Pokemon Classification

Nirmal Sai Swaroop Janapaneedi

11/08/2020

Introduction

Pokemon is a global multimedia brand spanning games, TV, movies, books and others. Originally introduced to the world as a pair of games for the Nintendo Gameboy in 1996, it has since spawned 17 sequels (in the main series) across eight generations of game. Pokemon is portmanteau of Pocket Monster - the original name of the franchise - which effective describes the main feature of the game; collecting, training and battling a collection of monsters. In the original Game there were 150 Pokemon, with a bonus Pokemon available from Nintendo events. In the subsequent games this has grown to 890, with numerous variations and mega-evolutions increasing that number depending on the selection criteria. Amongst all the Pokemon they can be categorised in many ways. One of the ways they are categorised is into four status subsets; Legendary, Mythical, Sub Legendary and Normal. These classifications are a reflection of the power and rarity and other factors of the individual Pokemon.

This project aims to select a machine learning model with which to predict the status of a Pokemon based upon various data. The project was selected for several reasons:

1. The Data available was not perfect, which allowed me to practice some data wrangling.
2. The problem is a classification problem, which I would like to work on post course, so an opportunity to gain some experience.
3. Being a classification, it is a multiple classification problem rather than binary which increases the complexity somewhat.
4. The dataset is large enough in both features and data points to be a valuable machine learning exercise, but manageable for the hardware I have available.

Methods and Analysis

Preparing the dataset

Our dataset is sourced from Kaggle, available here. Due to need to need for Kaggle login dataset is provided in my Git for this project, available here. To begin we will read the csv file into a data frame and have an initial look at the data.

```
#read csv into dataframe
fulldataset <- read.csv(file = "datasets_593561_1119602_pokedex_(Update_05.20).csv")

#check numbers of rows and columns
ncol(fulldataset)
nrow(fulldataset)

#  check the dataset names and classes
str(fulldataset)
```

We can see that there are 26 columns and 1028 rows in the dataset. The data is a mixture of factors, integers and numeric values. We will next remove some unneeded columns, the alternative language names and the X columns which is a representation of row number. The columns "against_x" refer to in game damage multipliers and are drawn from the Pokemon types fields, as such they are unneeded for our predictions.

```
#remove unneeded columns X - a holdover column number, german and japanese names, against_x values
fulldataset$X <- NULL
fulldataset$german_name <- NULL
fulldataset$japanese_name <- NULL
fulldataset[31:48] <- NULL
```

Looking to the abilities columns and using https://bulbapedia.bulbagarden.net/wiki/Main_Page, abilities are able to be changed by the trainer. Hidden abilities are also either dependent upon the random personality of a Pokemon or are unique to the Pokemon dependent upon which information you read. As such the changeability of these and lack of wider clarity on hidden abilities means they will not translate well to a general predictive model and will be removed.

```
fulldataset$abilities_number <- NULL
fulldataset$ability_1 <- NULL
fulldataset$ability_2 <- NULL
fulldataset$ability_hidden <- NULL
```

The Species column looks to contain some odd entries:

```
fulldataset$species[1:10]
```

```
##  [1] Seed PokÃ©mon        Seed PokÃ©mon        Seed PokÃ©mon
##  [4] Seed PokÃ©mon        Lizard PokÃ©mon      Flame PokÃ©mon
##  [7] Flame PokÃ©mon       Flame PokÃ©mon       Flame PokÃ©mon
## [10] Tiny Turtle PokÃ©mon
## 641 Levels: Abundance PokÃ©mon Acorn PokÃ©mon Alloy PokÃ©mon ... Zen Charm PokÃ©mon
```

It seems that the accented 'e' in Pokemon has been formatted into a different character. Looking into the actual content of this column over half of the dataset has unique entries. Focusing on the less prevalent of our target variables we see this ratio is consistent throughout. Given the prevalence for unique entries, this potential feature is not expected to translate to a general model and will be removed from our dataset

```
fulldataset %>% group_by(species) %>%
  summarise(no=n()) %>%
  group_by(no) %>%
  summarise(n())
```

```
## # A tibble: 10 x 2
##       no `n()`
##    <int> <int>
##  1  11412     1
##  2  22151     1
##  3      3    39
##  4      4    23
##  5      5     6
##  6      6     5
##  7      7     1
```

```
##  8       8      1
##  9       9      2
## 10      12      1
```

```
fulldataset %>%
  filter(status%in%c("Legendary", "Mythical", "Sub Legendary")) %>%
  group_by(status,species) %>%
  summarise(no=n()) %>%
  group_by(no) %>%
  summarise(n())
```

```
## # A tibble: 4 x 2
##          no `n()`
##       <int> <int>
## 1         1    54
## 2         2    15
## 3         3     3
## 4         4     5
```

```
fulldataset$species <- NULL
```

When we look at the egg data, we find a similar issue. Various Pokemon sources including the previously linked wiki, state that egg information is inconsistent and not actually displayed in any game, only "mentioned canonically" in spin-off titles. As such the data in these field is deemed to be unreliable and will not be used.

```
fulldataset$egg_type_1 <- NULL
fulldataset$egg_type_2 <- NULL
fulldataset$egg_type_number <- NULL
```

Now we have removed the unneeded columns we will look a bit deeper and find NA entries, then decide how we will handles these. this little code will enable us to quickly check for NA entries:

```
check_na <- function(x){
  any(is.na(x))
}

check_na(fulldataset)
```

```
## [1] TRUE
```

As this returns true we will use summary to look more closely:

```
summary(fulldataset)
```

```
##  pokedex_number                      name          generation
##  Min.   :  1.0   Abomasnow             :    1   Min.   :1.000
##  1st Qu.:213.8   Abra                  :    1   1st Qu.:2.000
##  Median :433.5   Absol                 :    1   Median :4.000
##  Mean   :437.7   Accelgor              :    1   Mean   :4.034
##  3rd Qu.:663.2   Aegislash Blade Forme :    1   3rd Qu.:6.000
##  Max.   :890.0   Aegislash Shield Forme:    1   Max.   :8.000
```

```
##                    (Other)                   :1022
##                 status         type_number         type_1              type_2
## Legendary       : 39      Min.   :1.000   Water   :134                 :486
## Mythical        : 29      1st Qu.:1.000   Normal  :115     Flying :109
## Normal          :915      Median :2.000   Grass   : 91     Fairy  : 41
## Sub Legendary: 45         Mean   :1.527   Bug     : 81     Ground : 39
##                           3rd Qu.:2.000   Psychic: 76      Poison : 38
##                           Max.   :2.000   Fire    : 65     Psychic: 38
##                                           (Other):466        (Other):277
##      height_m            weight_kg          total_points          hp
## Min.    :   0.100    Min.    :     0.10   Min.   : 175.0   Min.   :  1.00
## 1st Qu.:   0.600    1st Qu.:     8.80   1st Qu.: 330.0     1st Qu.: 50.00
## Median :   1.000    Median :    28.50   Median : 455.0   Median : 66.50
## Mean   :   1.368    Mean   :    69.75   Mean   : 437.6     Mean   : 69.58
## 3rd Qu.:   1.500    3rd Qu.:    69.10   3rd Qu.: 510.0     3rd Qu.: 80.00
## Max.   :100.000     Max.   :   999.90   Max.   :1125.0     Max.   :255.00
##                     NA's   :1
##       attack            defense          sp_attack          sp_defense
## Min.    :   5.00    Min.    :   5.00    Min.    :  10.00    Min.    :  20.00
## 1st Qu.: 55.00      1st Qu.: 50.00      1st Qu.:  50.00      1st Qu.:  50.00
## Median : 76.00    Median : 70.00        Median :  65.00    Median :  70.00
## Mean   : 80.12      Mean   : 74.48      Mean   :  72.73      Mean   :  72.13
## 3rd Qu.:100.00      3rd Qu.: 90.00      3rd Qu.:  95.00      3rd Qu.:  90.00
## Max.   :190.00      Max.   :250.00      Max.   :194.00       Max.   :250.00
##
##       speed            catch_rate       base_friendship    base_experience
## Min.    :   5.00    Min.    :   3.00    Min.    :   0.00    Min.    :  36.0
## 1st Qu.: 45.00      1st Qu.: 45.00      1st Qu.:  70.00      1st Qu.:  67.0
## Median : 65.00    Median : 60.00        Median :  70.00    Median :159.0
## Mean   : 68.53      Mean   : 93.17      Mean   :  64.14      Mean   :153.8
## 3rd Qu.: 90.00      3rd Qu.:127.00      3rd Qu.:  70.00      3rd Qu.:201.5
## Max.   :180.00      Max.   :255.00      Max.   :140.00       Max.   :608.0
##                     NA's   :104         NA's   :104          NA's   :104
##          growth_rate  percentage_male   egg_cycles
##                    :  1   Min.    :  0   Min.    :  5.00
## Erratic            : 26   1st Qu.: 50   1st Qu.: 20.00
## Fast               : 68   Median : 50   Median : 20.00
## Fluctuating: 14           Mean   : 55   Mean   : 30.32
## Medium Fast:432           3rd Qu.: 50   3rd Qu.: 25.00
## Medium Slow:245           Max.   :100   Max.   :120.00
## Slow               :242   NA's   :236   NA's   :1
```

