Glossary of relevant R functions

Creating training/validation/test splits

- `sample(vector/number, size)` – Samples a certain amount of numbers from a range/vector
  - Base R
- `sample_frac(proportion)` – samples a certain amount of a dataset
  - `dplyr` package
  - Can combine with `anti_join` from `dplyr` package to create opposite of dataset
- `sample.split(target_variable, split_ratio)`
  - `caTools` package
  - Creates a list of row indexes, while preserving the ratio of labels for the target variable
- `createDataPartition(target_variable, number_of_partitions, training_proportion)`
  - `caret` package
  - Creates training/test partitions with similar distributions of the target variable y.

EDA functions

- `hist(data, breaks)`
  - Plots a histogram of a vector of data
  - Breaks argument allows you to specify the number of breaks/bins to use
- `par(mfrow = c(a,b))`
  - Specifies plotting display in `R`
  - Will display a grid of plots `a` rows by `b` columns

- `pairs(data)`
  - Plots a matrix of scatterplots
  - Categorical and logical variables converted to numeric factors similar to `data.matrix()`

See also: `ggplot2` introduction and quick examples

- 3 Data visualisation | R for Data Science (had.co.nz)
- Histograms and frequency polygons — geom_freqpoly • ggplot2 (tidyverse.org)
- Points — geom_point • ggplot2 (tidyverse.org)

Linear models and generalised linear models

- `lm(target_variable ~ predictors, data, subset, offset)`
  - Fits a simple linear regression using the specified predictors on the target variable
  - Offset specifies if you would like to include any variables with known slope – such as using population to scale the predicted value on a proportional basis to population
  - Subset allows you to specify indexes to use to train the data – can use row indexes rather than manually subset data
  - Can call `plot(model_object)` to plot diagnostic plots
  - Can call `summary(model_object)` to display summary table of coefficients and p-values

- `glm(target_variable ~ predictors, family, data, offset, subset)`
  - Fits a glm model using a specified distributional family
  - Can specify custom link function to use instead of canonical link function – see documentation
  - Can call `plot(glm_object)` to plot diagnostic plots
  - Can call `summary(glm_object)` to display summary table of coefficients and p-values

Fitting a k-nearest neighbours model

Using class package: i.e. first run `install.packages("class")` and `library(class)`.

- `knn(train, test, cl, k, prob)`
  - `train` specifies training dataset to use for KNN
  - `test` specifies test dataset to predict using KNN model
  - `cl` is a vector of the true classification labels

  - `K` specifies the number of nearest neighbours
  - Outputs a list of predicted labels using the KNN model
  - Can use `prob = TRUE` argument to instead output probabilities calculated using KNN

Subset selection: Best, forward and backward

Using `leaps` package: i.e. first run `install.packages("leaps")` and `library(leaps)`.

- `regsubsets(Y_var ~ predictors, Data, method)`
  - Can perform stepwise, forward and backward selection by setting `method` to "forward", "stepwise" or "backward"
  - `summary(regsubsets_object)` returns a list of variables used for each model size
  - Summary object has sub-objects such as Mallow's C_p, BIC and Adjusted R^2
  - Can use `coef(regsubsets_object, num_variables)` to extract coefficients for a given model size

LOOCV and k-fold CV on GLM models

Using boot package: i.e. first run `install.packages("boot")` and `library(boot)`.

- `cv.glm(Data, glm_model_object, K)` – performs cross-validation using the fitted glm object and data
  - By default, performs LOOCV CV but can use `K` argument to specify number of folds, and then perform k-fold CV.
  - Access cross-validation errors using `$delta` on `cv.glm` object. Returns two values – one is the raw cross-validated error and the other is the bias corrected version for not using LOOCV
  - See documentation here: cv.glm function - RDocumentation
  - This approach only works for GLM objects

Alternative approach: Manually creating folds using `caret` package

- `createFolds(target_variable, k)`
  - Creates `k` number of folds, by default returned in a matrix, with roughly equal distribution of the target distribution in each fold
  - Could use these folds along with a loop to create cross-validated errors manually

Fitting ridge regression and lasso regression models

Using glmnet package: i.e. first run `install.packages("glmnet")` and `library(glmnet)`.

- `model.matrix(target_variable ~ predictors, Data)[, -1]`
  - Creates a model matrix with the predictors and an intercept. Use `[, -1]` to drop the created intercept column
  - Required when using glmnet to fit lasso and ridge regression models
- `glmnet(x_var, y_var, alpha, lambda)`
  - `x_var` is the matrix of predictors created using `model.matrix`
  - `alpha = 0` specifies a ridge regression, `alpha = 1` specifies a lasso regression
  - `lambda` allows you to specify a custom range of lambda values to look across
- `predict(glmnet_model, s, type, newx)`
  - Using predict with a glmnet model object allows you to specify s, the value of lambda
  - `type` = coefficients returns coefficients, otherwise returns predicted values by replacing type with newx argument.
- `cv.glmnet(x_var, y_var, alpha, nfolds = 10)`
  - Fits either a ridge regression or lasso regression based on the value of `alpha`
  - Allows you to extract the lambda that minimises the RSS using `$lambda.min` on the cv.glmnet object.
  - Also simultaneously performs either k-fold or LOOCV using `nfolds` argument (the number of folds)
  - See documentation here: Cross-validation for glmnet — cv.glmnet • glmnet (stanford.edu)

Fitting tree models

Using tree package: i.e. first run `install.packages("tree")` and `library(tree)`.

