MAT3373_A6

Mohammad Alqahtani, Yasmine Zoubdi, Sebastian Doka

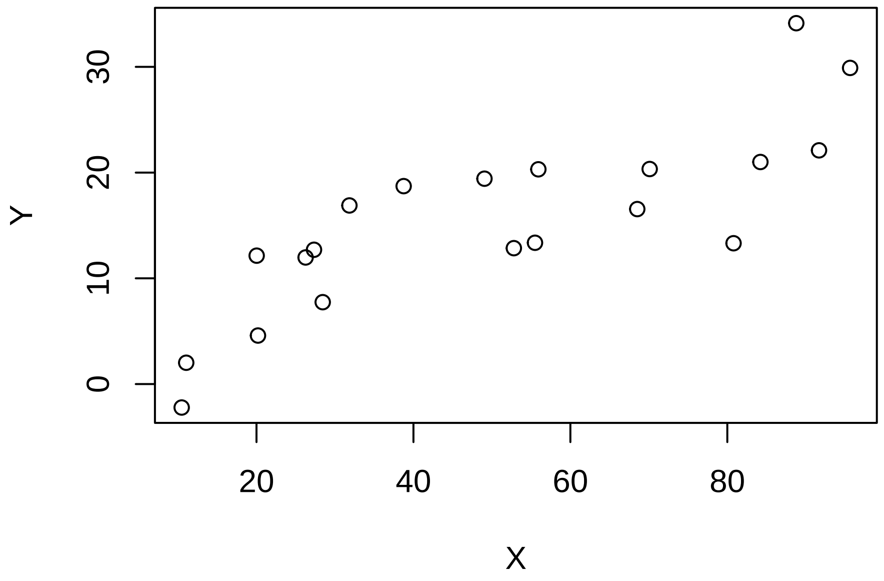
2023-04-11

Total Marks: 24. (5 questions on 4 pages)

Question 1 [8 marks]

The following code generates a synthetic training dataset (the same one as in the Midterm Question 3) using the relationship $Y = 0.3 * X + \varepsilon$, where ε is normally distributed around 0.

```r
set.seed(5)
N=20
X = runif(n=N, min=0, max=100)
Y = 0.3 * X + rnorm(n=N, 0, 5)
plot(X, Y)
```



A) Fit two models to this training data using Ordinary Least Squares (OLS) regression: a linear model and a cubic model. For each model, calculate the Mean Squared Error (MSE) on the training set.

A) The linear model

```r
model1 = lm(Y ~ X)
MSE = mean((Y - predict(model1, newdata = data.frame(X,Y)))^2)
MSE
```

```
## [1] 23.49516
```

The cubic model

```r
model2 = lm(Y ~ poly(X, 3, raw = TRUE))
MSE = mean((Y - predict(model2, newdata = data.frame(X,Y)))^2)
MSE
```

```
## [1] 14.7889
```

B) Generate a second dataset of 1000 points was synthesized using the same process but with `set.seed(1)`. Calculate the Test MSE on this dataset for each of the models from part (A).

```r
set.seed(1)
N=1000
X = runif(n=N, min=0, max=100)
Y = 0.3 * X + rnorm(n=N, 0, 5)
```

For the linear model:

```r
model1 = lm(Y ~ X)
MSE = mean((Y - predict(model1, newdata = data.frame(X,Y)))^2)
MSE
```

```
## [1] 26.67303
```

For the cubic model:

```r
model2 = lm(Y ~ poly(X, 3, raw = TRUE))
MSE = mean((Y - predict(model2, newdata = data.frame(X,Y)))^2)
MSE
```

```
## [1] 26.5958
```

C) Now for a theoretical question: suppose another cubic model is fitted to this training data using ridge regression, i.e. Ordinary Least Squares (OLS) regression with L2 regularization.

- State the model of the data.
  The model for the data is $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \epsilon$.
- State the objective function that is minimized during the fitting of this model to the data.
  $\Sigma_{i=1}^{N}(Y_i - \beta_0 - \beta_1 X_i - \beta_2 X_i^2 - \beta_3 X_i^3)^2 + \lambda \Sigma_{j=1}^{3} \beta_j^2$
- Assuming the regularization hyperparameter is large enough to make the fitted model visibly different than the original (unregularized) cubic model, does the ridge regression model have a lower or higher Training MSE than the original cubic model? What about Test MSE?

In the case of Ridge regression, the L2 penalty term in the objective function shrinks the estimated coefficients towards zero. That can increase the bias of the model. Since the unregularized cubic model doesn't have any penalty term in the objective function, it could have lower bias on the training set, resulting in a lower training MSE.

The Ridge regression model has lower variance than the unregularized model, that can help perform better on the test set by reducing overfitting, resulting in a lower test MSE.

Question 2 [4 marks]

A)

$$p(Y=1|X=x) = \frac{1}{1+e^{-(\beta_0+\beta_1*X_1+\beta_2*X_2+\cdots+\beta_p*X_p)}}$$

B) We can use the relationship between probability and log odds in logistic regression. Given a log odds of 2 the logit function is defined as:

$$2 = \log\left(\frac{p}{1-p}\right)$$

We solve for `p`:

$$e^2 = \frac{p}{1-p}$$

$$p = e^2 \times (1-p)$$

$$p = \frac{e^2}{1+e^2}$$

Now, we can calculate `p`:

```r
p <- exp(2) / (1 + exp(2))
p
```

```
## [1] 0.8807971
```

C) Given a set of data points $\{D = (x_i, y_i)\}$, where $i = 1, 2, \ldots, n$, and y_i is the true binary outcome (0 or 1) corresponding to the predictor variables x_i, the loss function $L(\boldsymbol{\beta})$ for logistic regression is defined as:

$$L(\boldsymbol{\beta}) = -\sum_{i=1}^{n} [y_i \log(p_i) + (1-y_i)\log(1-p_i)]$$

where p_i is the predicted probability.

D) In binary logistic regression, the coefficients in the model are fitted using maximum likelihood estimation. The goal is to find the coefficients $(\beta_0, \beta_1, \ldots, \beta_p)$ that maximize the likelihood of observing the data given the model. This is equivalent to minimizing the negative log-likelihood loss function $L(\boldsymbol{\beta})$ defined in C.

To find the optimal coefficients, we need to differentiate the negative log-likelihood loss function with respect to each of the coefficients $(\beta_0, \beta_1, \ldots, \beta_p)$. This is where the chain rule comes into play. More specifically, we differentiate the loss function with respect to the coefficients as follows.

For each coefficient β_j for $j = 0, 1, \ldots, p$, we need to compute the derivative of the loss function $L(\boldsymbol{\beta})$ with respect to β_j:

$$\frac{dL(\boldsymbol{\beta})}{d\beta_j}$$

Once we compute the derivatives, we will have a system of $p + 1$ nonlinear equations that describe the gradient of the negative log-likelihood function. To find the optimal coefficients, we need to use iterative optimization algorithms, such as gradient descent, which is studied in Lecture 7. The algorithm computes the coefficients iteratively until the loss function is minimized (possibly locally).

Question 3 [4 marks]

A) If the true distributions of the features are significantly different between the two classes, using the same variances for both classes may lead to a decrease in the model's performance.

Despite the low error rate of 10%, yes it can make sense to try Gaussian Naive Bayes with the same variances for both classes, using the same variances for both classes assumes that the distribution of the features in both classes is similar. If the distributions of the features are indeed similar for both classes, this simplified model might perform better or similarly to the model with different variances for each class.

Furthermore, using the same variances for both classes could lead to a more interpretable and simpler model which is desirable.

B) Acknoldgment: Stackoverflow, Wikipedia, Quora, ChatGPt and more.

Quadratic Discriminant Analysis (QDA) is a classification technique that assumes each class has its own covariance matrix. Unlike Linear Discriminant Analysis (LDA), which assumes that the covariance matrices for all classes are the same, QDA models each class with a separate covariance matrix.

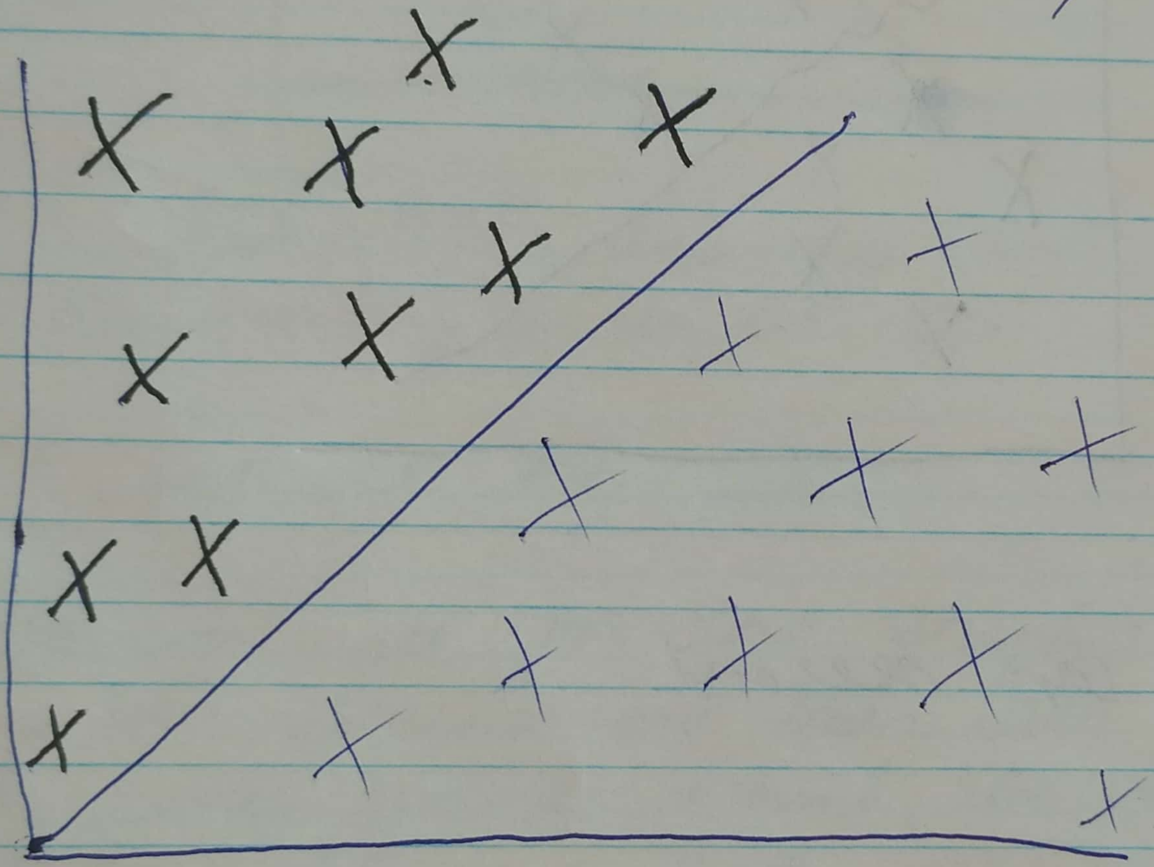
The problem with using QDA, especially for a large datasets like the one mentioned (1000 images, each 200 $\times$ 200 pixels) is overfitting. Since QDA has more parameters to estimate (a separate covariance matrix for each class), it is more prone to overfitting compared to LDA.

Each covariance matrix has $200 \times 200 = 40{,}000$ parameters, and since we have a binary classification problem, we will have to estimate 80,000 parameters for both covariance matrices.

Possible solutions to this problem can include regularization. To address the overfitting issue, we can apply regularization techniques to the covariance matrices, such as shrinkage, which essentially shrinks the estimated covariance matrix. This reduces the model complexity and can improve generalization to test data.

Another way is dimensionality reduction. To reduce the number of parameters to be fitted and the risk of overfitting, we can apply dimensionality reduction techniques, such as Principal Component Analysis (PCA), as seen in lecture 20, or Linear Discriminant Analysis (LDA), lecture 12, before fitting the QDA model. This reduces the number of features, which in turn reduces the number of parameters to be estimated, and can lead to more stable and accurate models.

A) Dataset with 20 data points



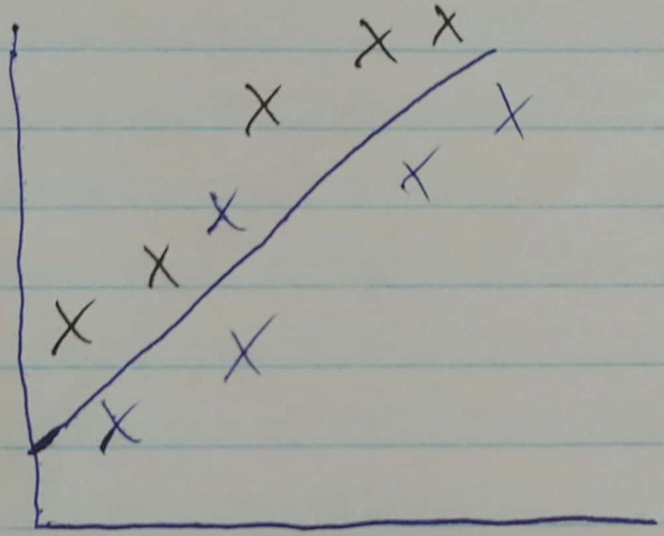
The data points in each class are unclustered random variables. L D A performs better in this setting than K N N because this is a small data set with a decision boundary that is linear.

Hilroy

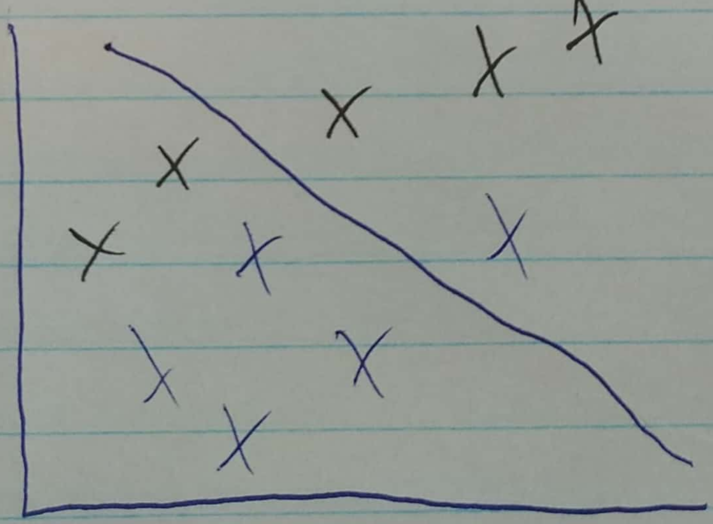
This means that the data points are
evenly spaced out all across the graphs
on both sides of the decision boundary.

Testing:

LDA:

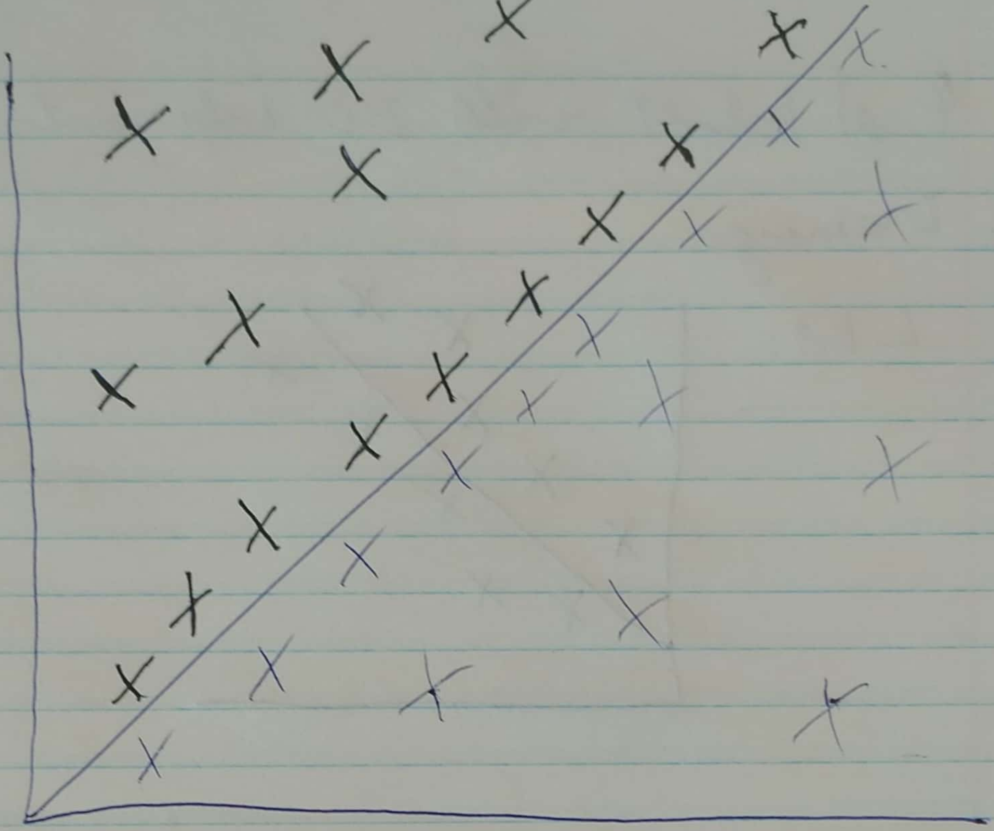


KNN:



As we can see on the graphs LDA has a low variance and bias with the data points evenly centered around the decision boundary. As for KNN it has a high variance and bias with the data points scattered on the graph crossing over to the other side of the decision boundary.

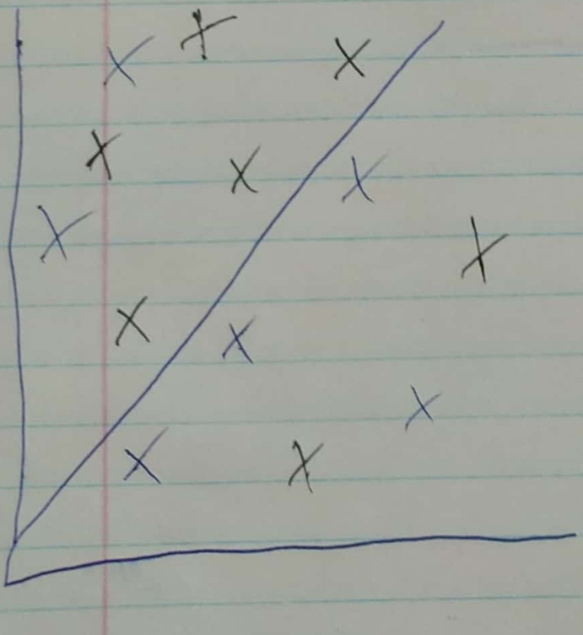
b)



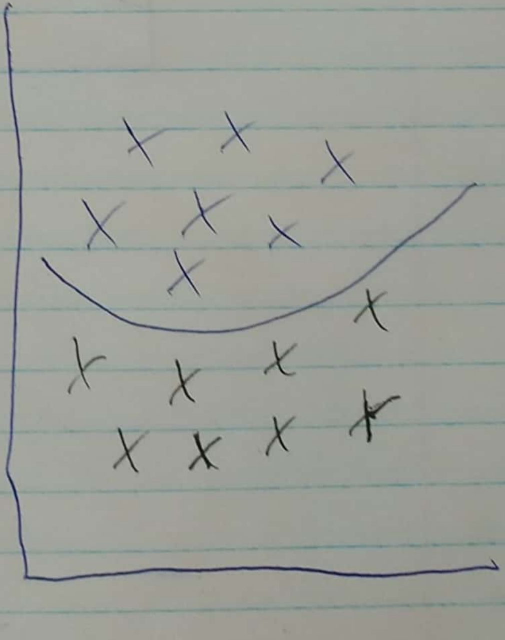
Let's take a balanced dataset with 20 data points. The data points in each class are random variables. If there is a zero corelation between the two classes then GNB with pooled variance would perform better than LDA.

