# 40.220 The Analytics Edge Mid Term Revision

This is a document containing a list of functions that would be useful for your mid term/finals. If necessary, to illustrate an example, the data will be loaded from a particular dataset. Ensure that the dataset is stored in a directory called 'Dataset'.

Datasets used:

- wine.csv
- Orings.csv
- Hitters.csv

## Before we jump in...

Loading dataset wine.csv to illustrate some examples because it is simple to understand and play around with

In [ ]:

```
wine<-read.csv("Dataset/wine.csv")
```

In [ ]:

```
str(wine)
```

In [ ]:

```
summary(wine)
```

## 1. General functions

### Matrix generation

Array must be listed column down first, then row

In [ ]:

```
m <- matrix(c(3,4,5,6,7,8), nrow=3, ncol=2)
m
```

### Array generation

Note: the difference between array and matrix is that matrix is max 2-dim but array can be > 2-dim

In [ ]:

```
n <- array(c(3,4,5,6,7,8), c(3,2,1))
n
```

## Dataframe generation

Creates a dataframe where columns can be viewed as attributes and rows viewed as observations

In [ ]:

```
t <- data.frame(names=c("karthik","sam","jim"),
                ages = c(36,34,40),
                sex = c("M","F","M"),
                children=c(2,0,1))
t
```

Call t$names to list out all the names

In [ ]:

```
t$names
names(t)
```

## Counting a particular column and table generation

Tabulate and sum up over the chosen attribute. Especially useful for COUNTING number of observations in an attribute (e.g. how many times is the temperature between 14-15 degree celcius)

In [ ]:

```
# rounded down to create a few categories

table(floor(wine$DEGREES))
```

Can further create a dataframe from a table, which might be useful

In [ ]:

```
DegreesDF<-as.data.frame(table(wine$DEGREES_ROUNDED))
```

## Obtaining the size/dimension

In [ ]:

```
dim(wine) # applies for dataframe/matrix
length(wine$VINT) # applies for a particular column/array
```

## Defining new columns in a dataframe

You can create a new column in a dataframe by simply naming a new variable on the left of '<-' and writing the expression on the right of '<-'

In [ ]:

```
wine$LPRICEx2 <- wine$LPRICE * 2
```

## Dealing with missing entries

We can find missing entries in LPRICE by returning a boolean array that checks if there is a missing entry

In [ ]:

```
is.na(wine$LPRICE)
```

We can remove the whole observation/row in the dataframe

In [ ]:

```
wine_narm<-wine[!is.na(wine$LPRICE),]
```

This method works as well

In [ ]:

```
wine_narm2<-na.omit(wine)
```

Removes missing entries when applying mean() function. na.rm argument exists for some functions

In [ ]:

```
mean(wine$LPRICE)
mean(wine$LPRICE, na.rm=TRUE)
```

## Choosing specific rows

We can use the splicing operator [] to choose specific rows

In [ ]:

```
# using row index
wine[2,] # to choose row 2
wine[-2,] # to choose every row except row 2
# using a condition
wine[wine$DEGREES>16,]
```

We can also use the subset() function to choose specific rows

subset(wine, wine$DEGREES>16)

## Choosing specific columns

We can use the splicing operator [] to choose specific columns as well

In [ ]:

```
wine[,c("WRAIN", "DEGREES")]
```

We can also use the subset() function to choose specific columns

Note: subset() function is used to segregate data based on certain condition

In [ ]:

```
subset(wine, select=c(WRAIN, DEGREES))
```

We can use subset() function to remove unwanted columns

In [ ]:

```
subset(wine, select=-c(WRAIN,DEGREES))
```

## which() function

which() function returns an index or array of indexes, for which the condition is TRUE. Note: in comparison to which.max() which only returns ONE index

In [ ]:

```
x<-c(1,2,3,6,3,6)
which(x==max(x))
```

Useful for checking indexes that have missing entries

In [ ]:

```
which(is.na(wine$LPRICE))
wine[which(is.na(wine$LPRICE)),]
```

Useful for finding indexes of highest/lowest values

In [ ]:

```
max(wine$HRAIN)
which(wine$HRAIN==max(wine$HRAIN)) # alternatively, look at which.max()
wine[17,]
```

## which.___ function

which.max() function returns the index of the maximum value

In [ ]:

```
which.max(wine$HRAIN)
```

## Casting to a new type

We can cast a variable to the desired type

In [ ]:

```
as.logical(1)
as.numeric(TRUE)
```

This is especially important when the data is a factor and you need to change it into numeric type. If you convert it into numeric directly from factor, R will convert it into integers based on the level of the factor

In [ ]:

```
wrain_factor<-as.factor(wine$WRAIN)
#wrain_factor

# Wrong
as.numeric(wrain_factor)

# Correct
as.numeric(as.character(wrain_factor))
```

## Family of apply functions

TAPPLY()

> R documentation: Apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.

