Train-Test Split

```
library(caTools)
set.seed(0)

split = sample.split(df, SplitRatio = 0.8)
train = subset(df, split == TRUE)
test = subset(df, split == FALSE)
```

Linear Regression (Default)

```
lm = lm(y ~ x1+ x2.. , data = train)
lm = lm(y ~ . , data =train)
summary(lm)
coef(lm)
```

Notes:
- to include ***Interaction*** between 2 variables, use **x1*x2**
(see Machine Learning with R, 3rd ed, p.195)

Linear Regression (Subset Selection)

```
library(leaps)

lm_best = regsubsets(y ~. , data= train, nvmax= #, method= " )

summary(lm_best)
summary(lm_best)$adjr2
which.max(summary(lm_best)$adjr2)
coef(lm_best, #max)
```

Notes:
- nvmax is max # of models
- creates models with multiple combinations, finds the model with the best R2 and its coefficients

Use a Model to predict y

```
y_pred = predict(lm, test)
```

Notes:
- lm is model name
- lm uses df to predict y
- df is predictor variables data set, can also be df_train or df_test (see test train split)

Logistic Regression

```
lm = glm(y ~. , data=  train, family= "binomial")
summary(lm)
coef(lm)

y_pred = predict(lm, test, type= "response")
y_pred = y_pred > 0.5
```

Notes:
- The 0.5 is an adjustable Cutoff to decide TRUE & FALSE. without it; continuous probabilities between 0 and 1 will be produced
- to change from boolean to 1 & 0 in order to use sum() in confusion matrix to save time, use: **y_pred[(y_pred== TRUE)] = 1**

Confusion Matrix / Crosstabs

```
table(y_pred, y)
sum(y_pred == y) / length(y_pred)
```

Notes:
- used to compare the accuracy of different models in predicting a categorical variable
- y_pred is the predicted categorical outcome using a model we created, y is the original true outcome (df$y)
- the Cutoff in the model can be changed to get better predictive values in the confusion matrix
-2nd line calculates ***Accuracy*** *(TP + TN divided by all)*

Detailed Crosstabs & Chi-Square

```
library(gmodels)
CrossTable(df$x, df$y, chisq = TRUE,  prop.chisq = FALSE)
```

R2, MSE & MAE

```
MSE = mean((y_pred - y) ^ 2)
MAE = mean(abs(y_pred - y))
MAE = function(x,y) {mean(abs(x-y))}
→ MAE(y_pred, test$y)
```

Note:
- used to compare the accuracy of different models in predicting a continuous variable

ROC & AUC

```
library(ROCR)
roc = prediction(x, y)
roc_perf = performance(roc, "sens", "fpr")
plot(roc_perf, colorize=TRUE, print.cutoffs.at = c(#, #) )
```

Notes:
- x is predictor (continuous var) & y is predicted (dichotomous var)
- “sens “& “fpr” is sensitivity and false positive rate
- cutoffs are a list of numbers desired to be plotted on the curve
- cutoffs also by: **seq(0.1, by=0.1)**, for eg, in logistic regression

Linear Discriminant Analysis (LDA)

```
library(MASS)

lda = lda(y ~ x1 + x2, data= train)
lda = lda(y ~. , data= train)

y_pred = predict(lda, test)$class

table(y_pred, y)
```

Notes:
y_pred is predicted clusters/classes

K-Nearest Neighbors (KNN)

```
library(class)

x_train = train[ , - y col #]
x_test = test[ , - y col #]
x_train_z = scale(x_train)
x_test_z = scale(x_test)

y_train = train$y

set.seed(0)

y_pred = knn(x_train_z, x_test_z, y_train, k= # )

table(y_pred, y)
```

Notes:
- first split df into train & test
- y_pred is predicted y classes, from x_test_z
- the **scale()** function standardizes x variables to Z-Scores, this has an advantage to min-max method that its resistant to outliers
- we need to standardize/normalize/scale the data first because KNN uses distance as a unit (Euclidean Distance is Mc and default, others: Manhattan & Weighted Voting)
- KNN does Not produce coefficients cuz it is a non-parametric classification method (it classifies the cases in test set directly by plotting them against the x & y cases in train set)
- we set seed cuz if there's a tie; the decision is done randomly
- small k --> low bias & high variance
- large k --> high bias & low variance

