


Those marginal pdfs are identical with Normal(0,1) pdf

2.2

In [4]:
```
Image('2_2.jpg')
```

Out[4]:



2.3

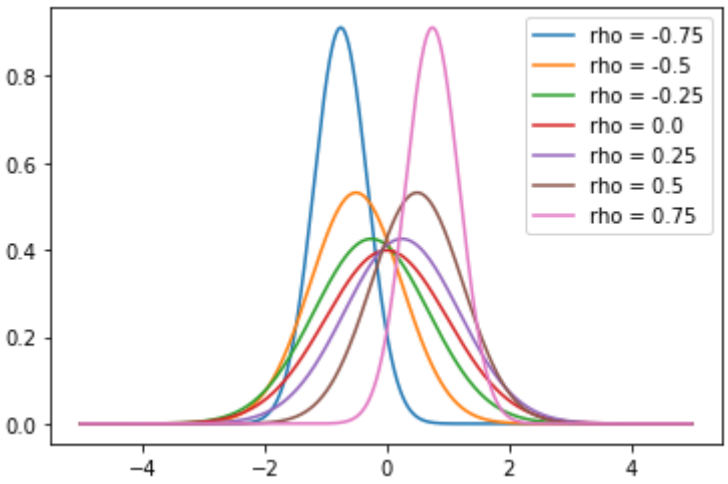
Given $Y=1$:

$E[X|Y=1]=\mu_x+\rho\sigma_x\frac{1-\mu_y}{\sigma_y}\$

$Var[X|Y=1]=(1-\rho^2)\sigma_x^2$

In [5]:
```
rho_grid = np.arange(-0.75,1,0.25)
x = np.arange(-5, 5, 0.01)
mu_x,mu_y,sig_x,sig_y = 0,0,1,1
plt.figure()
```

```
for rho in rho_grid:
    mu = mu_x+rho*sig_x*(1-mu_y)/sig_y
    sig = (1-rho**2)*sig_x**2
    plt.plot(x, norm.pdf(x, mu, sig),label=f'rho = {rho}')
plt.legend()
plt.show()
```



The negative and positive ρ both help to predict X since they indicate a linear relation bwtween X and Y. Larger ρ implies a stronger linearity. It is hard to predict X if X and Y are independent, i.e. $\rho=0$.

Q3

3.1

In [6]:
```
np.random.seed(0)
lam = 10
x = np.random.exponential(1/lam,100)
```

$L = \lambda e^{(-\lambda{x_1})}\lambda e^{(-\lambda{x_1})}...\lambda e^{(-\lambda{x_100})} = \lambda^{100} e^{(-\lambda(x_1+x_2+...+x_{100}))}$

$L'=\ln{L}=100\ln{\lambda}-\lambda\sum_{i=0}^{100}x_i$

$\frac{\partial L'}{partial \lambda} = \frac{100}{\lambda}-\sum_{i=0}^{100}x_i = 0$