Idea: Given tapply(arg1, arg2, func), we are applying the function over arg1 grouped according to arg2. (If you know Python Pandas, this is similar to GroupBy)

In the following example, we can find the mean LPRICE based on the each category of DEGREES

In [ ]:

```
# remove na data for tapply
wine_tapply<-na.omit(wine)
# floor it to create a few categories
tapply(wine_tapply$LPRICE, floor(wine_tapply$DEGREES), mean)
```

You can also create a table to tabulate the results

In [ ]:

```
table(tapply(wine_tapply$LPRICE, floor(wine_tapply$DEGREES), mean))
# not meaningful due to the data..
```

LAPPLY()/SAPPLY()

> R documentation: lapply returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X. sapply is a user-friendly version and wrapper of lapply by default returning a vector, matrix or, if simplify = "array", an array if appropriate, by applying simplify2array(). sapply(x, f, simplify = FALSE, USE.NAMES = FALSE) is the same as lapply(x, f).

Idea: Given sapply(arg1, func), we are applying the function over each element in arg1

In the following example, we can create a function to add 1 and apply it on an array

Note: sapply returns a vector or a matrix while lapply returns a list

In [ ]:

```
addOne<-function(x){
    return (x+1)
}
```

In [ ]:

```
myarray<-c(1,2,3)
sapply(myarray, addOne)
```

APPLY()

> Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

Idea: Given apply(arg1, margin, func), we are applying the function across the row if margin==1 and across column if margin==2. This is a neat function if you want to find out the average values across observation of a column for a dataset.

In the following example, we can determine the average WRAIN and DEGREES in the wine dataset

In [ ]:

```
#?apply
#apply(wine[,c('WRAIN')],2,mean) # x must be an array
apply(wine[,c('WRAIN','DEGREES')],2,mean)
```

## Splitting data between train and test

In [ ]:

```
library(caTools)
```

Method 1: using sample.split() with a balanced ratio of dependent variable

In [ ]:

```
# remove na data
wine_narm<-na.omit(wine)
# create categories for LPRICE
wine_narm$LPRICE_CAT <- wine_narm$LPRICE>-1.0
table(wine_narm$LPRICE_CAT)
```

In [ ]:

```
spl <- sample.split(wine_narm, 0.75)
train <- subset(wine_narm, spl==TRUE)
test <- subset(wine_narm, spl==FALSE)
table(train$LPRICE_CAT)
```

In [ ]:

```
table(test$LPRICE_CAT)
```

Metod 2: using sample

In [ ]:

```
trainID<-sample(1:nrow(wine_narm),nrow(wine_narm)/2) # sample(start_index:end_in
dex, number_of_rows_to_split)
train<-wine_narm[trainID,]
test<-wine_narm[-trainID,]
str(train)
```

In [ ]:

```
str(test)
```

## Statistical t-test

Performs a statistical one sample t-test to test the hypothesis if mean value=0.

In [ ]:

```
t.test(wine$DEGREES)
```

Performs a statistical two sample t-test to test if the sample mean of the two samples are sufficiently different. In this case, we are determining whether the sample mean of DEGREES of VINT < 1973 and sample mean of DEGREES of VINT >= 1973 is significantly different. Clearly, the null hypothesis should not be rejected, which means that the sample means are not significantly different

In [ ]:

```
t.test(subset(wine,wine$VINT<1973)$DEGREES, subset(wine,wine$VINT>=1973)$DEGREES
)
```

## Other (potentially) useful functions

- Finding min/max in a column

In [ ]:

```
min(wine$DEGREES)
max(wine$DEGREES)
```

- Sorting values

In [ ]:

```
wine$WRAIN
sort(wine$WRAIN)
```

- Correlation and removing NA values

In [ ]:

```
cor(wine$WRAIN, wine$LPRICE, use="pairwise.complete.obs")
```

- State a specific ordering for factors, instead of based on alphabetical order

In [ ]:

```
df <- data.frame(days=c("monday","tuesday","wednesday","friday","monday"),
                 customers = c(36,34,40,100,2))
str(df)
df
```

In [ ]:

```
tapply(df$customers, df$days, sum)
```

In [ ]:

```
df$days<-factor(df$days, ordered=TRUE, levels=c("monday", "tuesday", "wednesday"
, "thursday", "friday"))
df
```

In [ ]:

```
tapply(df$customers, df$days, sum)
```

- List attributes of objects

In [ ]:

```
names(wine)
wine$LPRICE
```

# 2. Graph Plotting

Loading dataset in-built faithful from R for graph plotting

In [ ]:

```
data(faithful)
str(faithful)
```

### Plotting a scatter plot

In [ ]:

```
plot(faithful)
```

Drawing lines between the plots

In [ ]:

```
plot(faithful, type="l")
```

### Plotting of Histogram

X axis starts from 1.6 and ends at 5.2 with 0.2 breaks

In [ ]:

```
hist(faithful$eruptions, breaks=seq(1.6,5.2,0.2))
```

### Plotting of the empirical culmulative distribution function

In [ ]:

```
plot.ecdf(faithful$eruptions)
```

### Plotting of Q-Q plot to generate the normal quantile plots

In [ ]:

```
qqnorm(faithful$eruptions)
```

## 3. Linear Regression

Splitting the dataset into train and test

In [ ]:

```
library(caTools)
set.seed(1)
spl<-sample.split(wine, SplitRatio = 0.75)
train<-subset(wine, spl==TRUE)
test<-subset(wine, spl==FALSE)
str(train)
```

### Training

Training a linear regression model with LPRICE as the output and the other variables (i.e. VINT, WRAIN, DEGREES, HRAIN, TIME-SV) as predictors

In [ ]:

```
lm_model <- lm(LPRICE~., data=train)
summary(lm_model)
```

**Linear regression formula**:

$$predicted\ LRPICE = 38.9 - 0.0253VINT + 0.000603WRAIN + 0.578DEGREES - 0.00396HRAIN$$

Notes on summary of linear regression model:

Predictor-specific

- **Estimate**: the beta values in the linear model
- **Std. Error**: varability in the beta value
- **t value**: the variable used for hypothesis testing for each predictor. The higher the absolute of t value, the smaller the p value, hence the more significant the variable
- **Pr(>|t|)**: the p value of the hypothesis testing for each predictor. Hypothesis test will test whether the null hypothesis (beta=0) should be rejected. If the p value is small (typically $<0.05$), we can reject the null hypothesis that beta=0 and claim that beta is significant in explaining the model

Overall model

(note: n=number of observations, p=number of predictor variables)

- **Residual standard error**: is a measure of the lack of fit of the model. The smaller the RSE, the better

$$RSE = \sqrt{\frac{SSE}{n-p-1}}$$

- **Multiple R-squared**: normal $R^2$
- **Adjusted R-squared**: takes into consideration of the complexity of the model (i.e. number of predictors in the model)

$$Adjusted\ R^2 = 1 - (1 - R^2)(\frac{n-1}{n-p-1})$$

- **F-statistic**: the variable used for hypothesis testing for the whole model
- **p-value**: the p value of hypothesis testing for the whole model. A low p-value means that the overall model is significant in explaining the data

Other kinds of ways to use lm() for linear regression

In [ ]:

```
lm_model2 <- lm(LPRICE~VINT+WRAIN, data=train) # train with VINT and WRAIN as pr
edictor
lm_model3 <- lm(LPRICE~.-VINT, data=train) # remove one predictor using '-' befo
re the predictor
lm_model4 <- lm(LPRICE~.-1, data=train) # remove the intercept using '-1'
```

## Prediction

Make predictions using the linear regression model on the test data

In [ ]:

```
predict(lm_model, newdata=test)
# ignore the warning message. it is due to empty beta coefficient for TIME_SV
```

If it is a single-variable model, we can plot the datapoints and the best fit graph from the model to see how well the model fits

In [ ]:

```
lm_model5 <- lm(LPRICE~DEGREES, data=train)
plot(train$DEGREES, train$LPRICE)
abline(lm_model5)
```

## Sum of squared errors (SSE)

SSE represents the variation not accounted for by the model. Remember this: (true - predicted)

$$SSE = \sum_{i=1}^{n} (true - predicted)^2$$

In [ ]:

```
# Method 1 (only on train data since it can only be obtained from the model)
sse<-sum(lm_model$residuals^2)
```

In [ ]:

```
# Method 2 (for both train and test data)
sse<-sum((train$LPRICE-predict(lm_model, newdata=train))^2, na.rm = TRUE)
sse
# again, ignore the warning message
```

Note that here we are finding the SSE of the train dataset. We can find the SSE of the test dataset as well by switching the arguments

## Sum of total squared (SST)

SST represents the total variation in the dataset. Remember this: (true - mean)

$$SSE = \sum_{i=1}^{n} (true - mean)^2$$

In [ ]:

```
sst<-sum((train$LPRICE-mean(train$LPRICE, na.rm=TRUE))^2, na.rm = TRUE)
sst
```

Note: when evaluating SST on the test data, we use the mean of the train data, instead of the test data. This is because the model is meant to fit the train dataset, hence the beta0, the intercept of the model is the mean of the train dataset. If we were to use the mean of the test dataset and the mean of the test dataset varies greatly from the train dataset, the model would perform poorly on the $r^2$

### $R^2$

$R^2$ represents the amount of variation in the dataset that is accounted for by the model. Essentially, it is a measure of model fit to the data. Hence, the formula:

$$R^2 = \frac{SST - SSE}{SST} = 1 - \frac{SSE}{SST}$$

In [ ]:

```
1-sse/sst
```

Note: In simple linear regression, (cor(output,predictor))$^2$ = $r^2$. Proof is in the notes. You can verify this using R

## 4. Logistic Regression

We use a different dataset for logistic regression: orings.csv

In [ ]:

```
orings<-read.csv("Dataset/Orings.csv")
```

In [ ]:

```
str(orings)
```

In [ ]:

```
summary(orings)
```

### Training

In [ ]:

```
glm_model<-glm(Field~Temp+Pres,data=orings,family=binomial)
summary(glm_model)
```

**Logistic regression formula**: \begin{equation} *P(Field=1) = \frac{e^{3.96 - 0.119Temp + 0.00869Pres}}{1+e^{3.96 - 0.119Temp + 0.00869Pres}}\ P(Field=0) = 1-P(Field=1)* \end{equation}

Notes on summary of logistic regression model:

Predictor-specific

- **Estimate**: the beta values in the logistic model
- **Std. Error**: varability in the beta value
- **z value**: the variable used for hypothesis testing for each predictor. The higher the absolute of z value, the smaller the p value, hence the more significant the variable
- **Pr(>|z|)**: the p value of the hypothesis testing for each predictor. Hypothesis test will test whether the null hypothesis (beta=0) should be rejected. If the p value is small (typically $<0.05$), we can reject the null hypothesis that beta=0 and claim that beta is significant in explaining the model

Overall model

- **Null deviance**: refers to how well the model fits if given the intercept as the only predictor
$$Null\ deviance = -2LL(only\ intercept)$$
- **Residual deviance**: refers to how well the model fits if given the other predictors as well. A significant decrease from null to residual deviance indicates that the predictors are useful
$$Residual\ deviance = -2LL(\hat{\beta})$$
- **AIC**: refers to deviance but penalises model complexity
  - accounts for both i) fit to data and ii) model complexity (similar to adjusted $R^2$)
  - the smaller the AIC, the better the model fit
  - AIC does not have a range, unlike $R^2$
  - AIC formula (note: # parameters include intercept):
$$AIC = -2LL(\hat{\beta}) + 2(\#\ parameters)$$

## Prediction

Predicting the log odds \begin{equation} *Log\ odds = log(\frac{P(Field=1)}{P(Field=0)}) = 3.96 - 0.119Temp + 0.00869Pres* \end{equation}

In [ ]:

```
predict(glm_model, newdata=orings) # this is logodds
```

Note: \begin{equation} *Odds = \frac{P(Field=1)}{P(Field=0)} = e^{3.96 - 0.119Temp + 0.00869Pres}* \end{equation}

Predicting the probability

$$P(Field = 1) = \frac{e^{3.96-0.119Temp+0.00869Pres}}{1 + e^{3.96-0.119Temp+0.00869Pres}}$$

In [ ]:

```
glm_predict<-predict(glm_model, newdata=orings,type="response") # predict probab
ilities and store in variable
glm_predict
```

Obtaining the confusion matrix with a threshold of 0.1

In [ ]:

```
table(glm_predict[1:138]>0.1, orings$Field[1:138])
```

True positive: 6, True negative: 110, False positive: 18, False negative 4

$$Accuracy = \frac{110 + 6}{110 + 6 + 18 + 4} = 0.84$$

We always want to compare our model against the baseline model, which is a model that predicts using the majority label across all test observation

In [ ]:

```
table(orings$Field)
```

Given that the majority of output is 0, the baseline model would predict 0 for all observation. In this case, we find that the baseline model performs better than the logistic regression model in terms of accuracy. However, in this domain of space travel, we are concerned about the false negative (i.e. predicting that there is no problem, when in fact there is a problem with the orings). Hence, we need to lower the threshold to reduce the false negatives.

$$Accuracy = \frac{128}{128 + 10} = 0.92$$

## ROC Curve and AUC

In [ ]:

```
library(ROCR)
ROCRpred <- prediction(glm_predict[1:138], orings$Field[1:138]) # prediction(pre
dicted_values, actual_values)
```

Plotting the ROC curve

In [ ]:

```
ROCRperf <- performance(ROCRpred, measure='tpr', x.measure='fpr')
plot(ROCRperf)
```

Obtaining the AUC value

In [ ]:

```
performance(ROCRpred, measure='auc')
```

# 5. Subset Selection

We use a different dataset for subset selection: Hitters.csv

In [ ]:

```
hitters<-read.csv("Dataset/Hitters.csv")
hitters<-na.omit(hitters) # to remove na values
```

In [ ]:

```
str(hitters)
```

In [ ]:

```
summary(hitters)
```

In [ ]:

```
library(leaps)
```

## Training

Using regression, we can select the best model at each size of subset (i.e. a subset with 1/2/3... features)

Parameters of regsubsets:

- **nvmax**: max size of feature subsets to examine (default=8)
- **method**: search method (default=exhaustive search)

In [ ]:

```
hitters<-hitters[,2:21] # to remove first column of names
regsubset_model1<-regsubsets(Salary~., data=hitters)
regsubset_model2<-regsubsets(Salary~., data=hitters, nvmax=19)
regsubset_model3<-regsubsets(Salary~., data=hitters, nvmax=19, method='forward')
```

In [ ]:

```
summary(regsubset_model1)
```

## Results of model

In [ ]:

```
summary(regsubset_model2)
```

Obtaining the i) $R^2$, ii) residual sum of squared and iii) adjusted $R^2$ value for various subset sizes

In [ ]:

```
summary(regsubset_model2)$rsq
summary(regsubset_model2)$rss
summary(regsubset_model2)$adjr2
```

Plotting the adjusted $R^2$. You can do the same with the other 2 measures

In [ ]:

```
plot(summary(regsubset_model2)$adjr2)
```

Finding the subset of features with the highest adjusted $R^2$ and to obtain the list of features and its beta coefficients of a particular subset

In [ ]:

```
which.max(summary(regsubset_model2)$adjr2)
coef(regsubset_model2, 11)
```

# 6. Lasso

Motivation: Lasso is a regression method that penalizes the absolute size of beta coefficients and may force some of coefficient to zero, which removes the feature from the model. This allows for automatic feature selection during the training of model, as opposed to subset selection

In [ ]:

```
library(glmnet)
```

## Data preparation

To run glmnet(), we need to pass i) the input matrix and ii) the vector output. This is abit different from the typical y~x format that we are used to. Furthermore, glmnet works only with quantitative variables, hence it will expand 'factor' variables to dummy variables

In [ ]:

```
x<-model.matrix(Salary~., data=hitters)
y<-hitters$Salary
```

Splitting the dataset into train and test

In [ ]:

```
set.seed(1) # always set seed the line before
train<-sample(1:nrow(x),nrow(x)/2) # if unclear about sample(), refer to above
test<--train
```

## Training

For each lambda value, glmnet will find the LASSO model with the least error using the following objective function

$$\min_{\beta_0,\beta_1,\ldots,\beta_p} \sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_1 - \ldots - \beta_p x_p)^2 + \lambda \sum_{j=1}^{p} \beta_j$$

In [ ]:

```
# generate a range of lambda values for LASSO model
grid<-10^seq(10,-2,length=10)
lasso_model<-glmnet(x[train,],y[train],lambda=grid)
```

## Results of model

Obtaining the i) overview of the model, ii) number of non-zero coefficients for each lambda value, iii) coefficient of beta for each lambda value in a matrix, iv) coefficient of beta for a SINGLE lambda value

In [ ]:

```
lasso_model # check names(lasso_model) if unsure
lasso_model$df
lasso_model$beta #or coef(lasso_model) to see the intercept beta0
coef(lasso_model, s = 1.000e+02)
```

