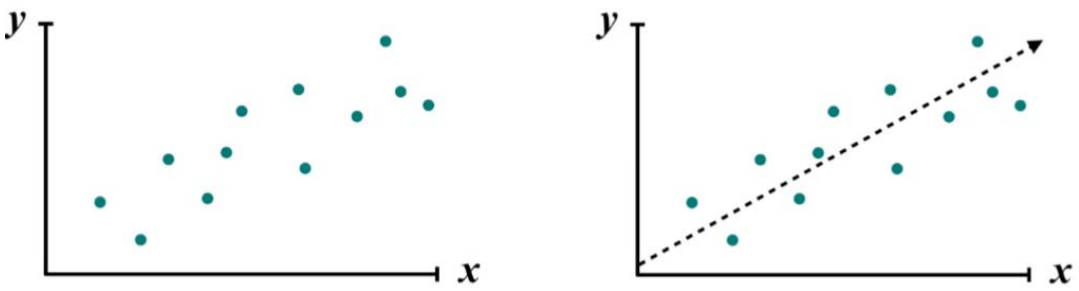
LOGISTIC REGRESSION

MAKING BINARY PREDICTIONS WITH REGRESSION

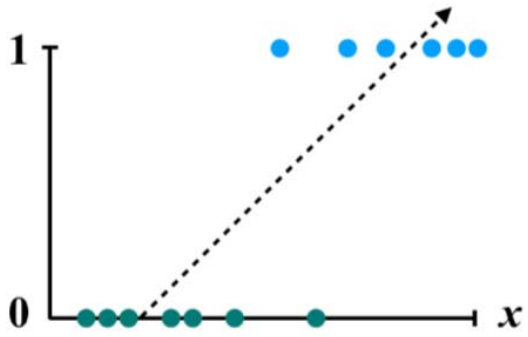
So imagine the classical example of a linear regression:



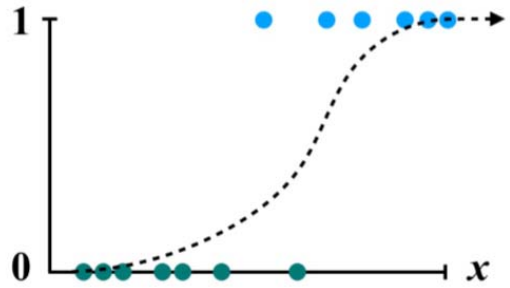
This works great for data that is continuous on both x and y

But what if you have data that is continuous on x and discrete on y (e.g. education level and employment status or fitness level and survival status):

You could fit a linear model to it, but you will get some predictions back like 0.5, which can be difficult to fit into binomial categories like 1/0 or alive/dead



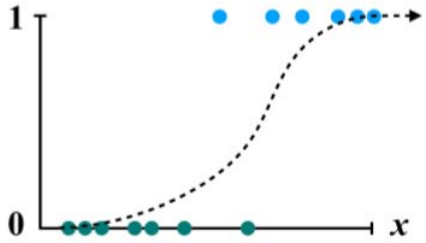
This is where the idea of a logistic regression comes in:



Although you still get values like 0.5, they will be significantly less frequent, and this type of a model conforms better to the observations which can only be 1/0 or alive/dead or whatever.

In this case values like 0.5 or 0.8 or 0.2 would correlate to the probability of that feature being exhibited according to the model

A logistic regression can be modelled in R thusly:




```
m <- glm(y ~ x1 + x2 + x3,
         data = my_dataset,
         family = "binomial")
```

The family parameter specifies the type of model being built because the `glm()` function can be used to do many different types of regression in this case `family = "binomial"` tells R to perform logistic regression.

Once the model has been constructed it can be used to provide probabilities with the `predict()` function, predicted probabilities are easier to interpret than the default 'log-odds' values, to have these returned by the predict function, supply the function with the `type = "response"` parameter.

LOGIT AND PROBIT

These slightly affect how the linear function is fed into the sigmoid function, basically just use logit unless you know better, in which case you would know whether to bother using probit (which is usually for economics or political science).[1]

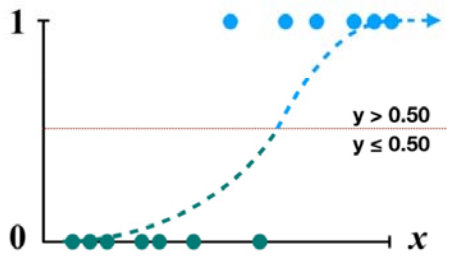
m <- glm(y ~ x1 + x2 + x3, data = my_dataset, family = binomial(link = "logit"))

[1] https://www.methodsconsultants.com/tutorial/what-is-the-difference-between-logit-and-probit-models

Although it will be necessary to use a decision boundary (i.e. an if else statement) in order to predict or classify data:

,

The probability can be converted into the classifications of interest using an `ifelse()` step, where to set the threshold may depend on how aggressive you want the model to be.




```
prob <- predict(m, test_dataset,
                type = "response")
```

```
pred <- ifelse(prob > 0.50, 1, 0)
```

PRACTICAL EXAMPLE

The `donors` dataset contains 93,462 examples of people mailed in a fundraising solicitation for paralyzed military veterans. The `donated` column is `1` if the person made a donation in response to the mailing and `0` otherwise. This binary outcome will be the DEPENDENT variable for the logistic regression model.

The remaining columns are features of the prospective donors that may influence their donation behavior. These are the model's INDEPENDENT VARIABLES.

When building a regression model, it it often helpful to form a hypothesis about which independent variables will be predictive of the dependent variable. `The bad_address`column, which is set to `1` for an invalid mailing address and `0` otherwise, seems like it might reduce the chances of a donation. Similarly, one might suspect that religious interest (`interest_religion`) and interest in veterans affairs (`interest_veterans`)

INSPECT THE DATASET

```
donated veteran bad_address age has_children wealth_rating interest_veterans
interest_religion
1       0       0           0  60           0             0                 0
0
2       0       0           0  46           1             3                 0
0
3       0       0           0  NA           0             1                 0
0
4       0       0           0  70           0             2                 0
0
5       0       0           0  78           1             1                 0
1
6       0       0           0  NA           0             0                 0
0
  pet_owner catalog_shopper recency  frequency  money
1         0               0 CURRENT   FREQUENT MEDIUM
2         0               0 CURRENT   FREQUENT   HIGH
3         0               0 CURRENT   FREQUENT MEDIUM
4         0               0 CURRENT   FREQUENT MEDIUM
5         0               1 CURRENT   FREQUENT MEDIUM
6       0             0 CURRENT INFREQUENT MEDIUM
```

Examine the structure of the data set:

```
> str(donors)
'data.frame':  93462 obs. of  13 variables:
 $ donated          : int  0 0 0 0 0 0 0 0 0 0 ...
 $ veteran          : int  0 0 0 0 0 0 0 0 0 0 ...
 $ bad_address      : int  0 0 0 0 0 0 0 0 0 0 ...
 $ age              : int  60 46 NA 70 78 NA 38 NA NA 65 ...
 $ has_children     : int  0 1 0 0 1 0 1 0 0 0 ...
 $ wealth_rating    : int  0 3 1 2 1 0 2 3 1 0 ...
 $ interest_veterans: int  0 0 0 0 0 0 0 0 0 0 ...
 $ interest_religion: int  0 0 0 0 1 0 0 0 0 0 ...
 $ pet_owner        : int  0 0 0 0 0 0 1 0 0 0 ...
 $ catalog_shopper  : int  0 0 0 0 1 0 0 0 0 0 ...
 $ recency          : Factor w/ 2 levels "CURRENT","LAPSED": 1 1 1 1 1 1 1 1 1 1 ...
 $ frequency        : Factor w/ 2 levels "FREQUENT","INFREQUENT": 1 1 1 1 1 2 2 1 2 2 ...
 $ money            : Factor w/ 2 levels "HIGH","MEDIUM": 2 1 2 2 2 2 2 2 2 2 ...
```

CREATE A LOGISTIC REGRESSION

We will see whether `interest_veterans, interest_religion` and

`bad_address` are good predictors of a donation occurring

CREATE THE MODEL

```
donation_model <- glm(formula = donated ~ interest_religion +
                        interest_veterans +
                        bad_address,
                      data = donors, family = "binomial")
```

SUMMARISE THE MODEL

```
# Summarise the model ------------------------------------------------------

  summary(donation_model)

Call:
glm(formula = donated ~ interest_religion + interest_veterans +
    bad_address, family = "binomial", data = donors)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-0.3480  -0.3192  -0.3192  -0.3192   2.5678

Coefficients:
                  Estimate Std. Error  z value Pr(>|z|)
(Intercept)       -2.95139    0.01652 -178.664   <2e-16 ***
interest_religion  0.06724    0.05069    1.327   0.1847
interest_veterans  0.11009    0.04676    2.354   0.0186 *
bad_address       -0.30780    0.14348   -2.145   0.0319 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 37330  on 93461  degrees of freedom
Residual deviance: 37316  on 93458  degrees of freedom
AIC: 37324

Number of Fisher Scoring iterations: 5
```

APPENDIX: NOTES ON FORMAT, CREATING MODELS AND PREDICT()

NAMING VARIABLES

you shouldn't use _ in naming variables only use period-dots (i.e. “.”) or Capitlisation, this is advised by the G*oogle style guide*[2] and could be because file names tend to have that format and it could be ambiguous, I have retained them here however to match what I've seen on *Data Camp.*

Although, to be fair, the *tidyverse* style guide says the direct opposite so who knows.[3]

[2] https://google.github.io/styleguide/Rguide.xml

[3] http://style.tidyverse.org/syntax.html#assignment

CODE STYLE, FORMATTING, NAMING AND THE PREDICT FUNCTION

You must use this syntax format:

```
model.glm <- glm(coumn.nameY ~ column.name.X1 + column.name.X2 …, data = data.frame)
newdata   <- data.frame(column.name.X1 = c(1,2,3), column.name.X2 = c(1,2,3))
predict(model.glm, newdata)
```

because otherwise the predict function can get really pissy to use, this is the correct way to do it, hence why it is similar to the general format used by *tidyverse* programs.

You should not use `""` in the `newdata` assignment because if you NEED to use them, you have incompatible names.

COLUMN NAMES

Now this format has the unsavoury ramification that data frames MUST ONLY have syntactically valid names[4] (i.e. no spaces or funny characters).

