Exploring R

Basic Data Structures in R

In R, several basic data structures are commonly used for organizing and manipulating data. These include vectors, matrices, arrays, data frames, and lists. Let's explore each of them briefly:

1. **Vectors**-

- A vector is a one-dimensional array that can hold elements of the same data type.
- You can create a vector using the `c()` function.

```
# Example of creating a numeric vector
numeric_vector <- c(1, 2, 3, 4, 5)

# Example of creating a character vector
character_vector <- c("apple", "orange", "banana")
```

2. **Matrices-**

- A matrix is a two-dimensional array that can hold elements of the same data type.
- You can create a matrix using the `matrix()` function.

```
# Example of creating a matrix
matrix_example <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol =
```

3. **Arrays-**

    - An array is a multi-dimensional structure that can hold elements of the same data type.
    - You can create an array using the `array()` function.

    ```
    # Example of creating a 3-dimensional array
    array_example <- array(1:24, dim = c(2, 3, 4))
    ```

4. **Data Frames-**

   - A data frame is a two-dimensional structure similar to a matrix, but it can store columns of different data types.
   - You can create a data frame using the `data.frame()` function.

```
# Example of creating a data frame
data_frame_example <- data.frame(
  Name = c("John", "Alice", "Bob"),
  Age = c(25, 30, 22),
  Gender = c("Male", "Female", "Male")
)
```

5. **Lists-**

   - A list is a versatile data structure that can contain elements of different data types, including other lists.
   - You can create a list using the `list()` function.

```
# Example of creating a list
list_example <- list(
  numeric_vector,
  character_vector,
  matrix_example,
  data_frame_example
)
```

These are the basic data structures in R. They provide a solid foundation for working with and organizing data in various ways. Depending on your specific needs and the complexity of your data, you can choose the appropriate data structure.

Linear Regression in R

Use Case of revenue prediction, featuring linear regression

Predicting the revenue from paid, organic, and social media traffic using a linear regression model in R.

We will now look at a real-life scenario where we will predict the revenue by using regression analysis in R. The sample dataset we will be working with is shown below:

	A	B	C	D
1	Paid	Organic	Social	Revenue
2	165349	136898	471784	192261.8
3	162598	151378	443899	191792.1
4	153442	101146	407935	191050.4
5	144372	118672	383200	182902
6	142107	91392	366168	166187.9
7	131877	99815	362861	156991.1
8	134615	147199	127717	156122.5
9	130298	145530	323877	155752.6
10	120543	148719	311613	152211.8
11	123335	108679	304982	149760
12	101913	110594	229161	146122
13	100672	91791	249745	144259.4
14	93864	127320	249839	141585.5
15	91992	135495	252665	134307.4
16	119943	156547	256513	132602.7

In this demo, we will work with the following three attributes to predict the revenue:

1. **Paid Traffic** - Traffic coming through advertisement
2. **Organic Traffic** - Traffic from search engines, which is non-paid
3. **Social Traffic** - Traffic coming in from various social networking sites

We will be making use of multiple linear regression. The linear regression formula is:

$$Y = m_1 * x_1 + m_2 * x_2 + m_3 * x_3 + c$$

m is the slope

Y is the Revenue

x_1 is Paid Traffic

x_2 is Organic Traffic

x_3 is Social Traffic

Before we begin, let's have a look at the program's flow:

1. Generate inputs using csv files
2. Import the required libraries
3. Split the dataset into train and test
4. Apply the regression on paid traffic, organic traffic, and social traffic
5. Validate the model

```
# Import the dataset
sales <- read.csv('Revenue.csv')
head(sales) #Displays the top 6 rows of a dataset
summary(sales) #Gives certain statistical information about t
```

#The output will look like below:

```
> head(sales)
    Paid Organic Social  Revenue
1 165349  136898 471784 192261.8
2 162598  151378 443899 191792.1
3 153442  101146 407935 191050.4
4 144372  118672 383200 182902.0
5 142107   91392 366168 166187.9
6 131877   99815 362861 156991.1
```