Regression / Classification Decision Tree

```
library(rpart)
library(rpart.plot)

dtree = rpart(y~. , data = train, control =
                rpart.control(maxdepth = # ))

rpart.plot(dtree, box.palette = "GnBu", digits = 2)

y_pred = predict(dtree, test)
```

Note:
- maxdepth can be adjusted (reduced) to avoid overfitting
- colors eg. "RdBu", “GnPu"
- for classification tree; add to dree: **,method = "class"**
  And add to y_pred predict: **type = "class"**

C5.0 Decision Tree & Cost-Matrix (Error Penalty)

```
library(C50)

matrix_labels = list(c("0", "1"), c("0", "1"))
names(matrix_labels) = c("y_pred", "actual")
cost_matrix = matrix(c(#, #, #, #), nrow = 2,
                     dimnames = matrix_labels)

dtree = C5.0(train$x, train$y, rules = ,costs = cost_matrix)
y_pred = predict(dtree, test)
```

Note:
- this adjusts for a specified error (i.e. FP or FN), this is by specifying the penalty/cost of an error in the cost matrix, using a number for each group (TP, TN, FP, FN)
- “0” & “1” if the output was in the form of 0 and 1. But if otherwise;the matrix labels should be the same as the output of the algorithm
- The rules parameter: creates rules from decision trees, boolean

1R Classifier

```
library(OneR)

dtree = OneR(df$type~ ., data = df)
y_pred = predict(dtree, test)
```

Note:
- This algorithm choses the single most important feature (variable) and makes its decions based on it

RIPPER Classifier

```
library(RWeka)

dtree = JRip(df$type ~ . , data = test)
y_pred = predict(dtree, test)
```

Cubist Model Tree

```
library(Cubist)

mtree = cubist(train[ ,- y col #], train$y,
               control = cubistControl(rules = #))
y_pred = predict(mtree, test)
```

Note:
- it's a hybrid regression tree that adds a linear regression model at the end of each node
- perfect for relationships involving curvilinear or plateau effects.

Bagging & Random Forest

```
library(randomForest)

dtree = randomForest(y ~., data = train, mtry = # , ntree = #)
y_pred = predict(dtree, test)
```

Notes:
- convert y to factor (categorical var) first if using for classification, using: **df$y= as.factor(dfy)**. only needed in a classification tree, this is done after dummy coding and before splitting. If regression, ignore this line ofc
-mtry is # of predictor variables (p). if bagging, we use them all. if random forest, we use a portion of variables in each tree (usually, p/3 in regression, and $\sqrt{p}$ in classification)
-ntree is # of trees created

Gradient Boost

```
library(gbm)

dtree=  gbm(y ~ . , data= train, distribution = " ,
                n.trees = # , interaction.depth = # )

y_pred = predict(dtree, test, type = "response") > 0.5
```

Notes:
- if regression; distribution = “gaussian”
- if classification; distribution = “bernoulli”
- add type & cutoff if classification tree. if regression tree ignore this portion ofc

AdaBoost

```
library(adabag)

dtree = boosting(y ~ . , data = train, boos = TRUE, mfinal = # )
y_pred = predict(dtree, test)

table(y_pred$class, y)
```

XGBoost

Support Vector Machine (SVM)

```
library(e1071)

svm = svm(y~ . , data = train, kernel = " ,
            cost = #, scale = TRUE)
y_pred = predict(svm, test)
table(y_pred, y)
```

Notes:
- convert y to factor (categorical var) first if using SVM for classification using: df$y= as.factor(dfy)

SVM-Tuning

```
library(e1071)
tuned_svms = tune(svm, y~. ,data = train,
                  kernel = "polynomial",
                  ranges = list(cost=c(0.1, 1, 10),
                  degree =c(1,3,4)))

best_svm = tuned_svms$best.model
best_svm

y_pred = predict(best_svm, test)
table(y_pred, y)
```

Notes:
- tunes the SVM model by trying all hyperparameter combinations
- SVM can be linear, polynomial or radial. If linear; no degree needed. If radial; gamma should also be added and specified
- if normal SVM code is written before this using the name svm; an error will occur, to solve this simple change the 1st svm model name or clear the saved variable names