Generally however this is an advisable practice for code stability anyway, to the extent that:

- Most packages have functionality to add labels later
    - E.g. `ggplot2` has the `+labs()` parameter
- R has a built in package to create syntactically valid names
    - `make.names(character.vector, unique = TRUE)`
- Some `tidyverse` packages may/may-not convert names behind the scenes to a syntactically valid name, this is undesirable because you can't really see it happening

[4] A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as `".2way"` are not valid, and neither are the reserved words. *(Refer to the make.names help page)*

CREATING SAFE NAMES

Say we did have a dataframe with names that were not *syntactically valid*:

```
names(donors) <- c("Donated", "Veteran", "Bad Adress", "Age", "Has Children",
                   "Wealth Rating", "Interest Veterans", "Interest Religion",
                   "Pet Owner", "Catolog Shopper", "Recency", "Frequency",
                   "Money" )
head(donors)
Donated Veteran Bad Adress Age Has Children Wealth Rating Interest Veterans Interest Religion
1       0       0          0  60           0             0                  0
0
2       0       0          0  46           1             3                  0
0
3       0       0          0  NA           0             1                  0
0
4       0       0          0  70           0             2                  0
0
5       0       0          0  78           1             1                  0
1
6       0       0          0  NA           0             0                  0
0
  Pet Owner Catolog Shopper Recency  Frequency  Money
1         0               0 CURRENT   FREQUENT MEDIUM
2         0               0 CURRENT   FREQUENT   HIGH
3         0               0 CURRENT   FREQUENT MEDIUM
4         0               0 CURRENT   FREQUENT MEDIUM
5         0               1 CURRENT   FREQUENT MEDIUM
6         0               0 CURRENT INFREQUENT MEDIUM
```

It would be necessary to rectify that before doing any work, because otherwise errors would be encountered later

Although the Data Frame looks better, we can't use this in our code because, again:

- It's bad practice
- It means we can't creat linear models if we use the correct syntax
  - i.e. the syntax that the `predict()` function likes, and the syntax generally required by tidyverse packages.
- If we use the `predict()` function, we will need to create a vector assignment with a syntactically correct name before creating the model anyway,
  - Which is like doing the same thing but with more typing…talk about taking a long walk to a drink of water.
- There is always the `attach()` command, DO NOT USE IT
  - Because the google style guide says not to.[5] The *tidyverse* style guide doesn't really mention it though.[6]

We can automatically fix these names to something desirable using the base package `make.names()`:

```
       names(donors) <- make.names(names(donors))
       print(names(donors))
       > print(names(donors))
 [1] "Donated"           "Veteran"           "Bad.Adress"        "Age"
 [5] "Has.Children"      "Wealth.Rating"     "Interest.Veterans" "Interest.Religion"
 [9] "Pet.Owner"         "Catolog.Shopper"   "Recency"           "Frequency"
[13] "Money"
```

[5] https://google.github.io/styleguide/Rguide.xml

[6] http://style.tidyverse.org/syntax.html#assignment

MODEL PERFORMANCE TRADE OFFS

RARE EVENTS

Rare events create challenges for classification models, where an event is rare (e.g. donation), predicting the opposite (refraining from donating) can result in a very high accuracy, as happened in our R script with the `donors` data set.

ROC AND AUC CURVES

ROC curves are an important tool for comparing models and selecting the best model for project needs.

When used with a single model it can help to visualise the trade off between True positives and False positives for the outcome of interest.

DISCRETE OR CATEGORICAL DATA

Where data is categorical

e.g.

1. poor

2. Middle Class

3. Rich

It is necessary to convert that data to a factor before it is used in a logistic regression.

Even where the data may be numeric as it is in the above example, it may be appropriate to convert it to a factor so each variable is considered on its own.

Remember when creating the factor to specify the levels of the factor if it is ordered:

```
factor(donors$wealth_rating, levels = 0:3, labels = c(0,1,2,3)
```

INTERACTION EFFECTS

Where data may interact with each other (e.g. stimulants and depressants neutralising perceived mood changes or smoking in combination with obesity raising the risk of heart disease more sharply than if they were separate), R will perform a logistic regression for the interaction effect between an x1 and x2 variable with the following formula:

```
glm(y ~ x1 * x2, family = "binomial")
```

AUTOMATIC FEATURE SELECTION

Unlike some machine learning methods, regression requires the human to select the parameters before the model is made.

So in the case of donations, it was possible to use insight into the potential factors that may be predictive of whether or not a person would make a donation.

If we don't have that insight ahead of time or there are too many predictors to go through individually a process called *automatic feature selection* can be used.

STEPWISE REGRESSION

Involves building a regression model step by step evaluating each predictor to determine which ones add value to the model.

The potential caveats of stepwise regression include:

- Forward and Backward stepwise selection may reach different models
- Neither method is guaranteed to find the best possible model
- There are concerns that a stepwise model violates some of the principles that allow a regression model to *explain* data as *predict*.
    - This might not be a large concern if you are ONLY interested in the predictive **power** of the regression model.
        - The use of stepwise does not mean that the models predictions are worthless, it simply means that the model may overstate or understate the importance of some of the predictors
- Feature selection methods like stepwise regression allow the model to be built in the absence of theory or even common sense, this can result in a model that is counterintuitive in the real world

Stepwise regression should be considered one tool for exploring potential models in the absence of some other good starting point

Though stepwise regression is frowned upon, it may still be useful for building predictive models, it would NOT be correct to say that a stepwise model's predictions could not be trusted.

BACKWARD DELETION

Backward stepwise deletion begins with a model that contains all of the predictors

It then checks to see what happens when each one of the predictors is removed from the model.

If removing a predictor does not substantially affect the models ability to predict the outcome then it can be safely deleted.

FORWARD SELECTION

This is like backward deletion but applied in the other direction

Start with a model that has no predictors, examine each predictor if any has any offers any improvement to the models predictive power.

Predictors are added step by step until no new predictors add substantial value to the model.

Backward and forward stepwise selection may end up with completely different models.

Training a Regression Model

Definitions

A model will often perform better on data from which it was created meaning that it can be difficult to assess the model's accuracy or efficacy; hence data will often be split up into multiple sets, the typical size ratio is in parenthesis (although this is a guide):

Training Data (50%)

Training data is the collection of data that you are going to use to create your model, it is a collection of inputs/output observations.

> Regression is a form of supervised learning because we have provided the output to every input.

Validation Data (25%)

Validation Data is a collection of data used purely for validating the model.

That means it is used to compare multiple models and determine the most appropriate model for the data.

Testing Data (25%)

Testing Data is used to estimate the accuracy of the selected approach

> (e.g. mean error for predictions (RMSE), perhaps compared to standard deviation of predictions.)

Excerpt from *Elements of Statistical Learning (2nd ed)*[1]

If we are in a data-rich situation, the best approach for both problems is to randomly divide the dataset into three parts:

- a training set,
  - The training set is used to fit the models;
- a validation set,
  - the validation set is used to estimate prediction error for model selection;
- a test set.
  - the test set is used for assessment of the generalization error of the final chosen model.

> Ideally, the test set should be kept in a "vault," and be brought out only at the end of the data analysis. Suppose instead that we use the test-set repeatedly, choosing the model with smallest test-set error. Then the test set error of the final chosen model will underestimate the true test error, sometimes substantially.

[1] Hastie, T., Tibshirani, R. and Friedman, J. (2009). *The elements of statistical learning*. 2nd ed. New York: Springer, p.222.

The Workflow for Machine Learning

1. Training Phase

Use the training data to create a model from the inputs for the outputs

2. Validate the Model (Model Selection)

Use the validation data to compare the performance of multiple models and favour the best performing and least complex model

3. Testing Data (Model Assessment)

Use the Testing Data to estimate model properties such as RMSE to assess the performance of the Model

4. Application of the Model

Apply the model to the real-world data to get predictions and speculate about the quality of the model using the results of the validation phase and perhaps the testing data phase.

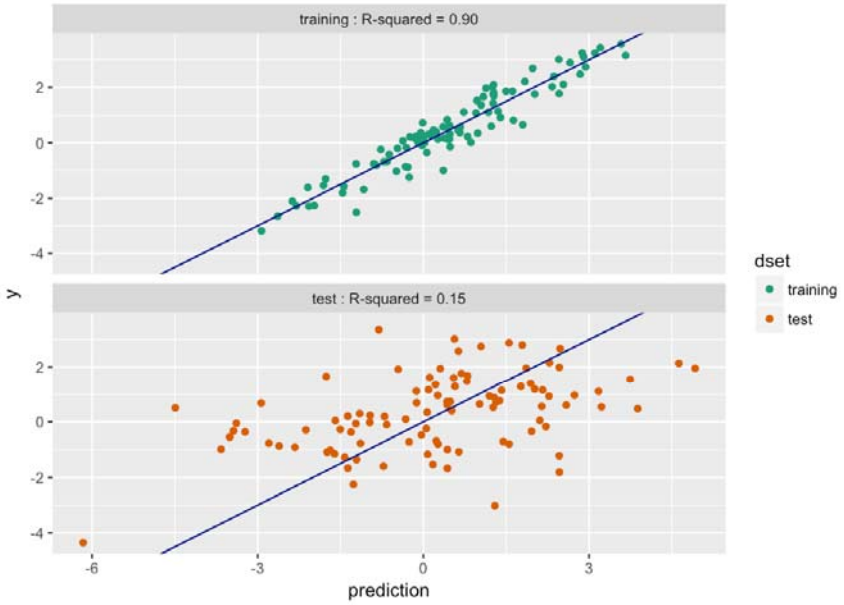
Properly Training a Model

Training, Testing and Validation Data

Training Data is used to create a model.

> A model will always perform better on its own training data than on data that it has not seen.
>
> For simple models like *Linear Regression* this bias is not often severe, however for more complex models using only the training data to evaluate the model can produce misleading results.

For example here is a ground-truth plot of a model fitted to training data and then applied to testing data it had not yet seen:



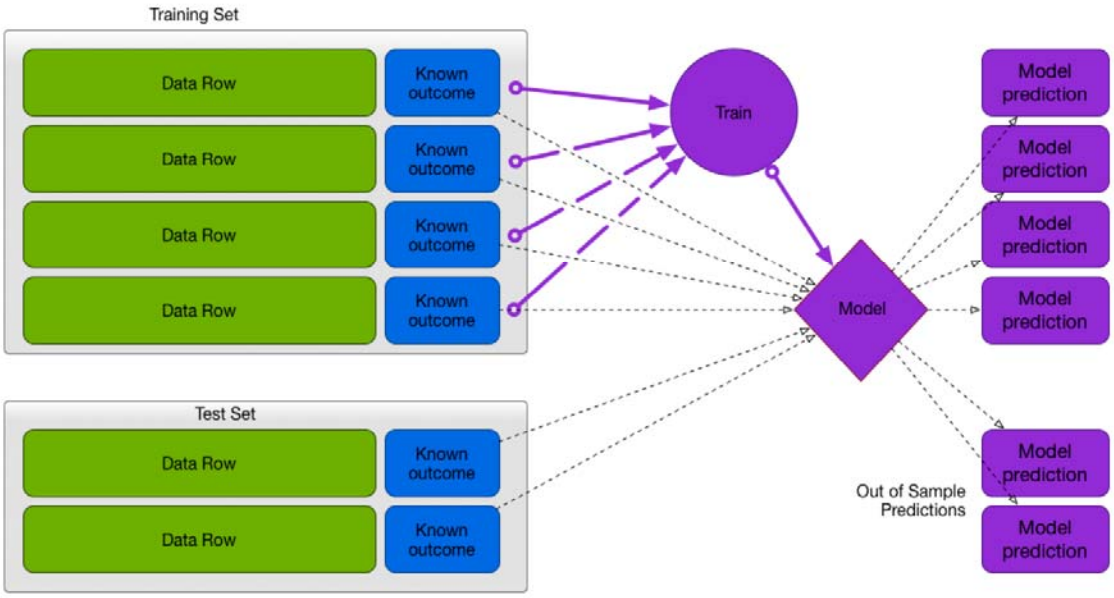
- Training R^2: 0.9; Test R^2: 0.15 -- **Overfit**

The poor performance on the testing data is indicative of an overfit model.