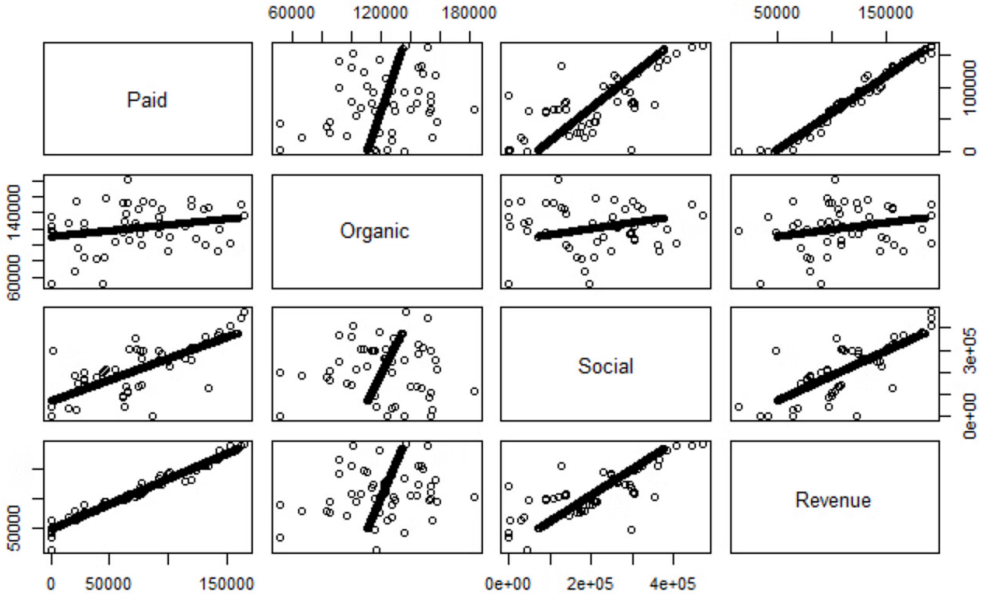
```
> summary(sales)
      Paid           Organic           Social      
 Min.   :     0   Min.   : 51283   Min.   :     0  
 1st Qu.: 43085   1st Qu.:116641   1st Qu.:150970  
 Median : 79755   Median :122410   Median :224033  
 Mean   : 81610   Mean   :122510   Mean   :226246  
 3rd Qu.:124079   3rd Qu.:129060   3rd Qu.:309128  
 Max.   :165349   Max.   :182646   Max.   :471784  
    Revenue      
 Min.   : 14681  
 1st Qu.: 85943  
 Median :117496  
 Mean   :118752  
 3rd Qu.:155033  
 Max.   :192262  
```

```
dim(sales) # Displays the dimensions of the dataset
```

```
> dim(sales)
[1] 1000    4
```

Now, we move onto plotting the variables.

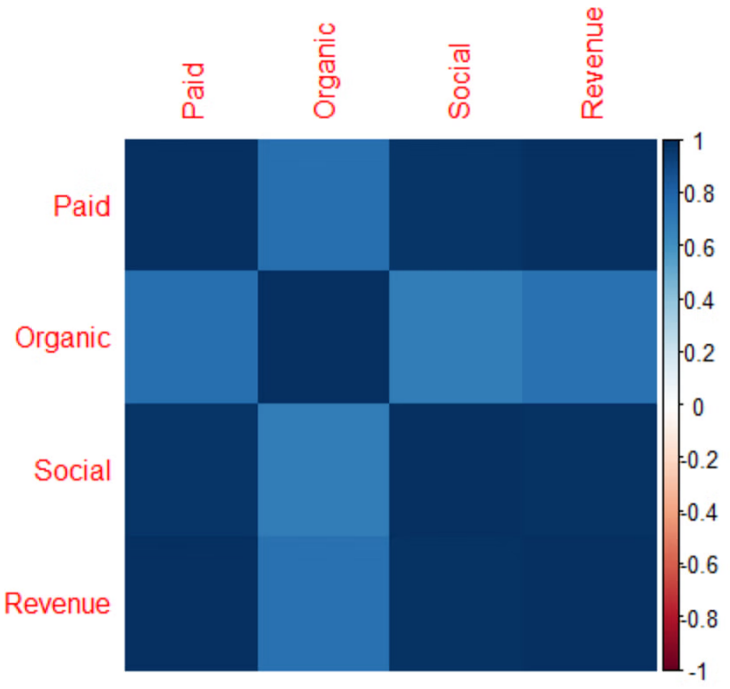
```
plot(sales) # Plot the variables to see their trends
```



Let’s now see how the variables are correlated to each other. For that, we’ll take only the numeric column values.

```
library(corrplot) # Library to finds the correlation between
num.cols<-sapply(sales, is.numeric)
num.cols
cor.data<-cor(sales[,num.cols])
cor.data
corrplot(cor.data, method='color')
```

```
> cor.data
             Paid   Organic    Social   Revenue
Paid    1.0000000 0.7519680 0.9790213 0.9986531
Organic 0.7519680 1.0000000 0.6945577 0.7456682
Social  0.9790213 0.6945577 1.0000000 0.9805634
Revenue 0.9986531 0.7456682 0.9805634 1.0000000
```



As we can see from the above correlation matrix, the variables have a high degree of correlation between each other and with the sales variable.

Let's now split the data from training and testing sets.

```
# Split the data into training and testing
# Split the data into training and testing

set.seed(2)
```

```
library(caTools) #caTools has the split function

split <- sample.split(sales, SplitRatio = 0.7)
# Assigning it to a variable split sample.split is one of the
#using. With the ration value of 0.7, it states that we will
# the sales data for training and 30% for testing the model

split

train <- subset(sales, split = 'TRUE') #Creating a training s

test <- subset(sales, split = 'FALSE') #Creating a testing se

head(train)

head(test)

View(train)

View(test)
```

Now that we have the test and train variables, let's go ahead and create the model:

```
Model <- lm(Revenue ~., data = train)
#Creates the model. Here, lm stands for the linear regression
#Revenue is the target variable we want to track.
summary(Model)
```

```
Call:
lm(formula = Revenue ~ ., data = train)

Residuals:
   Min     1Q Median     3Q    Max
-33534      0      0      0  17276

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  5.012e+04  1.313e+03  38.161  < 2e-16 ***
Paid         8.057e-01  7.605e-03 105.951  < 2e-16 ***
Organic     -2.681e-02  1.097e-02  -2.445   0.0146 *
Social       2.723e-02  3.535e-03   7.701 3.24e-14 ***
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1984 on 996 degrees of freedom
Multiple R-squared:  0.9975,    Adjusted R-squared:  0.9975
F-statistic: 1.335e+05 on 3 and 996 DF,  p-value: < 2.2e-16
```

Prediction

```
pred <- predict(Model, test) #The test data was kept for this
pred <- predict(Model, test) #The test data was kept for this
pred #This displays the predicted values
res<-residuals(Model) # Find the residuals
res<-as.data.frame(res) # Convert the residual into a datafra
res # Prints the residuals
```

Compare the predicted vs actual values

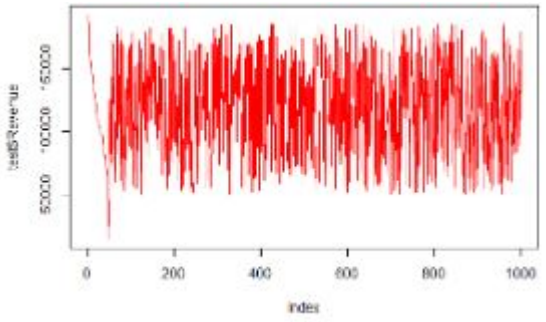
```
results<-cbind(pred,test$Revenue)
results
colnames(results)<-c('predicted','real')
results<-as.data.frame(results)
head(results)
```

```
> head(results)
  predicted     real
1  192521.1 192261.8
2  189157.0 191792.1
3  182147.6 191050.4
4  173696.4 182902.0
5  172139.2 166187.9
6  163580.8 156991.1
```