$\lambda = \frac{100}{\sum_{i=0}^{100}x_i}$

In [7]:
```
lam = 100/x.sum()
print(f'extimated lambda is {lam}')
```

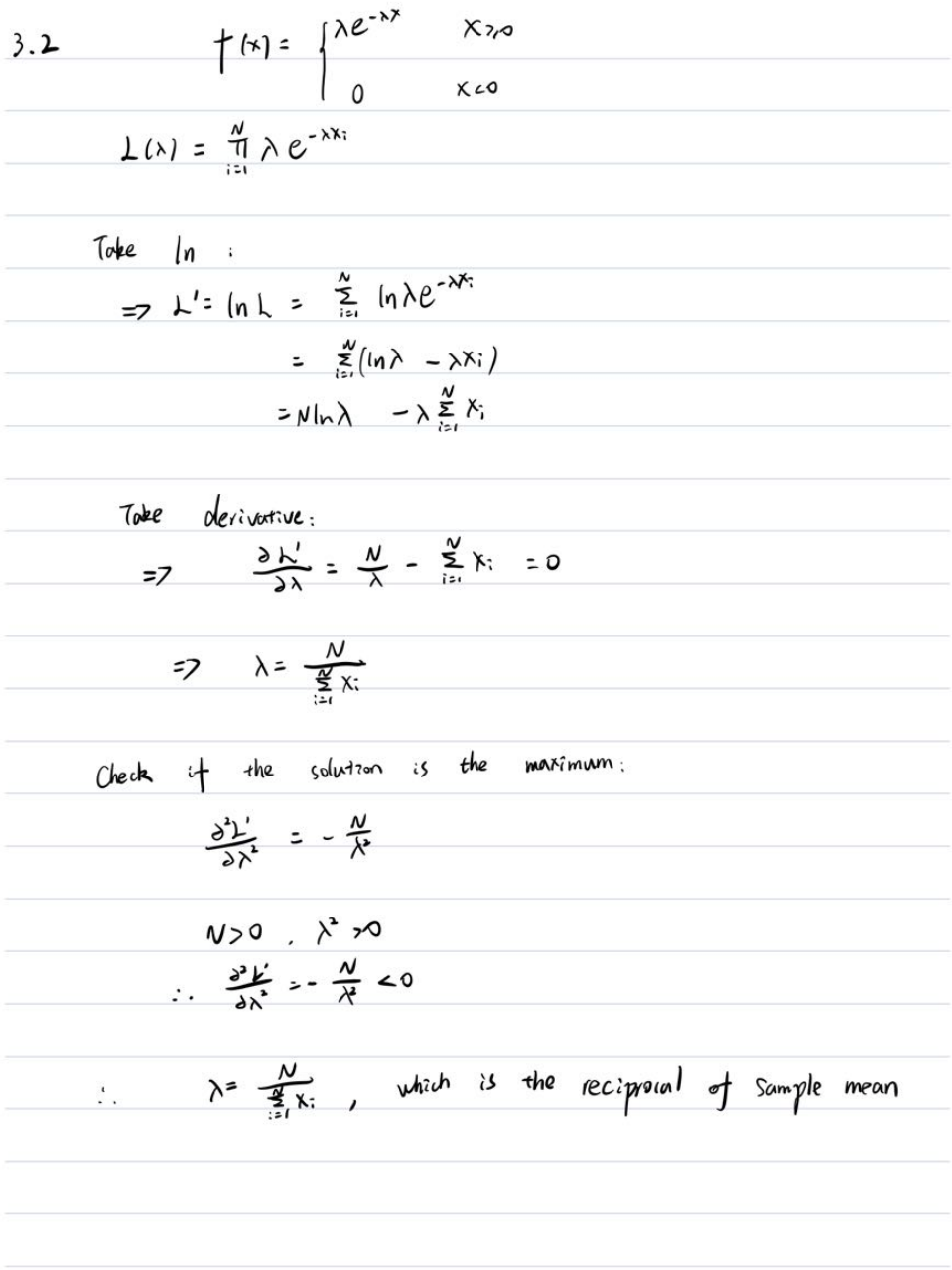
```
extimated lambda is 10.885557365453474
```

The estimated value of λ (10.89) is close to true λ (10)

3.2

In [8]:
```
Image('3_2.jpg')
```

Out[8]:



Q4

Case 1

In [9]:

```
seed_grid = np.arange(1,1001)
coverage = 0

for i in seed_grid:
    np.random.seed(i)
    x = np.random.normal(0,1,4)
    mu = x.mean()
```

```
    std = x.std()
    up = mu+1.6449*std/np.sqrt(4)
    lower = mu-1.6449*std/np.sqrt(4)
    if up>=0 and lower<=0:
        coverage += 1
    else:
        pass
ef = coverage/len(seed_grid)
print(f'The empirical frequency of coverage is {ef}')
```

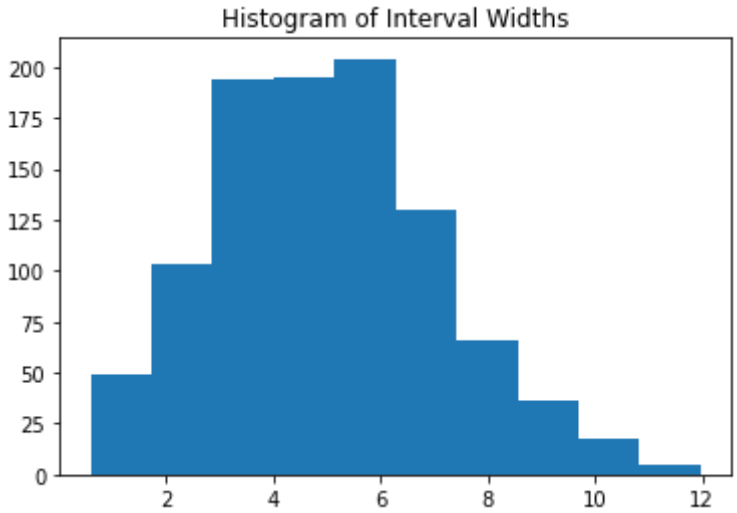
```
The empirical frequency of coverage is 0.749
```