Test/Train Split

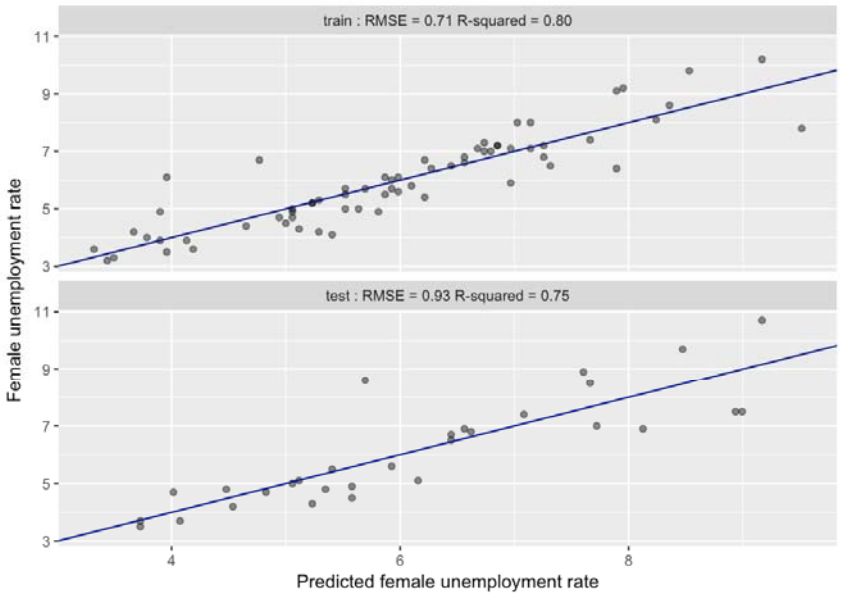
When data is plentiful, the best thing to do is to split the data in two:

- One Set to ***Train*** the model (i.e. create the model)
- Another Set to ***Test*** the model (i.e. evaluate graphically and observe values like RMSE)



Example

Take the data frame containing unemployment rates that we have been using, if the data was split into training and testing data and a linear model was fit to the training data and then used to predict the test data we would have a ground truth plot like this:



- Training: RMSE 0.71, R^2 0.8
- Test: RMSE 0.93. R^2 0.75

The diagnostic values are close and the model performs similarly on both sets, so we know the model isn't overfit

> (An understanding of the data helps, we would expect a directly proportional linear relationship between the unemployment rates in genders over time, i.e. if half the men looking for work were unable to find any then perhaps we could expect roughly half the women looking for work to have trouble to).

Cross-Validation

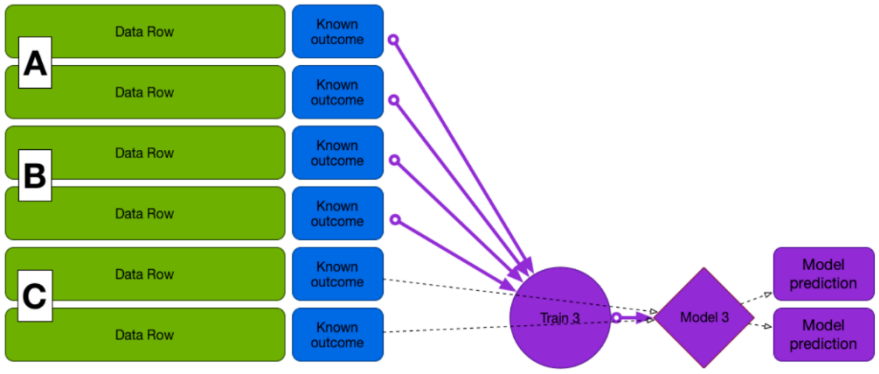
When there is insufficient data to split data up into training and test sets, we use ***cross validation*** to estimate a models *'out-of-sample performance'*.

In this case all our data would be used eventually as training data to create a model, however cross-validation is used in order to assess how the model might perform before hand (in this case you need to have already decided upon a model because there isn't a validation set, just a test and training).

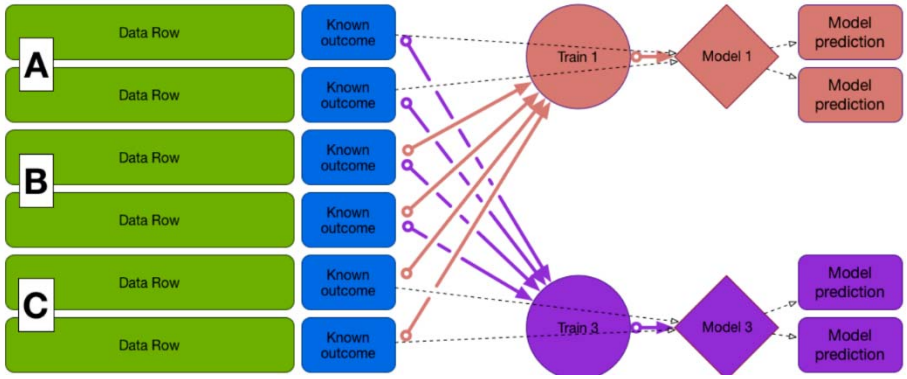
n-fold Cross-Validation

Partition the data into n subsets, for the example we will use $n = 3$ and call the sets A, B and C:

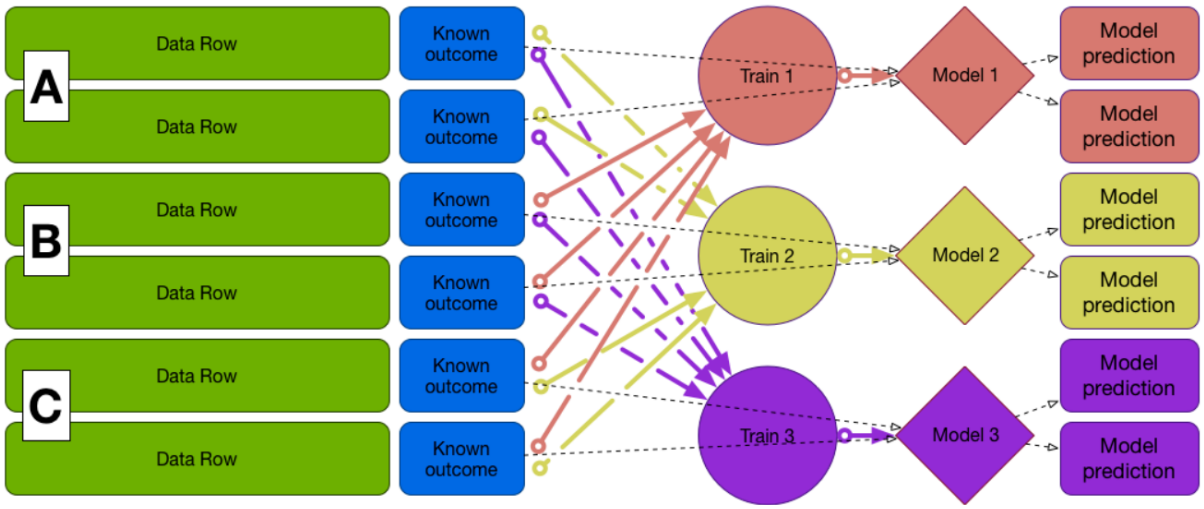
1. **Train a model using the data from sets A and B and use that model to make predictions on C:**



2. **Train a model on B and C to predict on A:**



3. **Then predict a model on A and C to predict on B:**



Now observe that none of these models has made a prediction on its own ***training data***, all the predictions are essentially, ***Test set*** predictions.

Hence, the R^2 and $RMSE$ calculated from this should provide an unbiased estimate on how a model fit to **all** the training data will perform on future data.

Implementing a Cross-Validation plan

A cross validation plan can be implemented by using the `kWayCrossValidation` program from the `vtreat` package:

```
> library(vtreat)
> kWayCrossValidation(nRows, nSplits, NULL, NULL)
```

Where:

- `nRows`
  - Is the number of rows in the training data
- `nSplits`
  - Is the number of folds (i.e. partitions) in the cross-validation
- The other two arguments are not needed here so just `NULL` them out

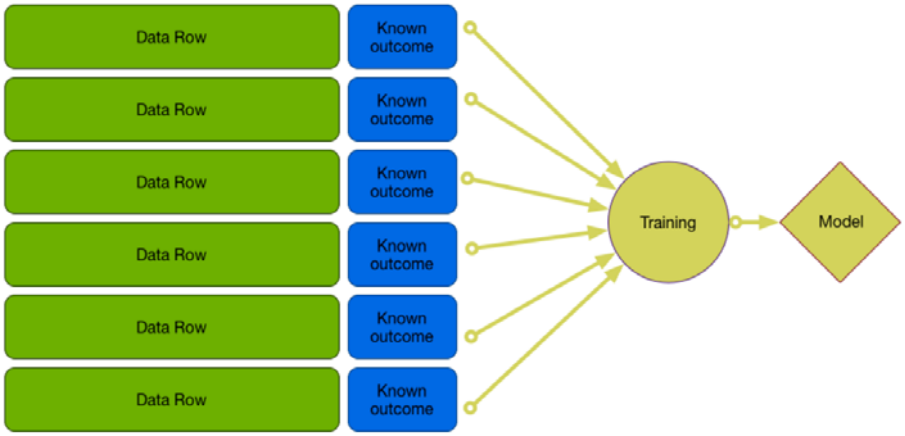
The program returns the indicies for testing and training data (it calls training app, same thing different name):

```
> splitPlan[[1]]
## $train
## [1]  1  2  4  5  7  9 10
##
## $app
## [1] 3 6 8
```

Use the data with the training indices to train the model and then make predictions using the app/testing indices:

```
> split <- splitPlan[[1]]
> model <- lm(fmla, data = df[split$train,])
> df$pred.cv[split$app] <- predict(model, newdata = df[split$app,])
```

If the estimated model performance looks good enough, use all the data to fit a final model:



You can't however, evaluate the future performance of this model because there isn't enough data to evaluate it with, however, that was the whole point of the cross validation from before, those diagnostics (e.g. $RMSE$, R^2) should be indicative of the performance of this model.

Cross Validation vs Test/Train Split

Cross validation only tests the modelling process while test/train split evaluates the final model.

The cross-validation estimates are similar to those from using a test/train split:

Example: Unemployment Model

Measure type	RMSE	R^2
train	0.7082675	0.8029275
test	0.9349416	0.7451896
cross-validation	0.8175714	0.7635331

Loss Functions and Errors

Training a Linear Regression Model | Week 4 Material

****I've inadvertently used order where I mean degree, order refers to the extent of differentiation performed, degree refers to the highest power present in a polynomial function, my bad…*

What we are trying to do

Take some data:

```
capture.output(print(head(all.df), print.gap =3), file ="alldata.txt")
```

```
    X       input      output
1   1   -16.99955    -3846633
2   2    79.99105   148007979
3   3    17.62219    -4311459
4   4   -19.50413     3852745
5   5    93.00449   168974952
6   6    34.37293     9399792
…
```

In order to fit a model to the data, it is necessary to use the following workflow:

1. Train the model using a sample of data
2. Validate the performance of models using another sample and select the best performing model that makes the least number of assumptions
3. Test the performance of the model using yet another sample.

Splitting the Data

Set the Seed

Setting the seed allows R to produce random numbers that will be consistent across like scripts, it is analogous to remembering which column and row random numbers are drawn from a table.

Strictly speaking we don't need to set the seed, this data was created as 5th degree polynomial with added noise and this process will always provide a model that is a 5th degree polynomial, moreover the plots are coded relatively such that the crosshairs are always drawn at the point of lowest validation error, it can be a good habit to remember to do this though:

```
set.seed(23)
```

There are two ways to split the data

- Cross-Validation
- Random Subsets

Where there is insufficient data, cross-validation may be employed, in this case we will perform a random split of the data.

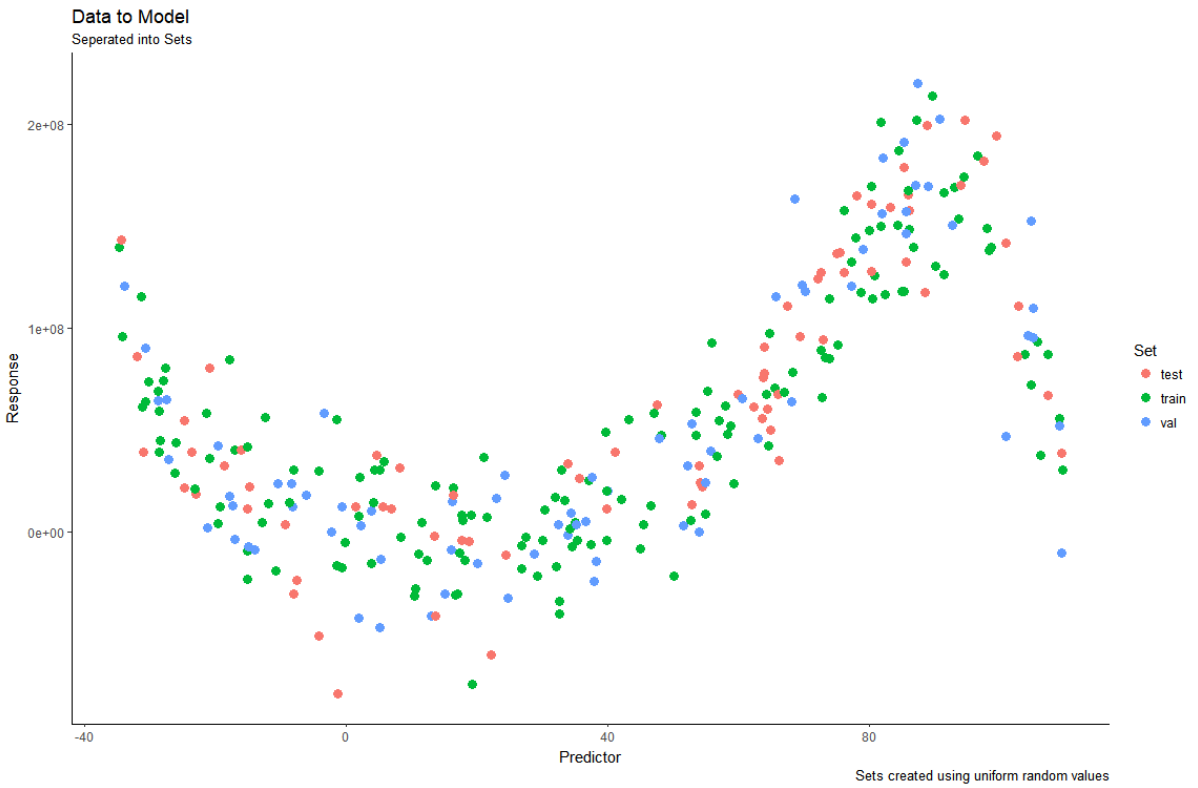
```r
# Create Data Subsets ------------------------------------------------------

  ##So we want to create 3 random samples of data

  N     <- nrow(all.df)
  idval <- sample(N)

  train.ID <- idval[0:(N*0.5)]
  val.ID   <- idval[((N*0.5)+1):(N*0.75)]
  test.ID  <- idval[((N*0.75)+1):N]

  train.df <- all.df[train.ID,]
  val.df   <- all.df[val.ID,]
  test.df  <- all.df[test.ID,]


      #Order the Data Frame
      train.df <- train.df[order(train.df$X),]
      val.df   <- val.df[order(val.df$X),]
      test.df  <- test.df[order(test.df$X),]
```

Visualise the Data

It isn't always possible to visualise the data, however where it is possible (by using 3 spatial dimensions, colour, shapes, tranperancy etc.) it should be done.

In this case the data is only two dimensional so it can be visualised:

```
all.df$sample            <- FALSE
  all.df[train.ID,]$sample <- "train"
  all.df[val.ID,]$sample   <- "val"
  all.df[test.ID,]$sample  <- "test"

 head(all.df)
##   X     input    output sample
## 1 1 -16.99955  -3846633  train
## 2 2  79.99105 148007979  train
## 3 3  17.62219  -4311459  train
## 4 4 -19.50413   3852745  train
## 5 5  93.00449 168974952    val
## 6 6  34.37293   9399792  train

 all.plot <- ggplot(all.df, aes(x = input)) +
   geom_point(data = all.df, size = 3, aes(y = output, col = sample)) +
   theme_classic() +
   labs(x = "Predictor", y = "Response", col = "Set", title = "Data to Model",
        subtitle = "Seperated into Sets",
        caption = "Sets created using uniform random values")
 #  scale_color_brewer(palette="Pastel2")
```



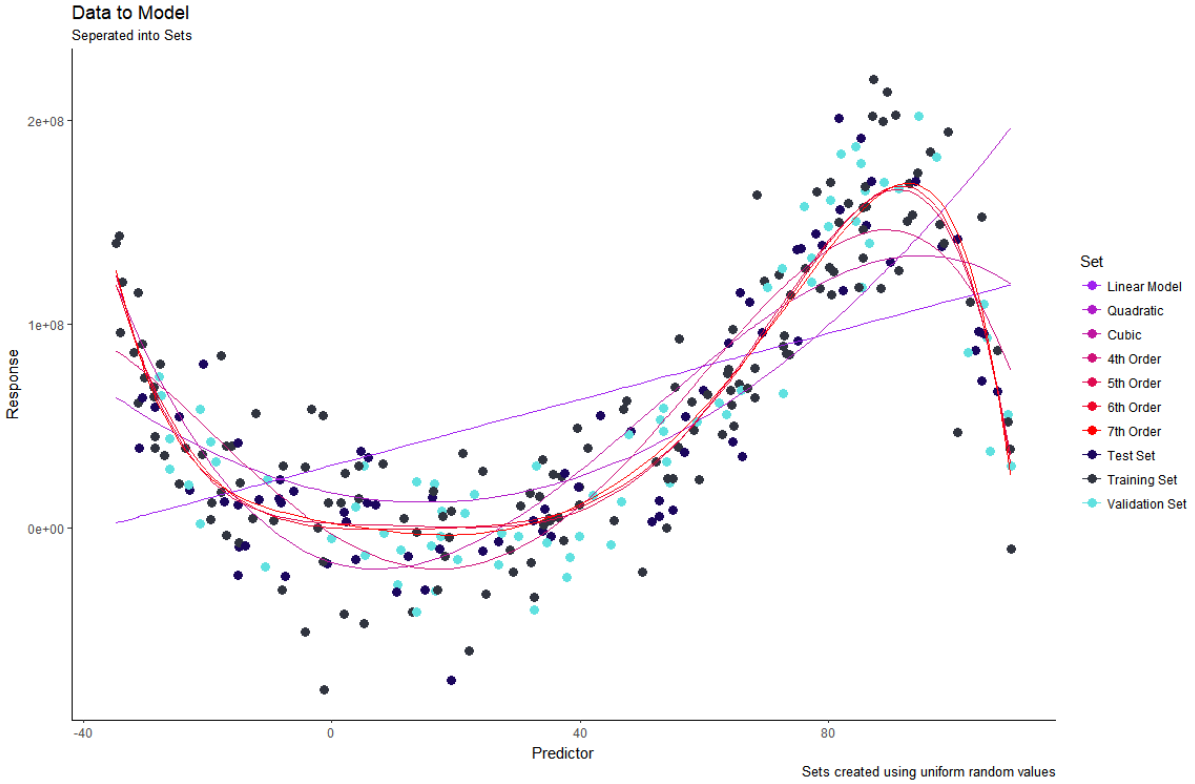
Model Training

Training a model refers to fitting models to the set of training data. We don't know which model will be appropriate for this data, but the data appears to have a natural curve so we will fit a few polynomial models and see how they perform:

> Be aware, that when making models `~I(X^n) or ~ poly(x, n)` can both be used, it has something to do with correlated coefficients and orthogonal coefficients; but I couldn't figure out which to use and why.

```
#train models of varying complexity
 # attach(train.df)
 head(train.df)

 mod.lm <- lm(output ~ input, data = train.df)
 mod.p2 <- lm(output ~ I(input^2) + input ,data = train.df)
 mod.p3 <- lm(output ~ I(input^3) + I(input^2) + input, data = train.df)
 mod.p4 <- lm(output ~ I(input^4) + I(input^3) + I(input^2) + input, data = train.df)
 mod.p5 <- lm(output ~ I(input^5) + I(input^4) + I(input^3) + I(input^2) + input, data
= train.df)
 mod.p6 <- lm(output ~ I(input^6) + I(input^5) + I(input^4) + I(input^3) + I(input^2)
+ input, data = train.df)
 mod.p7 <- lm(output ~ I(input^7) + I(input^6) + I(input^5) + I(input^4) + I(input^3)
+ I(input^2) + input, data = train.df)


  #Create Predictions and throw everything into a list
 train.df <- data.frame(train.df,
                        mod_lm = predict(mod.lm),
                        mod_p2 = predict(mod.p2),
                        mod_p3 = predict(mod.p3),
                        mod_p4 = predict(mod.p4),
                        mod_p5 = predict(mod.p5),
                        mod_p6 = predict(mod.p6),
                        mod_p7 = predict(mod.p7)
                        )[order(train.df$X),]

 train.mod.df <- melt(data = train.df, id.vars = c("X", "input", "output"))

  train.mod.list <- list(mod.lm, mod.p2, mod.p3, mod.p4, mod.p5, mod.p6, mod.p7)

  training.list <- list(models = train.mod.list, training_data = train.df,
model_predictions = train.mod.df )

  #Visualise the Models
    ##Create label and colour vectors
     plot.labels <- c("Linear Model", "Quadratic", "Cubic", "4th Order",
                  "5th Order", "6th Order", "7th Order",
                  "Test Set", "Training Set", "Validation Set")

     colfunc <- colorRampPalette(c("purple", "Red"))
     mycol <- c(colfunc(7), "#1B065E", "#30343F", "#60E1E0")
     plot(rep(1,length(mycol)),col=mycol,pch=19,cex=3)

     ##Plot the models over the Data
      all.plot +
        geom_line(data = train.mod.df, aes(x = input, y = value, col = variable)) +
        scale_color_manual(values = mycol, labels = plot.labels)
```

This returns the following plot of the models:



Hence, the question becomes, which model is the appropriate one to use?

Training and Validation Error

The root mean square error $\left(RMSE = \sqrt{\frac{1}{n} \times \sum_{i=1}^{n}[(y_i - \hat{y}_i)]^2}\right)$ is a measurement of the expected error from a data point to the model value.

It can be used, thusly, to measure model performance.

However, a more complicated model will always perform better on the data it was trained on:

> Imagine a polynomial of the 100th order, it could turn to intersect with every other data point, or, in a more abstract sense, imagine an explanation that is so convoluted that it couldn't not describe observation.

This is addressed by assessing the model performance on data that was not used to create the model, this set is called the validation data.

> In this case a 100th order polynomial would not explain the data it hadn't seen very well, because the model wouldn't generalise the shape very well and all the turns would only add to the model error (i.e. there is a saturation point) .
>
> In a more abstract sense, a good way to test convoluted theories is to test their capacity to predict future events, e.g. religion can explain the origin of species but it had failed to predict the degree of random mutation (think viruses, bacteria or even some mammalian species) regularly observed in species, this is good evidence that the model may be overly convoluted and that there is a simpler explanation.

Training Error

The training Error must be calculated for each model, this can be achieved thusly:

```
train.rmse.lm <- sqrt(sum((
    train.df$output-predict(mod.lm, newdata = train.df))^2
    )/nrow(train.df))
  train.rmse.p2 <- sqrt(sum((
    train.df$output-predict(mod.p2, newdata = train.df))^2
    )/nrow(train.df))
  train.rmse.p3 <- sqrt(sum((
    train.df$output-predict(mod.p3, newdata = train.df))^2
    )/nrow(train.df))
  train.rmse.p4 <- sqrt(sum((
    train.df$output-predict(mod.p4, newdata = train.df))^2
    )/nrow(train.df))
  train.rmse.p5 <- sqrt(sum((
    train.df$output-predict(mod.p5, newdata = train.df))^2
    )/nrow(train.df))
  train.rmse.p6 <- sqrt(sum((
    train.df$output-predict(mod.p6, newdata = train.df))^2
    )/nrow(train.df))
  train.rmse.p7 <- sqrt(sum((
    train.df$output-predict(mod.p7, newdata = train.df))^2
    )/nrow(train.df))
```

Although, because:

1. A mistake may be made when calculating the rmse for other sets
2. This is laborious
3. This is incompatible with loops

It is advisable to use a function to calculate the rmse:

```
rmse <- function(dframe, model){

    y     <- dframe[,names(dframe) == "output"]
    y.hat <- predict(object = model, newdata = dframe)
    resid <- y-y.hat
    RSS   <- sum(resid^2)
    T.Error <- RSS/nrow(dframe)
    RMSE <- sqrt(T.Error)

       return(RMSE)
  }
```

Now we can use a loop to calculate the RMSE values, this is important because:

1. For a large number of models, it won't be possible to write out every line (even with a function to expedite the process)
2. It increases the probability of a transcription error, a loop isn't immune to errors, but atleast they will be consistent

```
##Use a for loop to calculate all RMSE Values
    ##Create a data frame to store the values
       error.df <- data.frame(matrix(
         ncol = 4, nrow = length(training.list$models)))
       colnames(error.df) <- c("Order",
                               "Training Error",
                               "Validation Error",
                               "Test Error")

    ##Execute the loop
  for (i in 1:7) {

    rmse.val <- rmse(train.df, training.list$models[[i]])
```

```
  error.df[i,] <- c(order = i,
                    training_error = rmse.val,
                    validation_error = NA,
                    test_error = NA)
  }
```

Validation Error

The validation error can be calculated by reusing the function and loop:

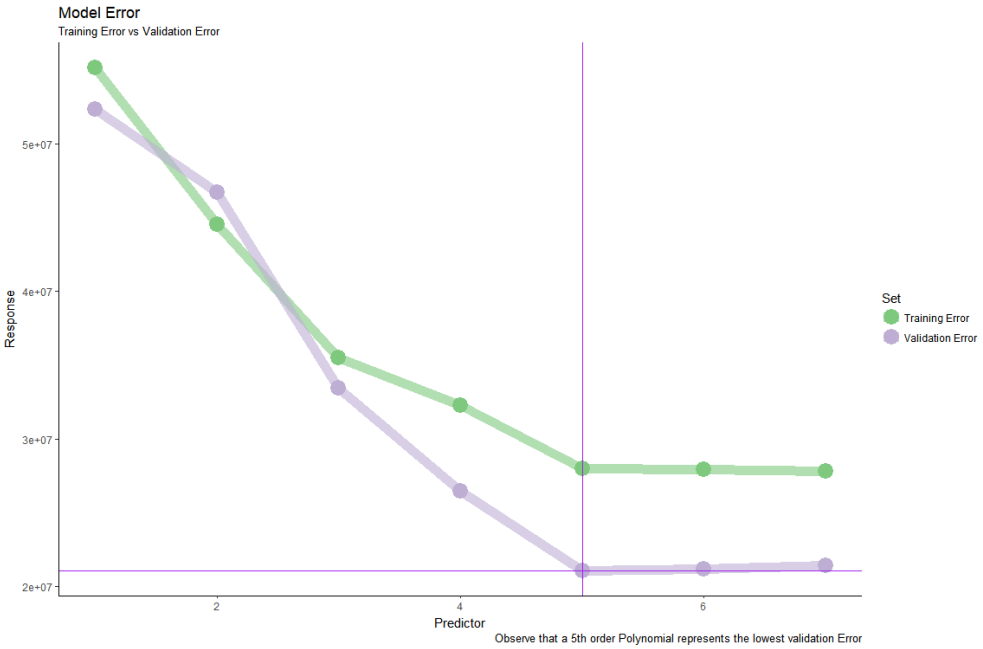
```
# Calculate the Validation Errors ----------------------------------------

  ##Use a for loop to calculate all validation error values

    #use the error.df data frame from before
  for(i in 1:7){
    rmse.val <- rmse(val.df, training.list$models[[i]])
    error.df[i,3] <- rmse.val
  }
      error.df
```

This returns the following Errors:

```
> head(error.df)
  Order Training Error Validation Error Test Error
1     1       55128411         52332313         NA
2     2       44557457         46697399         NA
3     3       35515482         33472469         NA
4     4       32259954         26480824         NA
5     5       27984169         21099237         NA
6     6       27932759         21207056         NA
```

These Errors can be plotted:

```
# Plot the Errors
      error.df.melt <- melt(error.df[,-4], id.vars = "Order", variable.name = "Set",
value.name = "Error")
      error.df.melt

      #Mininmum Validation Error
      min.val.error.y <- min(error.df$`Validation Error`)
      min.val.error.x <- error.df[
        error.df[,names(error.df)=="Validation Error"
                 ]==
          min(error.df$`Validation Error`),]$Order

 error.plot <- ggplot(error.df.melt, aes(x = Order, y = Error, col = Set)) +
   geom_line(size = 4, alpha = 0.6) +
   geom_point(size = 6) +
   theme_classic() +
   labs(x = "Predictor", y = "Response", col = "Set", title = "Model Error",
        subtitle = "Training Error vs Validation Error",
        caption = "Observe that a 5th order Polynomial represents the lowest
validation Error") +
   geom_vline(xintercept = min.val.error.x, col = "Purple") +
   geom_hline(yintercept = min.val.error.y, col = "Purple") ; error.plot +
   scale_color_brewer(palette="Accent")
```



If the Training Error continues to decrease, whilst the true error increases, it can be determined that models of such complexity have been overfit.

Hence it can be concluded that the appropriate model for this data is a 5^{th} order polynomial, because such a model has the lowest validation error.

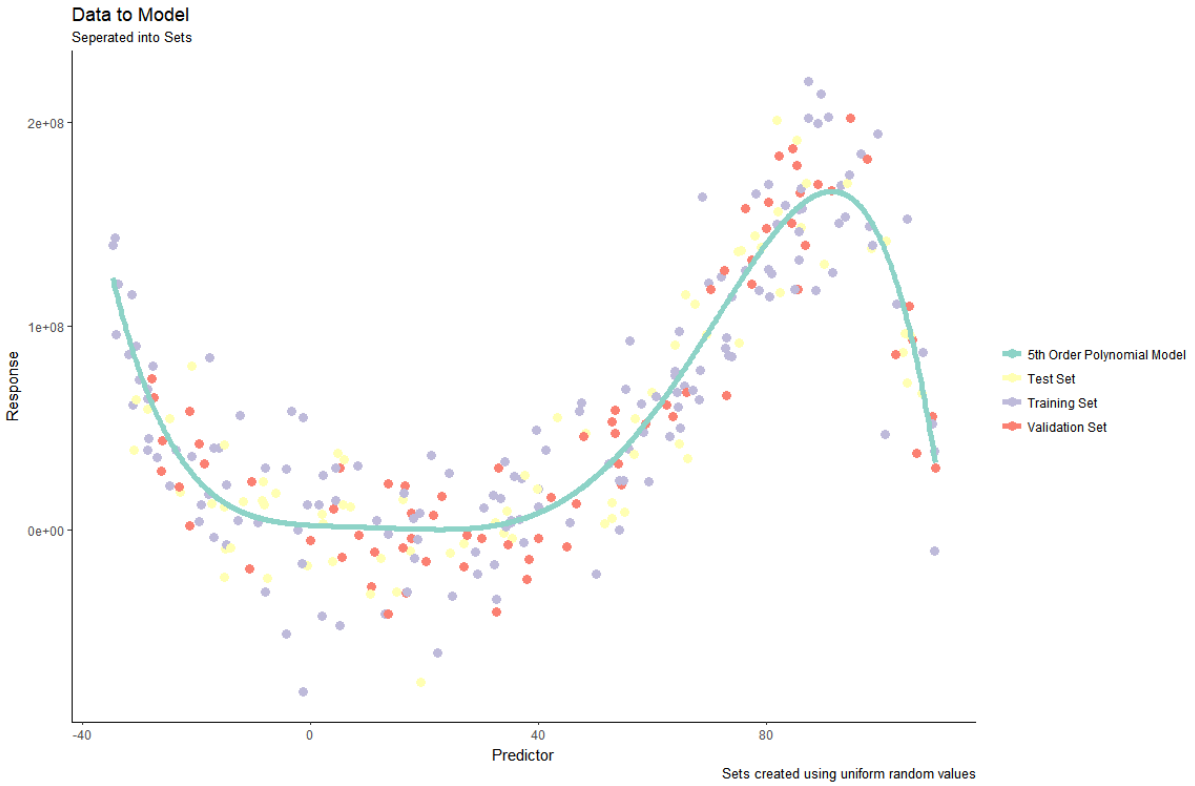
Assess the Model

Graphical Plot

The model can be plotted over all the data,

Although, it might be necessary to predict more data points to get a smoother curve over the points:

```
all.plot +
   geom_line(data = train.df, aes(y = mod_p5, col = "Purple"),
             size = 7, alpha = 0.6) +
   scale_color_brewer(palette="Set3", labels = c("5th Order Polynomial Model",
                                                 "Test Set",
                                                 "Training Set",
                                                 "Validation Set")) +
   guides(col = guide_legend(title = NULL))

##Make a slightly smoother plot
 p5.mod <- data.frame(input = seq(from = min(all.df$input),
                                  to = max(all.df$input), length.out = 1000))
 p5.mod$mod_p5 <- predict(object = mod.p5, newdata = p5.mod)

 all.plot +
   geom_line(data = p5.mod, size = 2, alpha = 1, aes(y = mod_p5,
                                                     col = "purple")) +
   scale_color_brewer(palette="Set3",
                      labels = c("5th Order Polynomial Model",
                                 "Test Set",
                                 "Training Set",
                                 "Validation Set")) +
   guides(col = guide_legend(title = NULL))
```



RMSE

The rmse and standard deviation of the test data is:

```
> rmse(test.df, mod.p5)
[1] 24154642
> sd(test.df$output)
[1] 61435612
> rmse(test.df, mod.p5)/sd(test.df$output)
[1] 0.39317
```

This provides, that the expected error from the model to a data point is 24×10^6, this may seem large, but is only 39% the standard deviation of the output data.

This implies that the model is better at predicting the data than merely taking the average value

Viewing the Underlying Data

The data for this model, was formed using the following function:

```
y <- -2685 - 121497.4*x + 11226.49*x^2 - 1000*x^3 + 30.4*x^4 - 0.2*x^5 +
rnorm(length(x),0,noise)
```

Although the coefficients are different, at least the degree of the polynomial is correct.

Notes

Sunday, 3 June 2018 8:51 AM

Cross Validation

splitting data into sets and using the validation set to determine model complexity may leave too little data in the training set to create the model.

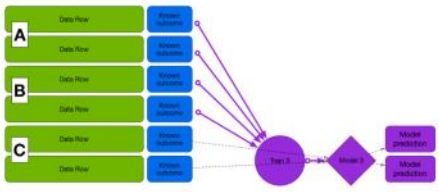
Cross validation splits the data into 'k' non-overlapping sets and the validation error is the average validation error over k trials.

Example of 3-fold cross-validation

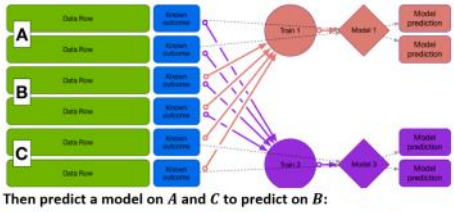
n-fold Cross-Validation

Partition the data into n subsets, for the example we will use $n = 3$ and call the sets A, B and C:

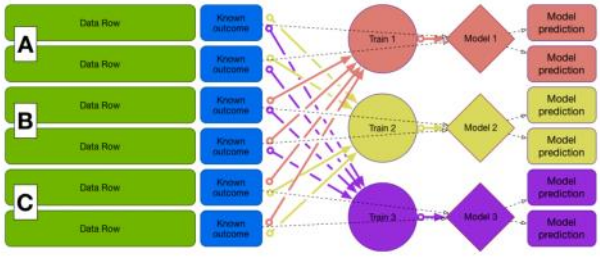
1. **Train a model using the data from sets A and B and use that model to make predictions on C:**



2. **Train a model on B and C to predict on A:**



3. **Then predict a model on A and C to predict on B:**



Now observe that none of these models has made a prediction on its own ***training data***, all the predictions are essentially, ***Test set*** predictions.

Hence, the R^2 and $RMSE$ calculated from this should provide an unbiased estimate on how a model fit to **all** the training data will perform on future data.

Implementing Cross Validation

it is not necessary to implement a full split plan because glmnet will do it automatically with "cv.glmnet"

Estimators

There are often parameters that are not known e.g. (μ, σ)

where we do not know the parameter we take an estimate.

if the unknown parameter is some θ

then the estimated value is $\hat{\theta}$

two properties of estimators are:

- bias (measures the expected deviation between the true value and the estimator value (of the function or parameter))
- Variance (measures the deviation from the expected estimator value between samples)

Bias

Bias is a property of an estimator.

it is the difference between the "expected estimate" and the actual parameter

$$bias(\hat{\theta}) = E(\hat{\theta}) - \theta$$

estimating bias

$$bias(\hat{\theta}) = E(\hat{\theta}) - \theta$$

estimating bias

Even if we dont know the value of θ the bias can be calculated for example:

Take a sample from a normally distributed population:

population size: N sample size: n

population mean: μ sample mean: $\bar{x}$

population std. dev: σ biased sample std. dev.: $\hat{\sigma}$

unbiased sample std. dev: s

Sample mean ($\bar{x}$) as an estimator for population mean (μ) in a Normal Distribution

if we take the sample mean as an estimator of the population mean:

$$\bar{x} = \hat{\mu}$$

we can determine the bias thusly:

$$\begin{aligned} bias(\hat{\mu}) &= E(\hat{\mu}) - \mu \qquad (E(\theta) \text{ means 'the expected value of } \theta') \\ &= E(\bar{x}) - \mu \\ &= E\left(\frac{1}{n} \times \sum_{i=1}^{n} [x_i]\right) - \mu \\ &= \frac{1}{n} \times \sum_{i=1}^{n} [E[x_i]] - \mu \qquad (\text{the expected value of any } x_i \text{ in the population or sample is the population mean } (\mu)) \\ &= \frac{1}{n} \times \sum_{i=1}^{n} [\mu] - \mu \\ &= \mu - \mu \\ &= 0 \end{aligned}$$

$\therefore$ the sample mean is an unbiased estimator of the population mean

(this says nothing of the variance, std. error or MeanStdError of the estimator)

Variance of sample as an estimator for population std deviation (σ)

So firstly a comment on nomenclature, im going to call:

- Variance of sample $\equiv$ biased sample variance $= \frac{1}{n} \times \sum_{i=1}^{n} \left[(x_i - \bar{x})^2 \right]$
- Sample variance $\equiv$ unbiased sample variance $= \frac{1}{(n-1)} \times \sum_{i=1}^{n} \left[(x_i - \bar{x})^2 \right]$

right now we are looking at the bias of the first one

let $\hat{\sigma}^2$ be variance of sample

$$\Rightarrow E(\hat{\sigma}^2) = E\left(\frac{1}{n} \times \sum_{i=1}^{n} \left[(x_i - \bar{x})^2 \right]\right)$$

$$= \frac{n-1}{n} \times \sigma^2$$

now,

$$\begin{aligned} bias(\hat{\sigma}^2) &= E(\hat{\sigma}^2) - \sigma^2 \\ &= \left(\frac{n-1}{n} \times \sigma^2\right) - \sigma \\ &\vdots \\ &= \frac{\sigma^2}{n} \end{aligned}$$

$\therefore$ taking the variance of a sample is a biased estimator of the population variance

Sample standard deviation (s) as an estimator of population variance (σ^2)

So take the sample std. dev. (s) as an estimator of the population std. dev. (σ)

$$s^2 = \hat{\sigma^2}$$

now we are looking at the bias of the second one

let $\hat{\sigma}^2$ be the expected variance of the sample

$$\Rightarrow E(\hat{\sigma}^2) = E\left(\frac{1}{n} \times \sum_{i=1}^{n} \left[(x_i - \bar{x})^2 \right]\right)$$

$$\begin{aligned} &= \frac{n}{n-1} \times E(\hat{\sigma}^2) \\ &= \frac{n}{n-1} \times \left(\frac{n-1}{n} \sigma^2\right) \\ &= \sigma^2 \end{aligned}$$

$$\begin{aligned} bias(\hat{\sigma}^2) &= E(\hat{\sigma}^2) - \sigma^2 \\ &= \sigma^2 - \sigma^2 \\ &= 0 \end{aligned}$$

$\therefore$ the sample variance s^2 is an unbiased estimator of the population variance σ^2.

So the whole point of going through that was to give an idea on how bias() actually works.

Variance

Variance is a function of:

- how much the estimator is expected to vary between samples

the square root of the variance is:

- approximately* the expected difference from the expected estimator between samples

* because $Var(X) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}[(x_i - \mu)^2]} \approx \text{mean}(|x_i - \mu|)$

└ this isn't used because $\frac{d}{dx}(|x|)$ is undefined $\forall x \in \mathbb{R}$

e.g. $f(x) = |x| \Rightarrow \frac{d}{dx}(f(0)) = \text{'undefined'}$

Mean Square Error

mean square error is a combination of bias and variance

$$MSE(\hat{\theta}) = (bias(\hat{\theta}))^2 + Var(\hat{\theta})$$
$$= E((\hat{\theta} - \theta)^2)$$

To choose the compromise between bias and variance, cross validation is used.

Generalised Linear Models

A generalised linear model has three components

- Probability distribution (e.g. normal, poisson, gamma, binomial etc...)
- linear predictor variables (i.e. $y = a_1x_1 + a_2x_2 + \dots$)
- link function (a transformation applied to the variables to make it work:

e.g. $y = mx + b$ applied to binomial data in a logistic regression:

$$p(Y=1) = \frac{1}{1+e^{-z}}; \quad z = mx + b$$

AIC

The Akaike Information Criterion (AIC) is a way of measuring model performance by promoting performance and penalising complexity:

if $\hat{y}$ is some model with 'k' parameters from a sample size of N:

$$AIC(\hat{y}) = N \times \log\left(\frac{RSS(\hat{y})}{N}\right) + 2k$$

$$e^{AIC} = \left(\frac{RSS}{N}\right)^N \times e^{2k}$$

Regression with lots of predictor variables

So if we had lots of predictor determining which to include becomes a pertinent question and it isn't practical to test every combination.

For linear regression in two variables we used:

let: the formula be $y = w_0 + w_1x$

: $w = (w_0, w_1)$

we chose W that returned:

$$\min\left(\text{sum}\left(L(y_i, \hat{y})\right)\right)$$

typically the loss function was the squared error so by way of the Ordinary Least Squares method:

choose W that corresponds to:

$$\min\left[\text{sum}\left((y_i - \hat{y})^2\right)\right]$$

we can still use this general idea if we penalise more complex models:

let: $P(W)$ be a function of W

$L(y_i, \hat{y})$ be a loss function

now, choose the W that corresponds to:

$$\min\left[\text{sum}\left(L(y_i, \hat{y})\right)\right] + \lambda \cdot P(W)$$

- λ is a scale of sorts, a 'tuning parameter' that allows the harshness of the penalty to be scaled. generally R (or python or whatever) will create a model for lots of λ values and the best performing one will be favoured.

this is a general formula that gives a general framework for model selection and model construction.

Ridge Regression

So ridge regression, is a form of regression that chooses model predictors and model coefficients.

let:

W be a vector of coefficients of not yet decided length or values

$W = (w_1, w_2, w_3, \ldots w_p)$

p be the number of coefficients

$L()$ be a loss function

$P(W) = \sum_{i=1}^{p}\left[(w_i)^2\right]$

then choose W such that it corresponds to the value:

$$\min\left\{\text{sum}\left[L(y_i, \hat{y})\right] + \lambda \cdot \left(\sum_{i=1}^{p}\left[(w_i)^2\right]\right)\right\}$$

the chosen model will hence favour:

- less coefficients
- coefficients closer to zero
- models with less error

by using different λ values we can help balance between minimizing error on training data and overparameterisation that may impede predictive capacity on testing data

The assumption of smaller coefficients

This technique will favour smaller model coefficients, assuming the data is first standardised (i.e. put on the same scale with $x_i = \frac{x_i - \bar{x}}{s}$)

this assumption is justified because the standardisation process means the modelling focus is on the shape, not the location of the data, look at the excel example

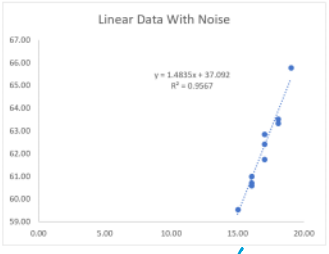
Celsius	Farenheit	Farenheit + WN	Std. Cel	Std. Far	Std. Far+WN
18.00	64.40	63.35	0.92	0.92	0.65

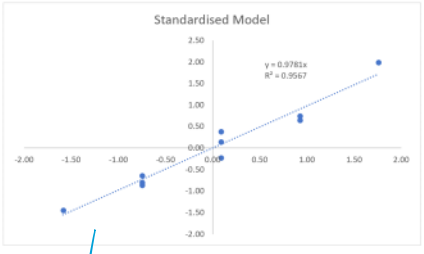
of the data, look at the excel example

Celsius	Farenheit	Farenheit + WN
18.00	64.40	63.35
16.00	60.80	61.01
17.00	62.60	62.44
16.00	60.80	60.73
18.00	64.40	63.53
19.00	66.20	65.79
17.00	62.60	61.76
15.00	59.00	59.56
17.00	62.60	62.86
16.00	60.80	60.61

Std. Cel	Std. Far	Std. Far+WN
0.92	0.92	0.65
-0.75	-0.75	-0.64
0.08	0.08	0.15
-0.75	-0.75	-0.79
0.92	0.92	0.75
1.75	1.75	1.99
0.08	0.08	-0.22
-1.59	-1.59	-1.43
0.08	0.08	0.39
-0.75	-0.75	-0.85

Mean	16.90	62.42	62.16
Std. Dev	1.20	2.15	1.82





$y = \frac{9}{5}x + 32$

$y_{std} = x_{std}$

Loss Functions and Errors

Training a Linear Regression Model | Week 4 Material

****I've inadvertently used order where I mean degree, order refers to the extent of differentiation performed, degree refers to the highest power present in a polynomial function, my bad…*

What we are trying to do

Take some data:

```
capture.output(print(head(all.df), print.gap =3), file ="alldata.txt")
```

```
    X       input      output
1   1   -16.99955    -3846633
2   2    79.99105   148007979
3   3    17.62219    -4311459
4   4   -19.50413     3852745
5   5    93.00449   168974952
6   6    34.37293     9399792
…
```

In order to fit a model to the data, it is necessary to use the following workflow:

1. Train the model using a sample of data
2. Validate the performance of models using another sample and select the best performing model that makes the least number of assumptions
3. Test the performance of the model using yet another sample.

Splitting the Data

Set the Seed

Setting the seed allows R to produce random numbers that will be consistent across like scripts, it is analogous to remembering which column and row random numbers are drawn from a table.

Strictly speaking we don't need to set the seed, this data was created as 5th degree polynomial with added noise and this process will always provide a model that is a 5th degree polynomial, moreover the plots are coded relatively such that the crosshairs are always drawn at the point of lowest validation error, it can be a good habit to remember to do this though:

```
set.seed(23)
```

Splitting the Data

There are two ways to split the data

- Cross-Validation
- Random Subsets

Where there is insufficient data, cross-validation may be employed, in this case we will perform a random split of the data.

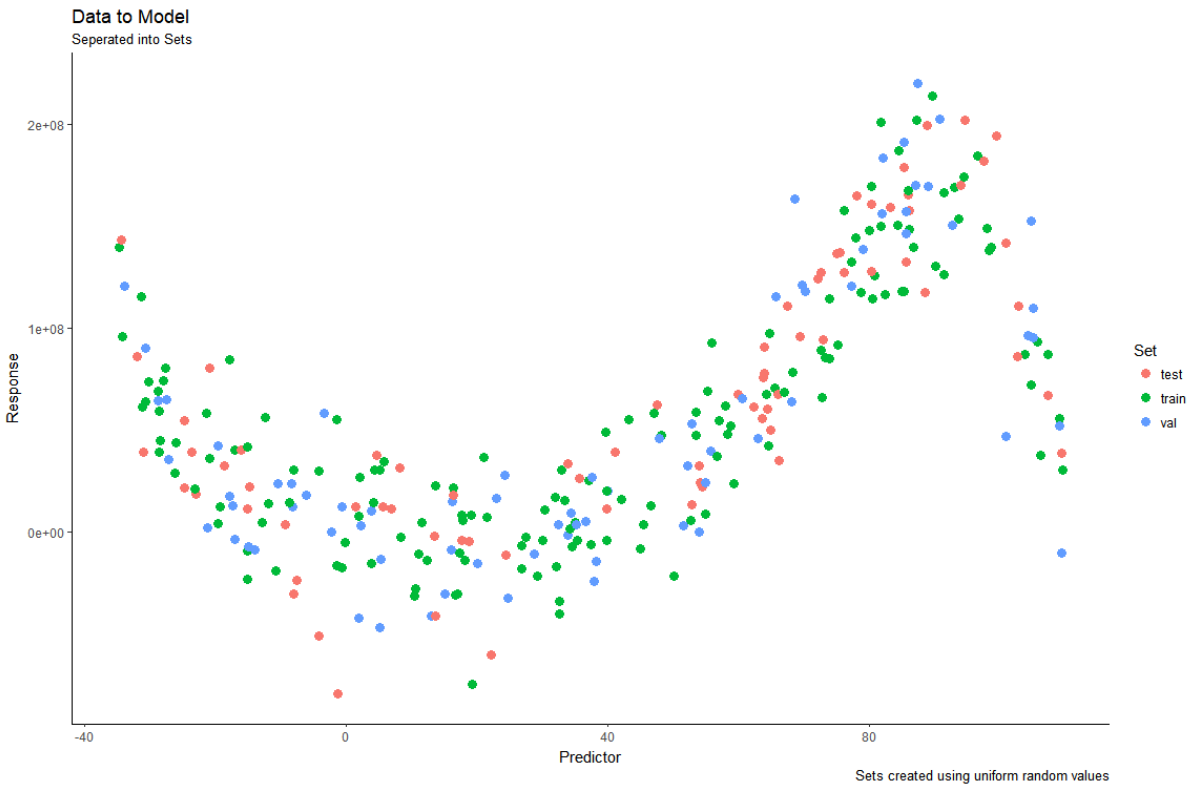
```r
# Create Data Subsets ------------------------------------------------------

  ##So we want to create 3 random samples of data

  N     <- nrow(all.df)
  idval <- sample(N)

  train.ID <- idval[0:(N*0.5)]
  val.ID   <- idval[((N*0.5)+1):(N*0.75)]
  test.ID  <- idval[((N*0.75)+1):N]

  train.df <- all.df[train.ID,]
  val.df   <- all.df[val.ID,]
  test.df  <- all.df[test.ID,]


      #Order the Data Frame
      train.df <- train.df[order(train.df$X),]
      val.df   <- val.df[order(val.df$X),]
      test.df  <- test.df[order(test.df$X),]
```

Visualise the Data

It isn't always possible to visualise the data, however where it is possible (by using 3 spatial dimensions, colour, shapes, tranperancy etc.) it should be done.

In this case the data is only two dimensional so it can be visualised:

```
all.df$sample            <- FALSE
  all.df[train.ID,]$sample <- "train"
  all.df[val.ID,]$sample   <- "val"
  all.df[test.ID,]$sample  <- "test"

 head(all.df)
##   X     input    output sample
## 1 1 -16.99955  -3846633  train
## 2 2  79.99105 148007979  train
## 3 3  17.62219  -4311459  train
## 4 4 -19.50413   3852745  train
## 5 5  93.00449 168974952    val
## 6 6  34.37293   9399792  train

 all.plot <- ggplot(all.df, aes(x = input)) +
   geom_point(data = all.df, size = 3, aes(y = output, col = sample)) +
   theme_classic() +
   labs(x = "Predictor", y = "Response", col = "Set", title = "Data to Model",
        subtitle = "Seperated into Sets",
        caption = "Sets created using uniform random values")
 #  scale_color_brewer(palette="Pastel2")
```



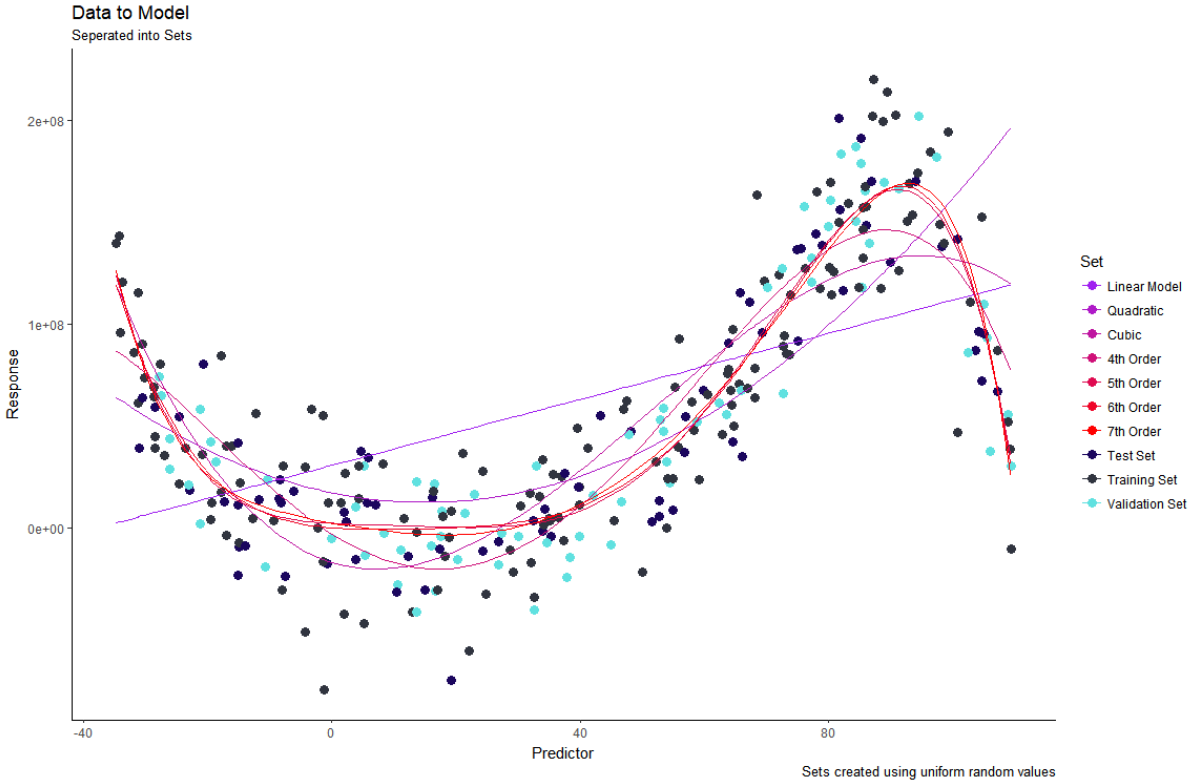
Model Training

Training a model refers to fitting models to the set of training data. We don't know which model will be appropriate for this data, but the data appears to have a natural curve so we will fit a few polynomial models and see how they perform:

> Be aware, that when making models `~I(X^n) or ~ poly(x, n)` can both be used, it has something to do with correlated coefficients and orthogonal coefficients; but I couldn't figure out which to use and why.

```
#train models of varying complexity
 # attach(train.df)
 head(train.df)

 mod.lm <- lm(output ~ input, data = train.df)
 mod.p2 <- lm(output ~ I(input^2) + input ,data = train.df)
 mod.p3 <- lm(output ~ I(input^3) + I(input^2) + input, data = train.df)
 mod.p4 <- lm(output ~ I(input^4) + I(input^3) + I(input^2) + input, data = train.df)
 mod.p5 <- lm(output ~ I(input^5) + I(input^4) + I(input^3) + I(input^2) + input, data
= train.df)
 mod.p6 <- lm(output ~ I(input^6) + I(input^5) + I(input^4) + I(input^3) + I(input^2)
+ input, data = train.df)
 mod.p7 <- lm(output ~ I(input^7) + I(input^6) + I(input^5) + I(input^4) + I(input^3)
+ I(input^2) + input, data = train.df)


  #Create Predictions and throw everything into a list
 train.df <- data.frame(train.df,
                        mod_lm = predict(mod.lm),
                        mod_p2 = predict(mod.p2),
                        mod_p3 = predict(mod.p3),
                        mod_p4 = predict(mod.p4),
                        mod_p5 = predict(mod.p5),
                        mod_p6 = predict(mod.p6),
                        mod_p7 = predict(mod.p7)
                        )[order(train.df$X),]

 train.mod.df <- melt(data = train.df, id.vars = c("X", "input", "output"))

  train.mod.list <- list(mod.lm, mod.p2, mod.p3, mod.p4, mod.p5, mod.p6, mod.p7)

  training.list <- list(models = train.mod.list, training_data = train.df,
model_predictions = train.mod.df )

  #Visualise the Models
    ##Create label and colour vectors
     plot.labels <- c("Linear Model", "Quadratic", "Cubic", "4th Order",
                  "5th Order", "6th Order", "7th Order",
                  "Test Set", "Training Set", "Validation Set")

     colfunc <- colorRampPalette(c("purple", "Red"))
     mycol <- c(colfunc(7), "#1B065E", "#30343F", "#60E1E0")
     plot(rep(1,length(mycol)),col=mycol,pch=19,cex=3)

     ##Plot the models over the Data
      all.plot +
        geom_line(data = train.mod.df, aes(x = input, y = value, col = variable)) +
        scale_color_manual(values = mycol, labels = plot.labels)
```

This returns the following plot of the models:



Hence, the question becomes, which model is the appropriate one to use?

Training and Validation Error

The root mean square error $\left(RMSE = \sqrt{\frac{1}{n} \times \sum_{i=1}^{n}[(y_i - \hat{y}_i)]^2}\right)$ is a measurement of the expected error from a data point to the model value.

It can be used, thusly, to measure model performance.

However, a more complicated model will always perform better on the data it was trained on:

> Imagine a polynomial of the 100th order, it could turn to intersect with every other data point, or, in a more abstract sense, imagine an explanation that is so convoluted that it couldn't not describe observation.

This is addressed by assessing the model performance on data that was not used to create the model, this set is called the validation data.