 # Let’s now, compare the predicted vs actual values

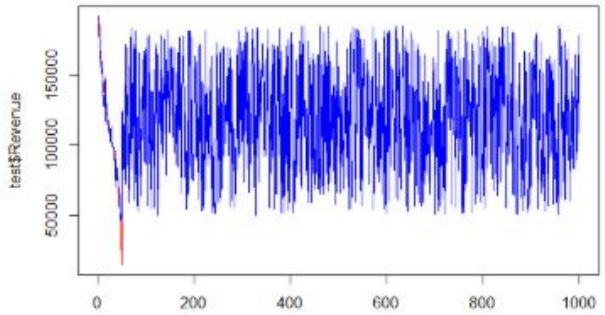
```
plot(test$Revenue, type = 'l', lty = 1.8, col = "red")
```

The output of the above command is shown below in a graph that shows the predicted revenue.

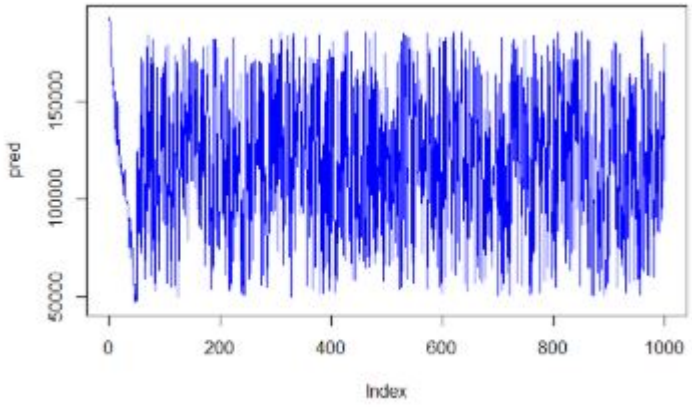


Now let’s plot our test revenue with the following command:

```
lines(pred, type = "l", col = "blue") #The output looks like
```



```
plot(pred, type = "l", lty = 1.8, col = "blue") #The output l
```



From the above output, we can see that the graphs of the predicted revenue and expected revenue are very close. Let’s check out the accuracy so we can validate the comparison.

Calculating the accuracy

```
rmse <- sqrt(mean(pred-sales$Revenue)^2)
# Root Mean Square Error is the standard deviation of the res
rmse
```

The output looks like below:

```
> rmse
[1] 3.339665e-11
```

Logistic Regression in R

We would use a **direct marketing campaign dataset** by a Portuguese banking institution using phone calls. The campaign aims to sell subscriptions of a bank term deposit represented by the variable y (subscription or no subscription). The objective of the logistic regression model is to predict whether a customer would buy a subscription or not based on the predictor variables, aka attributes of the customer, such as demographic information.

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	y
1	58	management	married	tertiary	no	2143	yes	no	unknown	5	may	261	1	-1	0	unknown	no
2	44	technician	single	secondary	no	29	yes	no	unknown	5	may	151	1	-1	0	unknown	no
3	33	entrepreneur	married	secondary	no	2	yes	yes	unknown	5	may	76	1	-1	0	unknown	no
4	47	blue-collar	married	unknown	no	1506	yes	no	unknown	5	may	92	1	-1	0	unknown	no
5	33	unknown	single	unknown	no	1	no	no	unknown	5	may	198	1	-1	0	unknown	no
6	35	management	married	tertiary	no	231	yes	no	unknown	5	may	139	1	-1	0	unknown	no
7	28	management	single	tertiary	no	447	yes	yes	unknown	5	may	217	1	-1	0	unknown	no
8	42	entrepreneur	divorced	tertiary	yes	2	yes	no	unknown	5	may	380	1	-1	0	unknown	no
9	58	retired	married	primary	no	121	yes	no	unknown	5	may	50	1	-1	0	unknown	no
10	43	technician	single	secondary	no	593	yes	no	unknown	5	may	55	1	-1	0	unknown	no