Case 2

In [10]:
```
seed_grid = np.arange(1,1001)
coverage = 0
width = []

for i in seed_grid:
    np.random.seed(i)
    x = np.random.normal(0,1,4)
    mu = x.mean()
    std = x.std()
    interval = t.interval(alpha=0.95, df=3, loc=mu, scale=std)
    if interval[1]>=0 and interval[0]<=0:
        coverage += 1
    else:
        pass
    width.append(interval[1]-interval[0])
ef = coverage/len(seed_grid)
print(f'The empirical frequency of coverage is {ef}')
```

```
C:\Users\GCM\AppData\Local\Temp\ipykernel_23480\3993943170.py:10: DeprecationWarni
ng: Use of keyword argument `alpha` for method `interval` is deprecated. Use first
positional argument or keyword argument `confidence` instead.
  interval = t.interval(alpha=0.95, df=3, loc=mu, scale=std)
The empirical frequency of coverage is 0.988
```

In [11]:
```
plt.figure()
plt.hist(width)
plt.title('Histogram of Interval Widths')
plt.show()
```

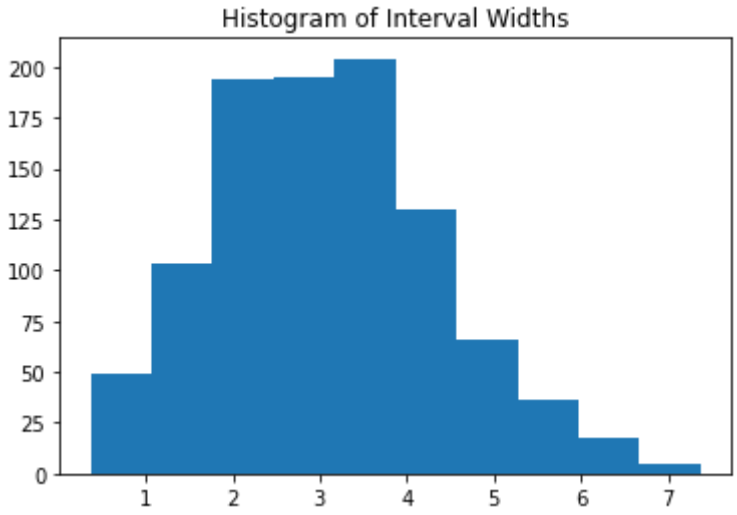


Case 3

In [12]:
```
seed_grid = np.arange(1,1001)
coverage = 0
width = []

for i in seed_grid:
    np.random.seed(i)
    x = np.random.normal(0,1,4)
    mu = x.mean()
    std = x.std()
    interval = norm.interval(alpha=0.95, loc=mu, scale=std)
    if interval[1]>=0 and interval[0]<=0:
        coverage += 1
    else:
        pass
    width.append(interval[1]-interval[0])
ef = coverage/len(seed_grid)
print(f'The empirical frequency of coverage is {ef}')
```

```
C:\Users\GCM\AppData\Local\Temp\ipykernel_23480\1891025027.py:10: DeprecationWarni
ng: Use of keyword argument `alpha` for method `interval` is deprecated. Use first
positional argument or keyword argument `confidence` instead.
  interval = norm.interval(alpha=0.95, loc=mu, scale=std)
The empirical frequency of coverage is 0.957
```

In [13]:
```
plt.figure()
plt.hist(width)
plt.title('Histogram of Interval Widths')
plt.show()
```



Case 2 has the highest empirical frequency of coverage when exact small sample CI is computed using t-distribution. That makes sense since the sample is very small (N=4). When using approximate large-sample CI in case 3, the empirical frequency decreases.

In [ ]: