Lab 10 - Dimensionality Reduction and Evaluation Metrics

Ken Ye

11/8/2023

1. Principal Components Analysis (PCA)

In this lab, we perform PCA on the `USArrests` data set, which is part of the base `R` package. The rows of the data set contain the 50 states, in alphabetical order.

```
states <- row.names(USArrests)
states
```

```
##  [1] "Alabama"        "Alaska"         "Arizona"        "Arkansas"
##  [5] "California"     "Colorado"       "Connecticut"    "Delaware"
##  [9] "Florida"        "Georgia"        "Hawaii"         "Idaho"
## [13] "Illinois"       "Indiana"        "Iowa"           "Kansas"
## [17] "Kentucky"       "Louisiana"      "Maine"          "Maryland"
## [21] "Massachusetts"  "Michigan"       "Minnesota"      "Mississippi"
## [25] "Missouri"       "Montana"        "Nebraska"       "Nevada"
## [29] "New Hampshire"  "New Jersey"     "New Mexico"     "New York"
## [33] "North Carolina" "North Dakota"   "Ohio"           "Oklahoma"
## [37] "Oregon"         "Pennsylvania"   "Rhode Island"   "South Carolina"
## [41] "South Dakota"   "Tennessee"      "Texas"          "Utah"
## [45] "Vermont"        "Virginia"       "Washington"     "West Virginia"
## [49] "Wisconsin"      "Wyoming"
```

The columns of the data set contain the four variables:

```
names(USArrests)
```

```
## [1] "Murder"   "Assault"  "UrbanPop" "Rape"
```

We first briefly examine the data. We notice that the variables have vastly different means:

```
apply(USArrests, 2, mean)
```

```
##   Murder  Assault UrbanPop     Rape
##    7.788  170.760   65.540   21.232
```

Note that the `apply()` function allows us to apply a function - in this case, the `mean()` function - to each row or column of the data set. The second input here denotes whether we wish to commute the mean of the rows, `1`, or the columns, `2`. We see that there are many more assaults than other types of violent crimes. We can also examine the variances of the four variables, using the `apply()` function.

```
apply(USArrests, 2, var)
```

```
##     Murder    Assault   UrbanPop       Rape
##   18.97047 6945.16571  209.51878   87.72916
```

Not surprisingly, the variables also have vastly different variances. The `UrbanPop` variable measures the percentage of the population in each state living in an urban area, which is not a comparable number to the number of violent crimes in each state of each type per 100,000 individuals.

If we failed to scale the variables before performing PCA, then most of the principal components that we observed would be driven by the `Assault` variable, since it has by far the largest mean and variance. Thus, it is important to standardize the variables to have mean zero and standard deviation one before performing PCA.

We now perform principal components analysis using the `prcomp()` function, which is one of several functions in `R` that perform PCA.

```
pr.out <- prcomp(USArrests, scale = TRUE)
```

By default, the `prcomp()` function centers the variables to have mean zero. By using the option `scale = TRUE`, we scale the variables to have standard deviation one. The output from `prcomp()` contains a number of useful quantities.

```
names(pr.out)
```

```
## [1] "sdev"     "rotation" "center"   "scale"    "x"
```

The `center` and `scale` components correspond to the means and standard deviations of the variables that were used for scaling prior to implementing PCA.

```
pr.out$center
```

```
##   Murder  Assault UrbanPop     Rape
##    7.788  170.760   65.540   21.232
```

```
pr.out$scale
```

```
##    Murder   Assault  UrbanPop      Rape
##  4.355510 83.337661 14.474763  9.366385
```

The `rotation` matrix provides the principal component loadings; each column of `pr.out$rotation` contains the corresponding principal component loading vector.

```
pr.out$rotation
```

```
##                 PC1        PC2        PC3         PC4
## Murder   -0.5358995 -0.4181809  0.3412327  0.64922780
## Assault  -0.5831836 -0.1879856  0.2681484 -0.74340748
## UrbanPop -0.2781909  0.8728062  0.3780158  0.13387773
## Rape     -0.5434321  0.1673186 -0.8177779  0.08902432
```

We see that there are four distinct principal components. This is to be expected because there are in general $\min(n-1, p)$ informative principal components in a data set with n observations and p variables.

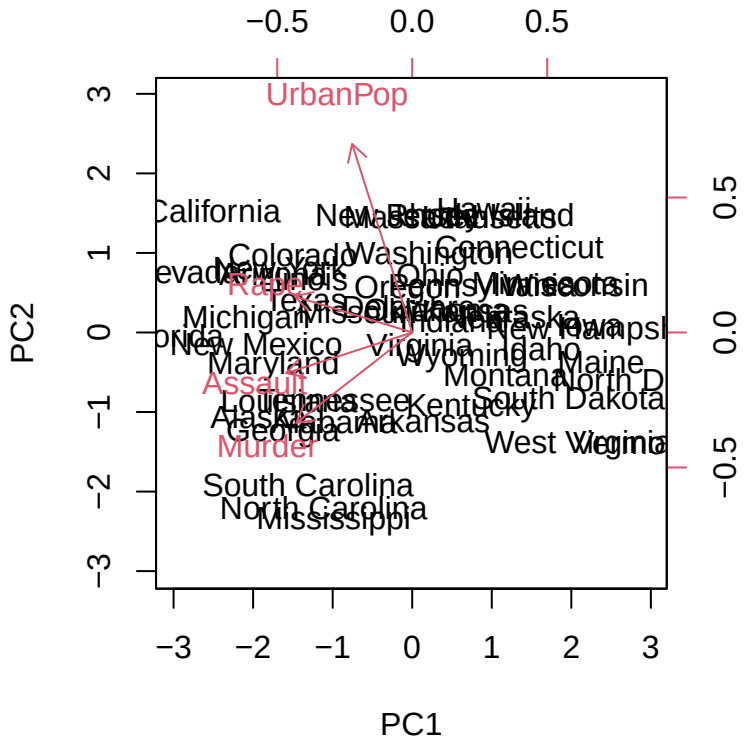
Using the `prcomp()` function, we do not need to explicitly multiply the data by the principal component loading vectors in order to obtain the principal component score vectors. Rather the 50x4 matrix x has as its columns the principal component score vectors. That is, the kth column is the kth principal component score vector.

```
dim(pr.out$x)
```

```
## [1] 50  4
```

We can plot the first two principal components as follows:

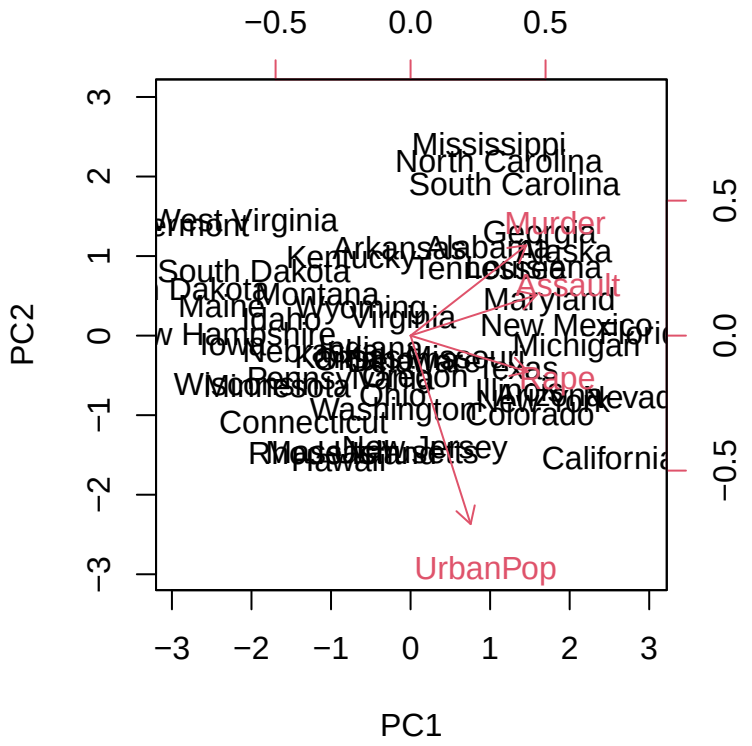
```
biplot(pr.out, scale = 0)
```



The `scale = 0` argument to `biplot()` ensures that the arrows are scaled to represent the loadings; other values for `scale` give slightly different biplots with different interpretations.

Notice that this figure is a mirror image of Figure 10.1. Recall that the principal components are only unique up to a sign change, so we can reproduce Figure 10.1 by making a few small changes:

```
pr.out$rotation <- -pr.out$rotation
pr.out$x <- -pr.out$x
biplot(pr.out, scale = 0)
```



The `prcomp()` function also outputs the standard deviation of each principal component. For instance, on the `USArrests` data set, we can access these standard deviations as follows:

```
pr.out$sdev
```

```
## [1] 1.5748783 0.9948694 0.5971291 0.4164494
```

The variance explained by each principal component is obtained by squaring these:

```
pr.var <- pr.out$sdev^2
pr.var
```

```
## [1] 2.4802416 0.9897652 0.3565632 0.1734301
```

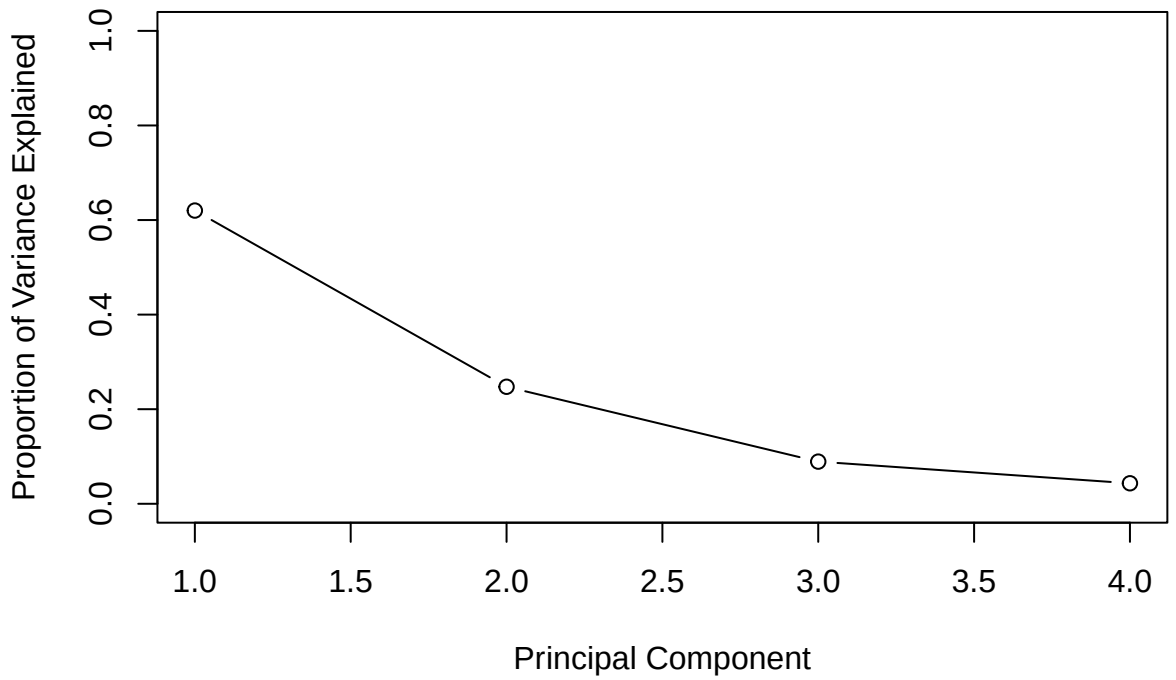
To compute the proportion of variance explained by each principal component, we simply divide the variance explained by each principal component by the total variance explained by all four principal components:

```
pve <- pr.var/sum(pr.var)
pve
```

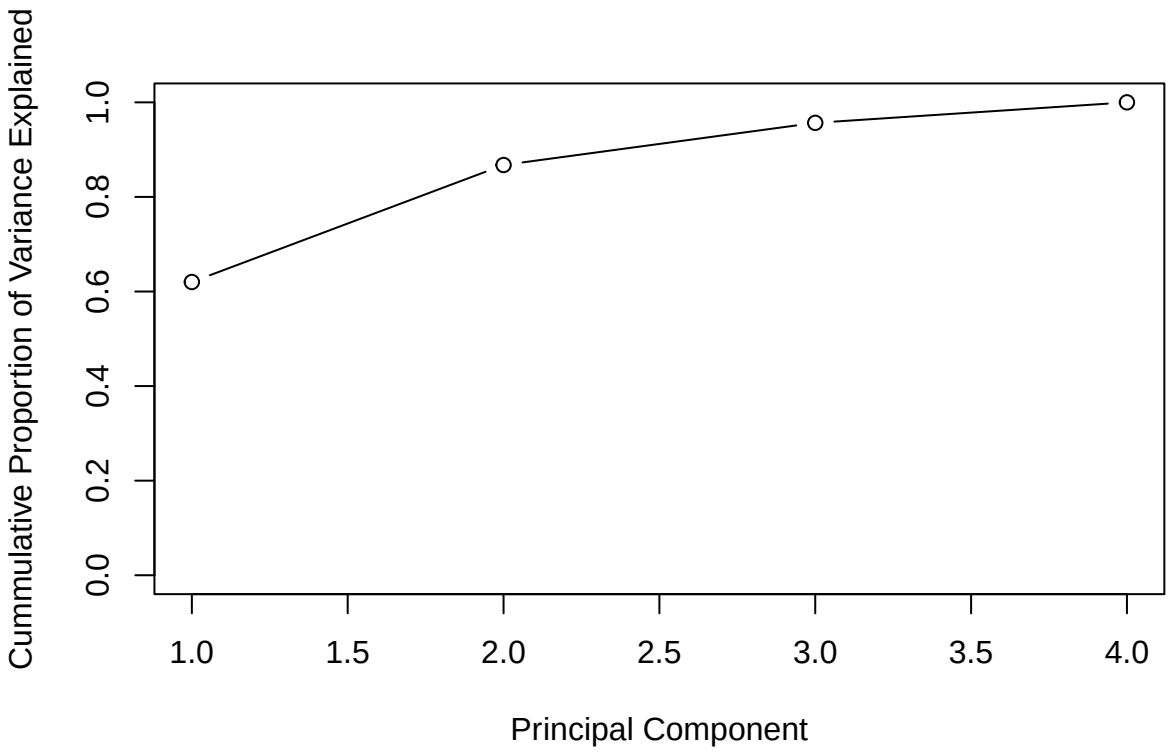
```
## [1] 0.62006039 0.24744129 0.08914080 0.04335752
```

We see that the first principal component explains 62.0% of the variance in the data, the next principal component explains 24.7% of the variance, and so forth. We can plot the PVE explained by each component, as well as the cumulative PVE, as follows:

```
plot(pve, xlab = "Principal Component",
     ylab = "Proportion of Variance Explained", ylim = c(0,1),
     type = 'b')
```




```
plot(cumsum(pve), xlab = "Principal Component",
     ylab = "Cummulative Proportion of Variance Explained",
     ylim = c(0,1), type = 'b')
```



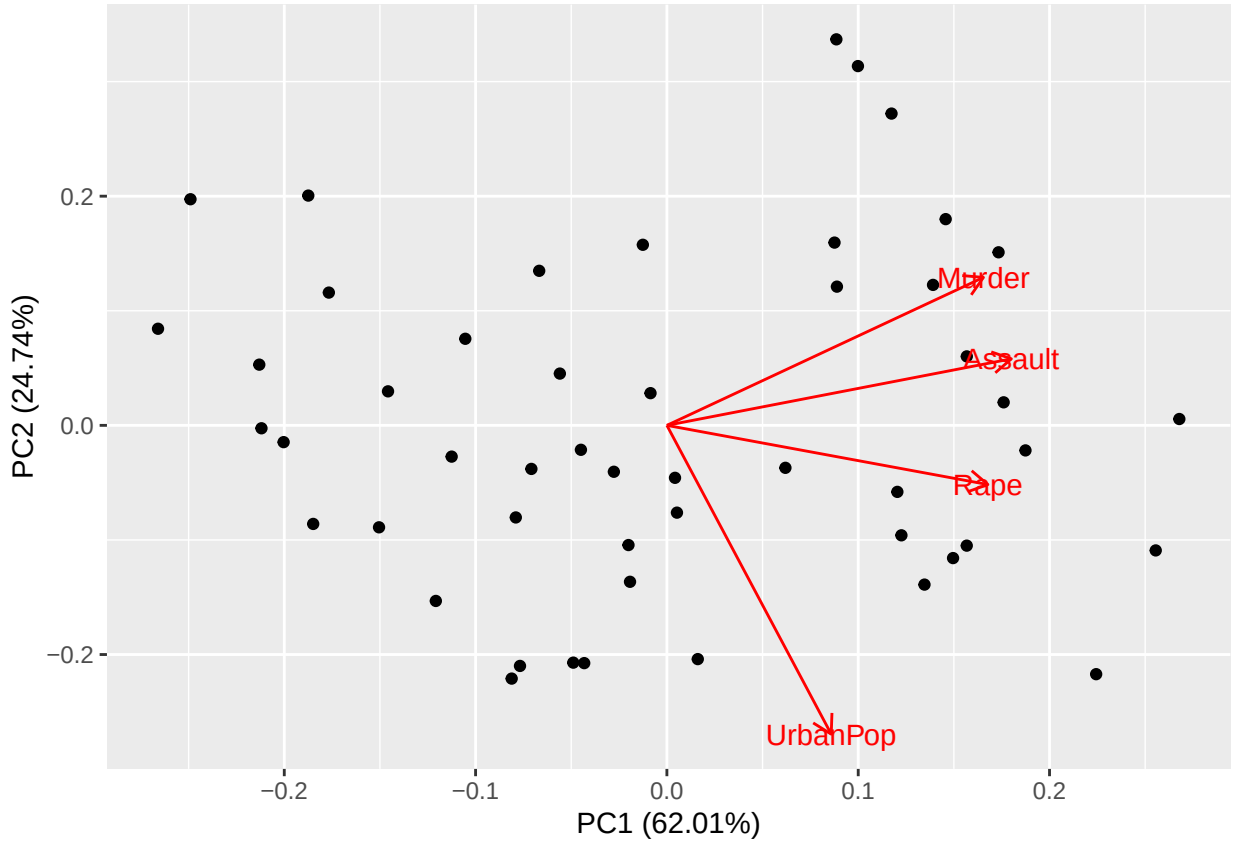
The result is shown in Figure 10.4. Note that the function `cumsum()` computes the cumulative sum of the elements of a numeric vector. For instance,

```
a <- c(1,2,8,-3)
cumsum(a)
```

```
## [1]  1  3 11  8
```

There are many different ways to calculate and display PCA in `R`. For example, a `ggplot` version can be plotted using the `ggfortify` package. This package also allows for several other types of commonly used plots beyond the functionality provided in regular `ggplot`, check out http://www.sthda.com/english/wiki/ggfortify-extension-to-ggplot2-to-handle-some-popular-packages-r-software-and-data-visualization.

```
library(ggfortify)
autoplot(pr.out, loadings = TRUE, loadings.label = TRUE,
         data = USArrests)
```



2. tSNE

Another commonly used dimensionality reduction technique is called t-Distributed Stochastic Neighbor Embedding. We can also run this in `R` on the `iris` dataset.

```
library(Rtsne)
library(ggplot2)
data(iris)
```

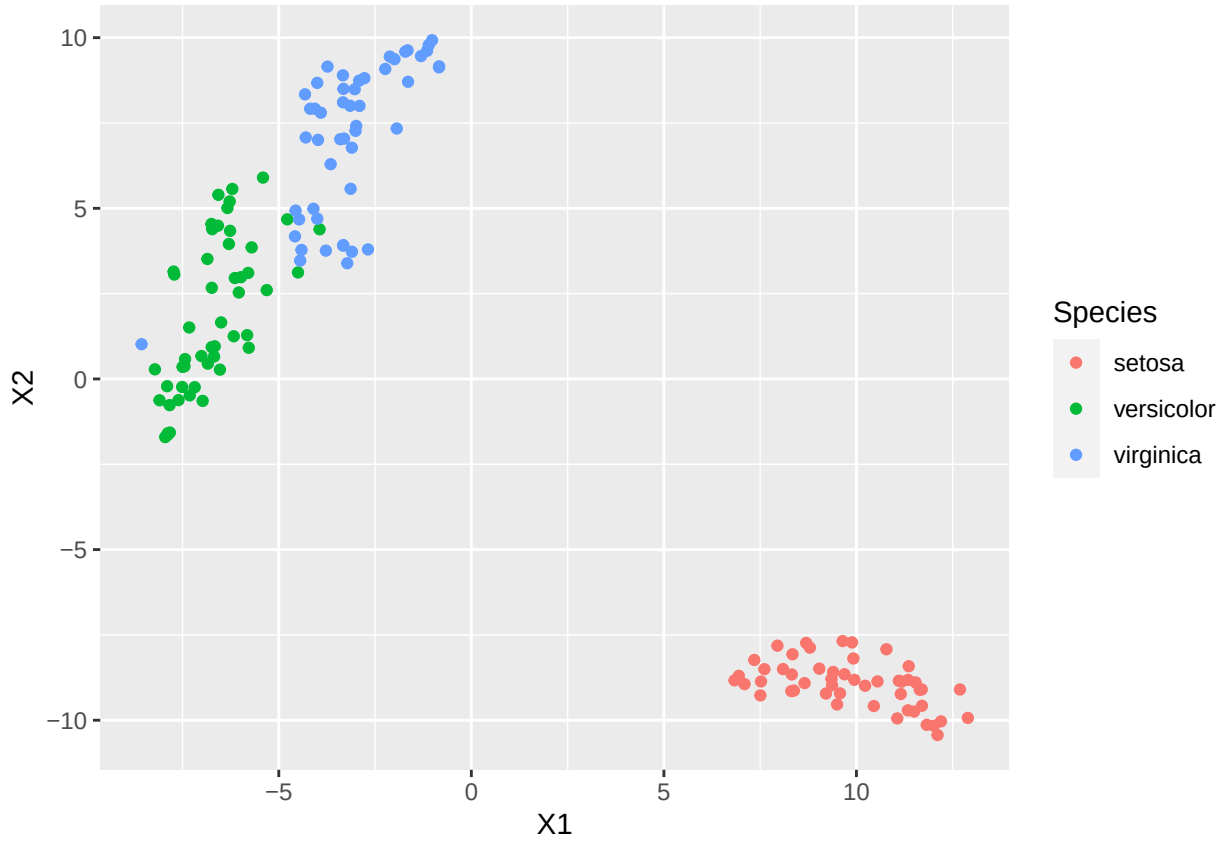
```
tsne <- Rtsne(iris[,-5], dims = 2, perplexity=30,
              verbose=TRUE,max_iter = 500,
              check_duplicates = FALSE)
```

```
## Performing PCA
## Read the 150 x 4 data matrix successfully!
## Using no_dims = 2, perplexity = 30.000000, and theta = 0.500000
## Computing input similarities...
## Building tree...
## Done in 0.00 seconds (sparsity = 0.706711)!
## Learning embedding...
## Iteration 50: error is 46.774650 (50 iterations in 0.01 seconds)
## Iteration 100: error is 44.894224 (50 iterations in 0.01 seconds)
## Iteration 150: error is 44.468849 (50 iterations in 0.01 seconds)
## Iteration 200: error is 44.217121 (50 iterations in 0.01 seconds)
```

```
## Iteration 250: error is 43.498370 (50 iterations in 0.01 seconds)
## Iteration 300: error is 0.339178 (50 iterations in 0.01 seconds)
## Iteration 350: error is 0.131432 (50 iterations in 0.01 seconds)
## Iteration 400: error is 0.128795 (50 iterations in 0.00 seconds)
## Iteration 450: error is 0.127497 (50 iterations in 0.00 seconds)
## Iteration 500: error is 0.128331 (50 iterations in 0.01 seconds)
## Fitting performed in 0.06 seconds.
```

```r
embedding <- data.frame(tsne$Y)
embedding$Species <- iris$Species

ggplot(embedding, aes(x = X1, y = X2, color = Species)) +
  geom_point()
```



We can see that the `iris` data separates well by species when we take into account all 4 features and perform a tSNE dimensionality reduction. A linear classifier would do quite well on this transformed data.

3. Evaluation Metrics

3.1 Classification

Write a function that calculates the precision, recall and accuracy of a classifier and also outputs the confusion matrix. Assume that the inputs to the function are the true labels and the predicted labels for the same input data.

Fit a classifier of your choice on the iris data and test your function (only use the species versicolor and virginica for the classifer so there are only 2 classes). Build a different model and compare the two results using your evaluation function. Which model would you select and why based on the outputs of your function?

```
class.eval <- function(true.labels, pred.labels) {
  # Calculate confusion matrix
  conf_matrix <- table(true.labels, pred.labels)

  # Calculate precision, recall, and accuracy
  precision <- conf_matrix[2, 2] / sum(conf_matrix[, 2])
  recall <- conf_matrix[2, 2] / sum(conf_matrix[2, ])
  accuracy <- sum(diag(conf_matrix)) / sum(conf_matrix)

  # Print confusion matrix and evaluation metrics
  cat("Confusion Matrix:\n")
  print(conf_matrix)

  cat("\nPrecision:", precision, "\n")
  cat("Recall:", recall, "\n")
  cat("Accuracy:", accuracy, "\n")
}
```

```
true.labs <- c(0,1,1,0,1,0,1)
pred.labs <- c(0,0,1,1,1,1,0)
class.eval(true.labs, pred.labs)
```

```
## Confusion Matrix:
##            pred.labels
## true.labels 0 1
##           0 1 2
##           1 2 2
##
## Precision: 0.5
## Recall: 0.5
## Accuracy: 0.4285714
```

```
# Filter the dataset to include only 'versicolor' and 'virginica' classes
iris_subset <- subset(iris, Species %in% c("versicolor", "virginica"))

# Remove 'setosa' level from the 'Species' variable
iris_subset$Species <- droplevels(iris_subset$Species, exclude = "setosa")
```

```
# Fit Logistic Regression model
model_logreg <- glm(Species ~ ., data = iris_subset, family = "binomial")
predictions_logreg <- predict(model_logreg, newdata = iris_subset, type = "response") > 0.5
predictions_logreg <- as.factor(ifelse(predictions_logreg, "virginica", "versicolor"))

# Evaluate Logistic Regression model
class.eval(iris_subset$Species, predictions_logreg)
```

```
## Confusion Matrix:
```

```
##             pred.labels
## true.labels  versicolor virginica
##   versicolor         49         1
##   virginica           1        49
##
## Precision: 0.98
## Recall: 0.98
## Accuracy: 0.98
```

```
library(rpart)
# Fit Decision Tree model
model_dt <- rpart(Species ~ ., data = iris_subset, method = "class")
predictions_dt <- predict(model_dt, newdata = iris_subset, type = "class")

class.eval(iris_subset$Species, predictions_dt)
```

```
## Confusion Matrix:
##             pred.labels
## true.labels  versicolor virginica
##   versicolor         49         1
##   virginica           5        45
##
## Precision: 0.9782609
## Recall: 0.9
## Accuracy: 0.94
```

Comparing the logistic regression model and the decision tree model, I would choose the logistic regression model because its precision, recall, and accuracy values are all higher.

3.2 Implementation

Give an example of a case where false positives are much worse than false negatives. When is a case where false negatives could be much worse than false positives?

- In the spam email filtering scenario, false positives can inconvenience users by filtering out legitimate emails, but it is usually not life-threatening.
- In the medical testing scenario, false negatives can have serious consequences, as failing to identify a severe disease can delay treatment and harm the patient.

3.3 Regression

Write a function that calculates the MSE, RMSE and mean absolute error for a regression function. The inputs to your function should be the true Y_i values and the predicted $\hat{Y}_i$ values.

Use the `mtcars` data to test your function. Build a regression model of your choice to test your function. Assume a 70%-30% training-test split and evaluate your model on the test set. The response variable is `mpg`.

```
data(mtcars)
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
regress.eval <- function(true.vals, pred.vals) {
  # Calculate Mean Squared Error (MSE)
  mse <- mean((true.vals - pred.vals)^2)

  # Calculate Root Mean Squared Error (RMSE)
  rmse <- sqrt(mse)

  # Calculate Mean Absolute Error (MAE)
  mae <- mean(abs(true.vals - pred.vals))

  # Print evaluation metrics
  cat("Mean Squared Error (MSE):", mse, "\n")
  cat("Root Mean Squared Error (RMSE):", rmse, "\n")
  cat("Mean Absolute Error (MAE):", mae, "\n")
}
```

```
toy.true <- c(1,2,3,4,5)
toy.pred <- c(2,1,3.1,5.4, 9.0)
regress.eval(toy.true, toy.pred)
```

```
## Mean Squared Error (MSE): 3.994
## Root Mean Squared Error (RMSE): 1.998499
## Mean Absolute Error (MAE): 1.5
```

```
set.seed(1)
train_indices <- sample(1:nrow(mtcars), 0.7 * nrow(mtcars))
train_data <- mtcars[train_indices, ]
test_data <- mtcars[-train_indices, ]
```

```
# Fit a linear regression model
model <- lm(mpg ~ ., data = train_data)

# Make predictions on the test set
predictions <- predict(model, newdata = test_data)

# Evaluate the regression model using the custom function
regress.eval(test_data$mpg, predictions)
```

```
## Mean Squared Error (MSE): 14.18818
## Root Mean Squared Error (RMSE): 3.76672
## Mean Absolute Error (MAE): 3.411117
```