Logistic Regression Packages

In R, there are two popular workflows for modeling logistic regression: base-R and tidymodels.

The base-R workflow models is simpler and includes functions like glm() and summary() to fit the model and generate a model summary.

The tidymodels workflow allows easier management of multiple models and a consistent interface for working with different model types.

Visualize the Data

Import the tidymodels package by calling the library() function.

The dataset is in a CSV file with European-style formatting (commas for decimal places and semi-colons for separators).

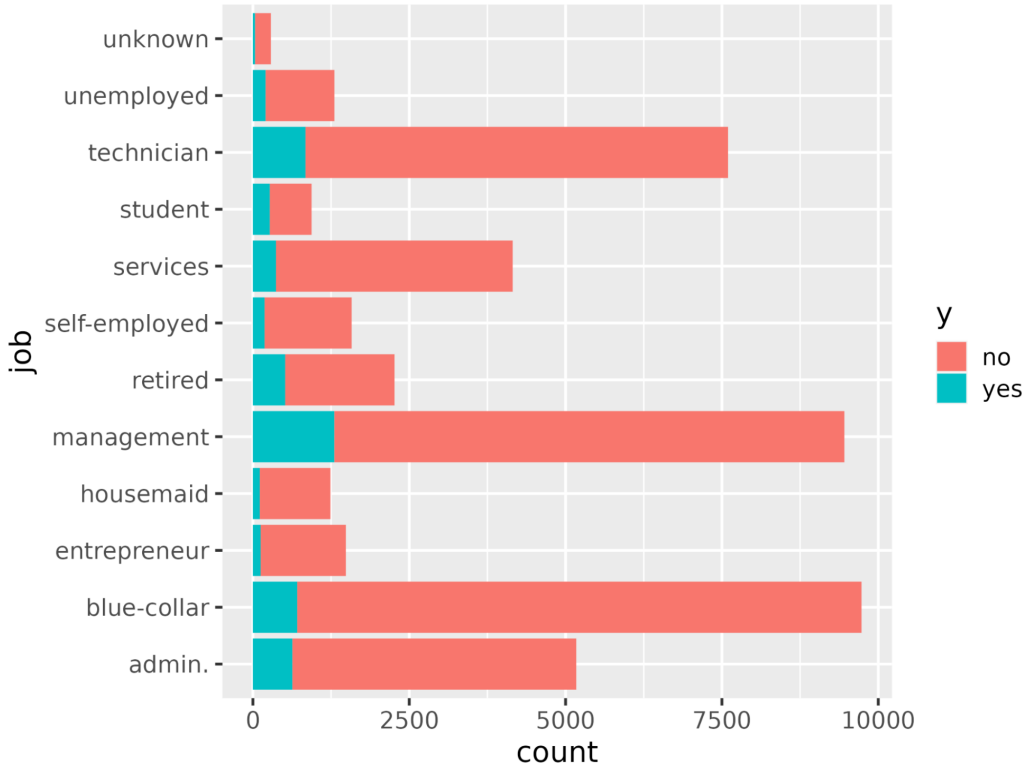
Using the ggplot() function plot the count of each job occupation with respect to y.

```
library(readr)
library(tidymodels)

# Read the dataset and convert the target variable to a facto
bank_df <- read_csv2("bank-full.csv")
```

```
bank_df$y = as.factor(bank_df$y)

# Plot job occupation against the target variable
ggplot(bank_df, aes(job, fill = y)) +
    geom_bar() +
    coord_flip()
```



Splitting the Data

Let's split the dataset into a training set for fitting the model and a test set for model evaluation to make sure the model thus trained works on an unseen dataset.

Splitting data into training and test set can be performed using the initial_split() function and the prop attribute defining the train data proportion.

```
# Split data into train and test
set.seed(421)
split <- initial_split(bank_df, prop = 0.8, strata = y)
train <- split %>%
         training()
```

```
test <- split %>%
        testing()
```

Fitting and Evaluating the Model

To create the model, declare a logistic_reg() model. This needs mixture and penalty arguments which control the amount of regularization. A mixture value of 1 denotes a lasso model and 0 denotes ridge regression. Values in between are also allowed. The penalty argument denotes the strength of the regularization.

Note that we need to pass "double" floating point numbers to the mixture and penalty

Set the "engine" (the backend software used to run the calculations) with set_engine(). There are several choices: The default engine is "glm", which performs a classical logistic regression. This is often favored by statisticians because you get p-values for each coefficient, making it easier to understand the importance of each coefficient.

Here's we'll use the "glmnet" engine. This is favored by machine learning scientists because it allows regularization, which can improve predictions, particularly if we have a lot of features. (In Python, the scikit-learn package defaults to including some regularization in logistic regression.)

Call the fit() method to train the model on the training data created in the previous step.

```
# Train a logistic regression model
model <- logistic_reg(mixture = double(1), penalty = double(1
  set_engine("glmnet") %>%
  set_mode("classification") %>%
  fit(y ~ ., data = train)

# Model summary
tidy(model)
```

The output is shown below with the estimate column representing coefficients of the predictor.

```
# A tibble: 43 × 3
   term              estimate penalty
```

```
   <chr>                <dbl>   <dbl>
 1 (Intercept)      -2.59           0
 2 age              -0.000477       0
 3 jobblue-collar   -0.183          0
 4 jobentrepreneur  -0.206          0
 5 jobhousemaid     -0.270          0
 6 jobmanagement    -0.0190         0
 7 jobretired        0.360          0
 8 jobself-employed -0.101          0
 9 jobservices      -0.105          0
10 jobstudent        0.415          0
# ... with 33 more rows
# ℹ Use `print(n = ...)` to see more rows
```

Making predictions

Make predictions on the testing data using the predict() function. We have choice of the type of predictions.

- type = "class" returns the most likely target value for each observation. Here, it will return a "yes" or a "no" depending on whether the model thinks the client is likely to subscribe to a term deposit or not.
- type = "prob" returns the probability of each target value for each observation. Here, it will return a the probability of a "yes" and the probability of a "no" (which add up to one for each observation).

```
# Class Predictions
pred_class <- predict(model,
                      new_data = test,
                      type = "class")

# Class Probabilities
pred_proba <- predict(model,
                      new_data = test,
                      type = "prob")
```

Evaluate the model using the accuracy() function with truth argument as y and estimate the argument value to be the predictions from the previous step.

```
results <- test %>%
           select(y) %>%
           bind_cols(pred_class, pred_proba)

accuracy(results, truth = y, estimate = .pred_class)
```

Hyperparameter Tuning

Rather than passing specific values to the mixture and penalty arguments (the "hyperparameters"), you can optimize the predictive power of the model by tuning it.

The idea is that you run the model lots of times with different values of the hyperparameters, and see which one gives the best predictions.

- In the logistic_reg() function, set the mixture and penalty arguments to a call to tune().
- Use the grid_regular() function to define a grid of possible values for mixture and penalty.
- The workflow() function creates an object to store the model details, which is needed when you run it many times.
- Choose the best model using the select_best() function. You can pick from a choice of metrics that defines the "best" one. Here, we'll use the "receiver operating characteristic area under the curve" (ROC AUC) metric.

```
# Define the logistic regression model with penalty and mix
ture hyperparameters
log_reg <- logistic_reg(mixture = tune(), penalty = tune(),
engine = "glmnet")

# Define the grid search for the hyperparameters
grid <- grid_regular(mixture(), penalty(), levels = c(mixtu
re = 4, penalty = 3))

# Define the workflow for the model
log_reg_wf <- workflow() %>%
  add_model(log_reg) %>%
```

```
  add_formula(y ~ .)

# Define the resampling method for the grid search
folds <- vfold_cv(train, v = 5)

# Tune the hyperparameters using the grid search
log_reg_tuned <- tune_grid(
  log_reg_wf,
  resamples = folds,
  grid = grid,
  control = control_grid(save_pred = TRUE)
)

select_best(log_reg_tuned, metric = "roc_auc")
```

```
# A tibble: 1 × 3
       penalty mixture .config
         <dbl>   <dbl> <chr>
1 0.0000000001       0 Preprocessor1_Model01
```

More Evaluation Metrics

Using the best hyperparameters:

1. Let's train a logistic regression model
2. Use it to generate predictions on test set
3. Create a confusion matrix using the true values, and the estimates

```
# Fit the model using the optimal hyperparameters
log_reg_final <- logistic_reg(penalty = 0.0000000001, mixtu
re = 0) %>%
                 set_engine("glmnet") %>%
                 set_mode("classification") %>%
                 fit(y~., data = train)

# Evaluate the model performance on the testing set
pred_class <- predict(log_reg_final,
```

```
                       new_data = test,
                       type = "class")
results <- test %>%
  select(y) %>%
  bind_cols(pred_class, pred_proba)

# Create confusion matrix
conf_mat(results, truth = y,
         estimate = .pred_class)
```

```
Truth
Prediction   no  yes
       no  7838  738
       yes  147  320
```

We can calculate the precision (positive predictive value, the number of true positives divided by the number of predicted positives) with the precision() function.

```
precision(results, truth = y,
          estimate = .pred_class)
```

```
# A tibble: 1 × 3
  .metric   .estimator .estimate
  <chr>     <chr>          <dbl>
1 precision binary         0.914
```

Similarly, we can calculate the recall (sensitivity, the number of true positives divided by the number of actual positives) with the recall() function.

```
recall(results, truth = y,
          estimate = .pred_class)
```

```
# A tibble: 1 × 3
  .metric .estimator .estimate
```

```
  <chr>   <chr>          <dbl>
1 recall  binary         0.982
```