Plotting the beta coefficient values for each lambda. The values of coefficient tends to zero as lambda increases. As illustrated in the lasso equation, as lambda increases, the term $\lambda \sum_{j=1}^{p} \beta_j$ becomes more important. Since this is a minimization problem, beta naturally will tend to zero

In [ ]:

```
plot(lasso_model, xvar = 'lambda')
```

## Prediction

Predict values using LASSO model

Recall the grid values used

In [ ]:

```
grid
```

- Default: all lambda values used in the grid

In [ ]:

```
predict(lasso_model, newx=x[test,])
```

- Using a specific lambda value of the grid

In [ ]:

```
predict(lasso_model, newx=x[test,], s=100)
```

- Using a specific lambda value within the range of the grid but not a value of the grid (linear interpolation). If you used a lambda value outside of the range of the grid, it will still be linearly extrapolataed

In [ ]:

```
predict(lasso_model, newx=x[test,], s=1.25e+02)
```

- Using a specific value of the grid without linear interpolation. You need to refit the model with the train dataset

In [ ]:

```
predict(lasso_model, newx=x[test,], s=1.25e+02, exact=T, x=x[train,], y=y[train
])
```

We can calculate the mean squared error of the test set

In [ ]:

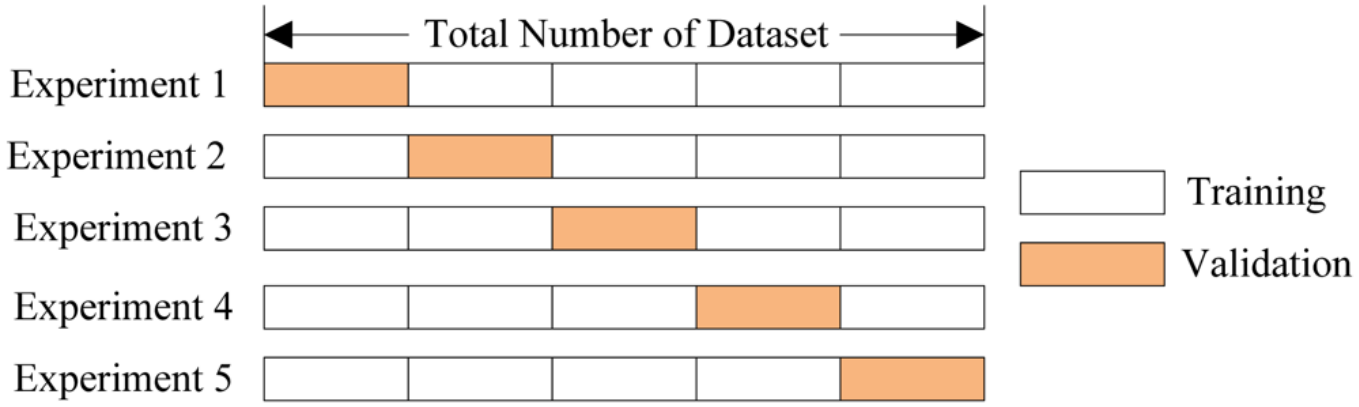
```
predicted <- predict(lasso_model, newx=x[test,], s=100)
mean((predicted-y[test])^2)
```

# 7. Cross validation for Lasso

Motivation: while we have been able to generate a list of fitted model with a range of lambda values, we are still unsure which lambda value to choose. Granted, we can derive the mean squared error for our prediction, but thats only for one instance of the test set. We use a more robust method to help us decide: cross validation

## Training and evaluating

Performs k-fold cross validation for glmnet, where k=10 by default. An illustration of k-fold with k=5. Note that the glmnet will run once to choose its own lambda sequence, hence it will run (k+1) times




In [ ]:

```
set.seed(1)
cvlasso_model<-cv.glmnet(x[train,],y[train])
```

## Results of model

Obtaining the i) value of lambda that gives the lowest mean cross-validated error, ii) lambda values used, iii) mean cross-validated error for each lambda value, iv) number of non-zero coefficients for each lambda value

In [ ]:

```
cvlasso_model$lambda.min
cvlasso_model$lambda
cvlasso_model$cvm
cvlasso_model$nzero
```

## Prediction

Using the optimal lambda value, we can predict values using original LASSO model in 6. Lasso

- Using a specific lambda value of the grid without linear interpolation

In [ ]:

```
predict(lasso_model, newx=x[test,], s=16.7801585216616, exact=T, x=x[train,], y=
y[train])
```

Obtain the beta coefficients for a selected lambda value

In [ ]:

```
coef(lasso_model, s = 16.7801585216616)
```

In [ ]:

```

```