> In this case a 100th order polynomial would not explain the data it hadn't seen very well, because the model wouldn't generalise the shape very well and all the turns would only add to the model error (i.e. there is a saturation point) .
>
> In a more abstract sense, a good way to test convoluted theories is to test their capacity to predict future events, e.g. religion can explain the origin of species but it had failed to predict the degree of random mutation (think viruses, bacteria or even some mammalian species) regularly observed in species, this is good evidence that the model may be overly convoluted and that there is a simpler explanation.

Training Error

The training Error must be calculated for each model, this can be achieved thusly:

```
train.rmse.lm <- sqrt(sum((
    train.df$output-predict(mod.lm, newdata = train.df))^2
    )/nrow(train.df))
  train.rmse.p2 <- sqrt(sum((
    train.df$output-predict(mod.p2, newdata = train.df))^2
    )/nrow(train.df))
  train.rmse.p3 <- sqrt(sum((
    train.df$output-predict(mod.p3, newdata = train.df))^2
    )/nrow(train.df))
  train.rmse.p4 <- sqrt(sum((
    train.df$output-predict(mod.p4, newdata = train.df))^2
    )/nrow(train.df))
  train.rmse.p5 <- sqrt(sum((
    train.df$output-predict(mod.p5, newdata = train.df))^2
    )/nrow(train.df))
  train.rmse.p6 <- sqrt(sum((
    train.df$output-predict(mod.p6, newdata = train.df))^2
    )/nrow(train.df))
  train.rmse.p7 <- sqrt(sum((
    train.df$output-predict(mod.p7, newdata = train.df))^2
    )/nrow(train.df))
```

Although, because:

1. A mistake may be made when calculating the rmse for other sets
2. This is laborious
3. This is incompatible with loops

It is advisable to use a function to calculate the rmse:

```
rmse <- function(dframe, model){

    y     <- dframe[,names(dframe) == "output"]
    y.hat <- predict(object = model, newdata = dframe)
    resid <- y-y.hat
    RSS   <- sum(resid^2)
    T.Error <- RSS/nrow(dframe)
    RMSE <- sqrt(T.Error)

       return(RMSE)
  }
```

Now we can use a loop to calculate the RMSE values, this is important because:

1. For a large number of models, it won't be possible to write out every line (even with a function to expedite the process)
2. It increases the probability of a transcription error, a loop isn't immune to errors, but atleast they will be consistent

```
##Use a for loop to calculate all RMSE Values
    ##Create a data frame to store the values
       error.df <- data.frame(matrix(
         ncol = 4, nrow = length(training.list$models)))
       colnames(error.df) <- c("Order",
                               "Training Error",
                               "Validation Error",
                               "Test Error")

    ##Execute the loop
  for (i in 1:7) {

    rmse.val <- rmse(train.df, training.list$models[[i]])
```

```
  error.df[i,] <- c(order = i,
                    training_error = rmse.val,
                    validation_error = NA,
                    test_error = NA)
  }
```

Validation Error

The validation error can be calculated by reusing the function and loop:

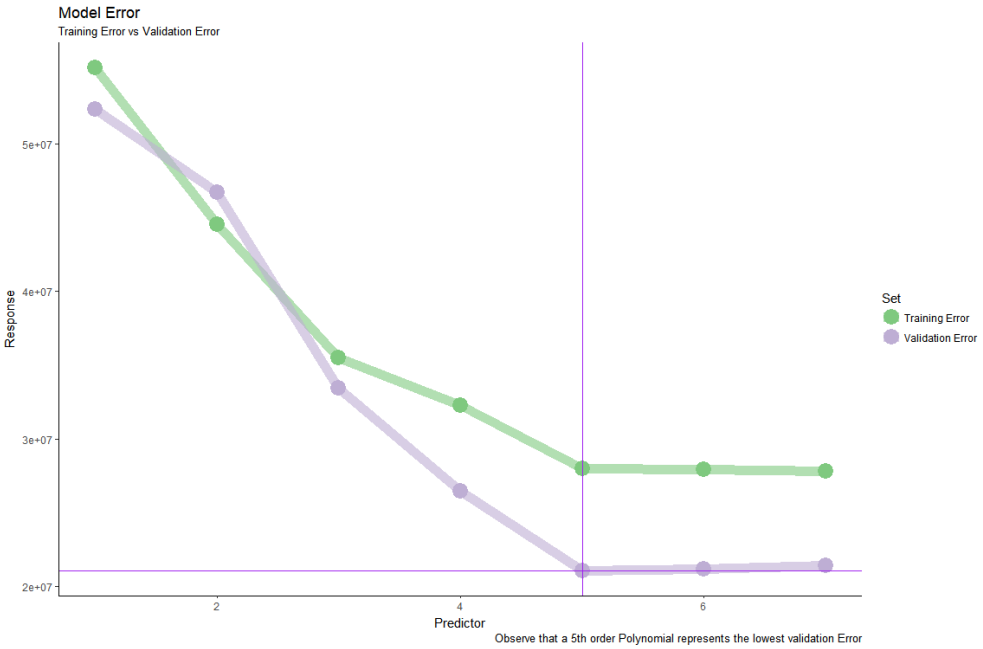
```
# Calculate the Validation Errors ----------------------------------------

  ##Use a for loop to calculate all validation error values

    #use the error.df data frame from before
  for(i in 1:7){
    rmse.val <- rmse(val.df, training.list$models[[i]])
    error.df[i,3] <- rmse.val
  }
      error.df
```

This returns the following Errors:

```
> head(error.df)
  Order Training Error Validation Error Test Error
1     1       55128411         52332313         NA
2     2       44557457         46697399         NA
3     3       35515482         33472469         NA
4     4       32259954         26480824         NA
5     5       27984169         21099237         NA
6     6       27932759         21207056         NA
```

These Errors can be plotted:

```
# Plot the Errors
      error.df.melt <- melt(error.df[,-4], id.vars = "Order", variable.name = "Set",
value.name = "Error")
      error.df.melt

      #Mininmum Validation Error
      min.val.error.y <- min(error.df$`Validation Error`)
      min.val.error.x <- error.df[
        error.df[,names(error.df)=="Validation Error"
                 ]==
          min(error.df$`Validation Error`),]$Order

 error.plot <- ggplot(error.df.melt, aes(x = Order, y = Error, col = Set)) +
   geom_line(size = 4, alpha = 0.6) +
   geom_point(size = 6) +
   theme_classic() +
   labs(x = "Predictor", y = "Response", col = "Set", title = "Model Error",
        subtitle = "Training Error vs Validation Error",
        caption = "Observe that a 5th order Polynomial represents the lowest
validation Error") +
   geom_vline(xintercept = min.val.error.x, col = "Purple") +
   geom_hline(yintercept = min.val.error.y, col = "Purple") ; error.plot +
   scale_color_brewer(palette="Accent")
```



If the Training Error continues to decrease, whilst the true error increases, it can be determined that models of such complexity have been overfit.

Hence it can be concluded that the appropriate model for this data is a 5^{th} order polynomial, because such a model has the lowest validation error.

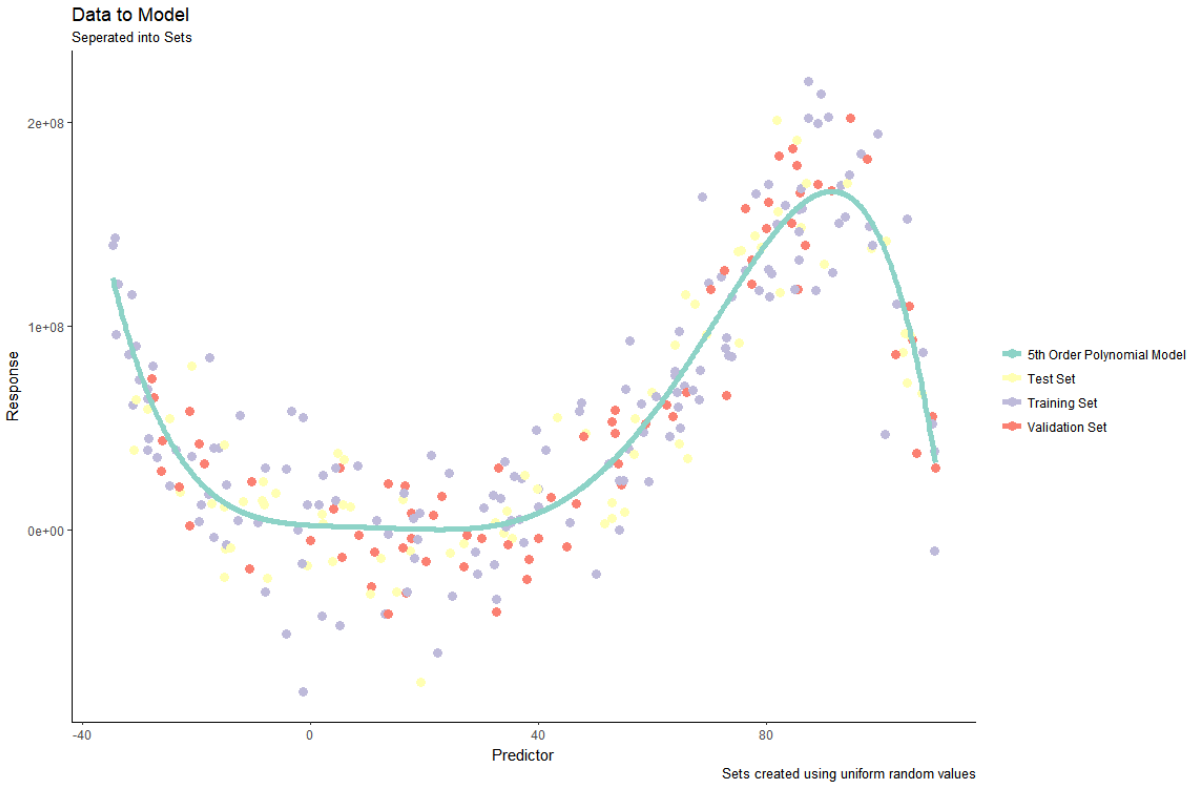
Assess the Model

Graphical Plot

The model can be plotted over all the data,

Although, it might be necessary to predict more data points to get a smoother curve over the points:

```
all.plot +
   geom_line(data = train.df, aes(y = mod_p5, col = "Purple"),
             size = 7, alpha = 0.6) +
   scale_color_brewer(palette="Set3", labels = c("5th Order Polynomial Model",
                                                 "Test Set",
                                                 "Training Set",
                                                 "Validation Set")) +
   guides(col = guide_legend(title = NULL))

##Make a slightly smoother plot
 p5.mod <- data.frame(input = seq(from = min(all.df$input),
                                  to = max(all.df$input), length.out = 1000))
 p5.mod$mod_p5 <- predict(object = mod.p5, newdata = p5.mod)

 all.plot +
   geom_line(data = p5.mod, size = 2, alpha = 1, aes(y = mod_p5,
                                                     col = "purple")) +
   scale_color_brewer(palette="Set3",
                      labels = c("5th Order Polynomial Model",
                                 "Test Set",
                                 "Training Set",
                                 "Validation Set")) +
   guides(col = guide_legend(title = NULL))
```



RMSE

The rmse and standard deviation of the test data is:

```
> rmse(test.df, mod.p5)
[1] 24154642
> sd(test.df$output)
[1] 61435612
> rmse(test.df, mod.p5)/sd(test.df$output)
[1] 0.39317
```

This provides, that the expected error from the model to a data point is 24×10^6, this may seem large, but is only 39% the standard deviation of the output data.

This implies that the model is better at predicting the data than merely taking the average value

Viewing the Underlying Data

The data for this model, was formed using the following function:

```
y <- -2685 - 121497.4*x + 11226.49*x^2 - 1000*x^3 + 30.4*x^4 - 0.2*x^5 +
rnorm(length(x),0,noise)
```

Although the coefficients are different, at least the degree of the polynomial is correct.