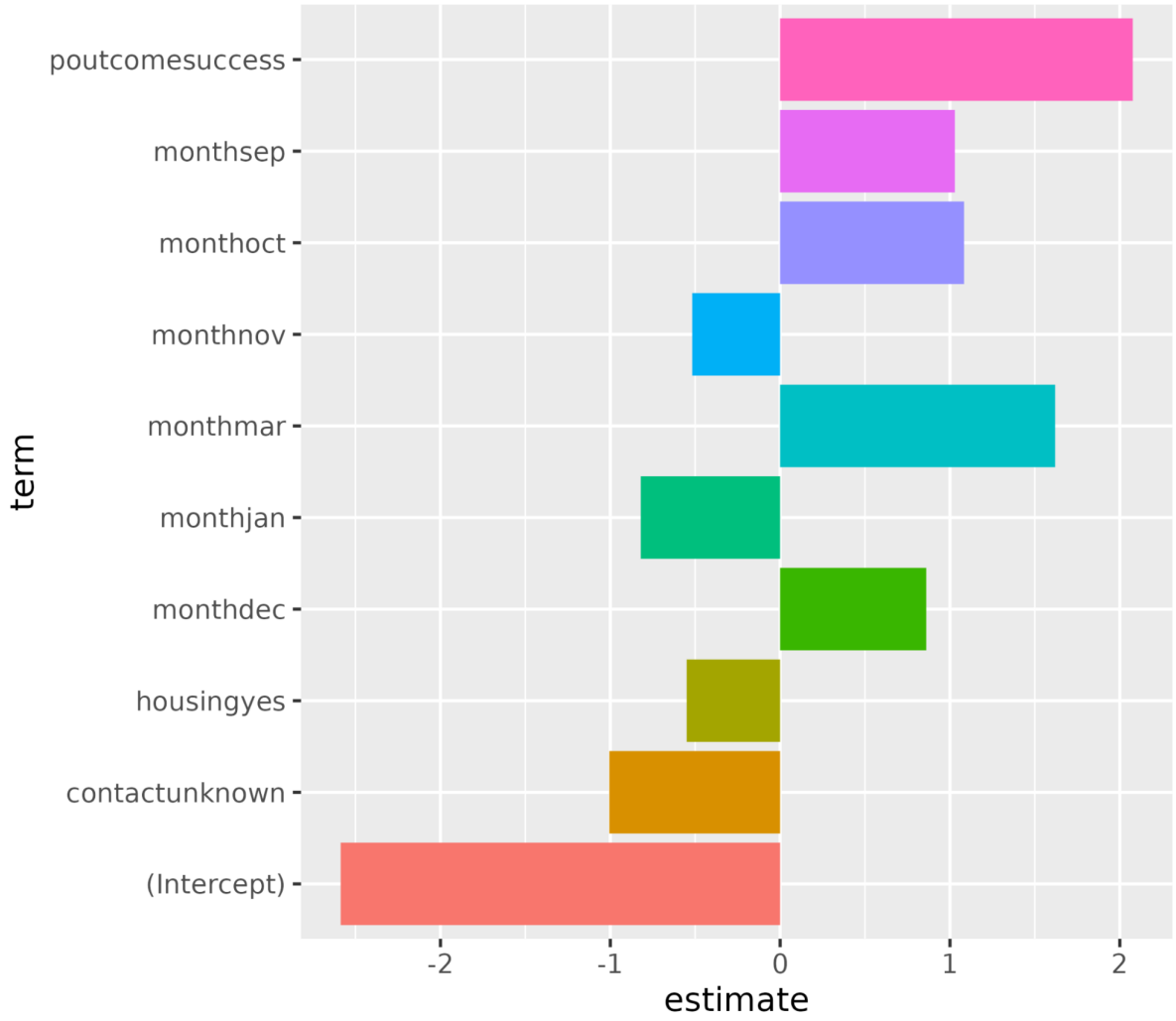
Let's understand the variables impacting the subscription buying decision. In a logistic regression scenario, the coefficients decide how sensitive the target variable is to the individual predictors. The higher the value of coefficients the higher their importance is. Sort the variables in descending order of the absolute value of their coefficient values and display only the coefficients with an absolute value greater than 0.5.

```
coeff <- tidy(log_reg_final)
  arrange(desc(abs(estimate)))
  filter(abs(estimate) > 0.5)
```

```
# A tibble: 10 × 3
   term            estimate      penalty
   <chr>              <dbl>        <dbl>
 1 (Intercept)       -2.59  0.0000000001
 2 poutcomesuccess    2.08  0.0000000001
 3 monthmar           1.62  0.0000000001
 4 monthoct           1.08  0.0000000001
 5 monthsep           1.03  0.0000000001
 6 contactunknown    -1.01  0.0000000001
 7 monthdec           0.861 0.0000000001
 8 monthjan          -0.820 0.0000000001
 9 housingyes        -0.550 0.0000000001
10 monthnov          -0.517 0.0000000001
```

Plot the feature importance using the ggplot() function.

```
ggplot(coeff, aes(x = term, y = estimate, fill = term)) + g
eom_col() + coord_flip()
```


poutcomesuccess
monthsep
monthoct
monthnov
monthmar
monthjan
monthdec
housingyes
contactunknown
(Intercept)
term
-2
-1
0
1
2
estimate