testing:

LDA:



GNB:



As you can see for GNB, the data points are evenly distributed on both sides of the decision boundary. this based on the fact that on the original dataset wine data points were evenly centered around the decision boundary but there few a data points all around the graph in random places and since GNB calculates point by point all of thes off-diagonal points weren't taken ^that^ seriously. On the other hand for LDA because it calculates the whole data together all the data points were merged together and created a distribution of uneven data points all spreading all the graph and getting on the other side of the decision boundary

Question 5)

a)

```r
load('mnist23small.Rdata') # 1000 training digits, 1000 test
test$y = as.factor(test$y)
train$y = as.factor(train$y)
# prcomp() below will automatically centre the data
# (and optionally scale it, though this is not needed),
# so we don't normalise it here.
Xtrain = subset(train, select = -y)
Xtest = subset(test, select = -y)
pca.fit = prcomp(Xtrain, center=TRUE, retx = TRUE)
M = 20
ztrain = pca.fit$x[,1:M]
# alternatively, the following would produce the same result:
#ztrain = predict(pca.fit, newdata=Xtrain)[,1:M]
train_pc = data.frame(y=train$y, ztrain)
ztest = predict(pca.fit, newdata=Xtest)[,1:M]
test_pc = data.frame(y=test$y, ztest)

error.rate = function(ypred, ytrue){
err.rate = mean(ypred != ytrue)
return(err.rate)
}

library(class)
```

```
## Warning: package 'class' was built under R version 4.1.3
```

```r
library(caret)
```

```
## Warning: package 'caret' was built under R version 4.1.3

## Loading required package: ggplot2

## Warning: package 'ggplot2' was built under R version 4.1.3

## Loading required package: lattice
```

```r
Xtrain = subset(train, select = -y)
Xtest = subset(test, select = -y)

start.time = proc.time()
ypred = knn(Xtrain , Xtest ,train$y, k = 30)
print(paste('Error rate on original dataset = ', error.rate(ypred,test$y)))
```

```
## [1] "Error rate on original dataset =  0.027"
```

```r
elapsed.time.knn.test = proc.time() - start.time
print(paste('KNN Test Time on original dataset:', round(elapsed.time.knn.test[1], 2), ' seconds'))
```

```
## [1] "KNN Test Time on original dataset: 8.55  seconds"
```

B)

```r
library(class)
library(caret)

Xtrain_pc = subset(train_pc, select = -y)
Xtest_pc = subset(test_pc, select = -y)

start.time = proc.time()
ypred = knn(Xtrain_pc , Xtest_pc ,train_pc$y, k = 30)
print(paste('Error rate on the first 20 PC scores = ', error.rate(ypred,test_pc$y)))
```

```
## [1] "Error rate on the first 20 PC scores =  0.021"
```

```r
elapsed.time.knn.test = proc.time() - start.time
print(paste('KNN Test Time on the first 20 PC scores :', round(elapsed.time.knn.test[1], 2), ' seconds'
```

```
## [1] "KNN Test Time on the first 20 PC scores : 0.04  seconds"
```

C)

```r
library(caret)
set.seed(1)
# Select the best K in KNN by cross-validation
ctrl = trainControl(method="cv")
knn_cv = train(y ~ ., data=train_pc, method = "knn", trControl = ctrl,
tuneLength = 10)
# The returned variable `knn_cv` contains many attributes,

# including a final trained model with an optimal K, in this case K=7.
# Uncomment the following line to print a summary:
#knn_cv
knn_pred = predict(knn_cv, newdata = test_pc)
print(paste('KNN (optimal K) Test Error rate = ', error.rate(knn_pred,test_pc$y)))
```

```
## [1] "KNN (optimal K) Test Error rate =  0.015"
```