- `tree(target_variable ~ predictors, data, subset)`
  - Fits a simple decision tree model using specified predictors
  - Can use subset argument, similar to a linear model
  - Can plot a graph of the fitted tree using:
    * `plot(tree_model)`
    * `text(tree_model, pretty = 0)`

Using the rpart & rpart.plot packages: i.e. first run `install.packages(c("rpart", "rpart.plot"))` then `library(rpart)` and `library(rpart.plot)`.

- `rpart(Sales ~., data, subset)`
  - Similar to tree but allows plotting using `rpart.plot` function
- `rpart.plot(rpart_tree_model)`
  - Plots the rpart tree model in a nice plot

Cross validating optimal decision tree size and pruning tree

- `cv.tree(tree_model, k)`
  - Input a fitted tree model into function to perform cross validation
  - Can specify `k`, the number of folds to use for cross validation
  - Can access `cv_tree_object$size`, `cv_tree_object$dev` and `cv_tree_object$k`, for vectors of the size, corresponding deviance and value of alpha (the cost complexity parameter for pruning), to find optimal cost complexity parameter based on lowest deviance
- `prune.tree(tree_model, best, k)`
  - Creates a new pruned tree based on an already fitted tree model, the specified number of terminal nodes, or alternatively, the cost complexity parameter
  - `best` refers to the number of terminal nodes
  - `k` refers to the cost complexity parameter
  - Only one of `best` or `k` needs to be specified

Fitting bagging and random forest models

Using randomForest package: i.e. first run `install.packages("randomForest")` and `library(randomForest)`.

- `randomForest(target_variable ~ predictors, data, importance, mtry, subset)`
  - Fits either a random forest model or a bagged model based on what is specified for the `mtry` argument
  - `mtry` refers to the number of variables to randomly sample at each split. When fitting a bagged model, `mtry` should equal the number of predictors in the data, while in a random forest model, it can be any value (by default it is $\sqrt{(p)}$ for classification and $p/3$ for regression, where p is the number of predictors)

- `importance(rf_model)`
  - Outputs a list of variable importance for the fitted `rf_model`, based on an averaged MSE across fitted trees and total decrease in node purity
- `varImpPlot(rf_model, sort)`
  - Plots a variable importance plot based on the averaged MSE metric and decrease in node purity metric
  - `sort` specifies whether to sort variables by importance in descending order. By default, is true.

Fitting a gradient boosted model

Using gbm package: i.e. first run `install.packages("gbm")` and `library(gbm)`.

- `gbm(target_variables ~ predictors, distribution, data, n.trees, interaction.depth, shrinkage)`
  - Fits a generalised gradient boosted regression model
  - `distribution` refers to the distribution used for the loss function when performing splits using the GBM model
  - `n.trees` refers to the total number of ensemble trees to fit
  - `interaction.depth` specifies the number of splits in each tree – 1 refers to trees with one split, depth is 2 is typically used to incorporate interaction effects
  - `shrinkage` specifies the learning rate to be used in the gradient boosting algorithm
  - `cv.folds` specifies how many folds to use when performing cross-validation – can use to instruct gbm function to perform cross-validation

Fitting hierarchical clustering

- `hclust(dist(data), method)`
  - Need to wrap data using `dist()` function to create a dissimilarity matrix based on the data
  - Method specifies the linkage method to be used. Can specify complete, average, and single
  - Can use `plot(hclust_object)` to plot dendrogram
- `cutree(hclust_object, k, h)`
  - Cuts a `hclust_object` and returns cluster labels corresponding to each observation
  - Can either specify `k` or `h` to cut the tree

    * `k` refers to the desired number of clusters
    * `h` refers to the height at which to cut the tree

Fitting a kmeans model

- `kmeans(data, centers, nstart)`
  - Performs kmeans clustering on data, using specified number of clusters, specified by centers
  - `nstart` specifies the number of different initial conditions to use to compare. R will run *k*-means on each of iterations and choose the best solution with the lowest within-cluster variance.
  - Can access within-cluster sum of squares using `$tot.withinss` object of the `kmeans_model`
  - Can access final cluster labels output by kmeans algorithm using `$cluster` object of `kmeans_model`

Performing principal components analysis

- `prcomp(data, scale, center)`
  - Performs prinicipal components analysis on the data
  - `scale` specifies whether to scale variables to have standard deviation one
  - `center` specifies whether to shift variables to have mean of 0
  - `$rotation` object of pca_model contains the component loadings on each of the principal components
  - `$x` contains the principal component scores or the coordinates of the predictor variable in each direction of the principal component
  - `$sdev`contains the standard deviation of each principal component – you can square this variable to obtain the variance of each principal component, and hence calculate the total variance explained by each principal component
- `biplot(pr_object, scale = 0)`
  - Plots a biplot based on the pr_object fitted where it plots the datapoints on a scatterplot of the first two principal components