First we'll tackle weight which has only one NA. Looking at this data entry we will reference an external resource here . This shows the Pokemon with the missing weight is an evolution. To resolve this we will multiply the pre-evolution weight by 5, the same factor as the height increase.

```
fulldataset %>% filter(is.na(weight_kg))
```

```
##   pokedex_number      name generation  status type_number type_1
## 1            890 Eternatus Eternamax          8 Legendary           2 Poison
##   type_2 height_m weight_kg total_points  hp attack defense sp_attack
## 1 Dragon      100        NA         1125 255    115     250       125
##   sp_defense speed catch_rate base_friendship base_experience growth_rate
```

```
## 1             250     130              NA                    NA                    NA           Slow
##     percentage_male egg_cycles
## 1                NA         120
```

```
fulldataset$weight_kg <- replace_na(fulldataset$weight_kg, 950*5)
```

Looking to catch rate, base_friendship and Base_experience, all have 104 missing entries. Different sites offer different values and explanations for these variables. With no reliable consistent sources of information on these, and no way to know how these may affect our modelling if we choose either replacement with minimum , mean, or out of scope values, for example, we will remove these features. The same will apply to the percentage male with 236 NAs.

```
fulldataset[17:19] <- NULL
fulldataset$percentage_male <- NULL
```

Next we will look to egg cycles where is again one NA entry. Looking at that entry we can see that it is part of a set of Pokemon. Investigating this shows that the growth rate entry for this Pokemon is also blank; "". This can also be seen in the previous summary of the dataset. We will update the egg cycles and growth rate entries to match the other Pokemon of the set. Finally we will re-factor the growth rate column to remove the erroneous"" entry.

```
fulldataset[653,]$egg_cycles <- 20 fulldataset[653,]$growth_rate <-
"Medium Slow" fulldataset$growth_rate <-
factor(fulldataset$growth_rate)
```

Another check of the NAs shows that we now have a complete dataset. Note that the Pokemon number and names fields have been left in the dataset as they may prove useful for looking at Pokemon in exploratory data analysis but will not be used in modelling. Our tidy dataset has 1028 rows of data across 18 columns. The final step before commencing analysis is splitting the dataset into training and test datasets. The test dataset will only be used for final results. Due to relatively small dataset will not split into three sets - test, train and validation - to enable modelling and intermediate testing, and final testing respectively, and will instead use cross validation on the training set to tune models.

The data will be split using the createdatapartion function from the caret package to ensure spread of the target classifications across the sets. We will use a 9:1 training to test ratio as we want as much data to work with to train the models with only a small total dataset, and still enough to prove a valuable final test. To enable reproduction of the results will for accurate assessment we will set the seed.

```
set.seed(1, sample.kind = "Rounding")
test_index <- createDataPartition(fulldataset$status, times = 1, p = 0.1, list = FALSE)
testset <- fulldataset[test_index,]
trainset <- fulldataset[-test_index,]
```

Exploratory Data Analysis

Our newly prepared training dataset has 924 rows and 18 columns, and our test set 104 and 18. Looking at the head (split for visibility) we have:

pokedex_number	name	generation	status	type_number	type_1
1	Bulbasaur	1	Normal	2	Grass
2	Ivysaur	1	Normal	2	Grass
3	Venusaur	1	Normal	2	Grass
3	Mega Venusaur	1	Normal	2	Grass
4	Charmander	1	Normal	1	Fire
5	Charmeleon	1	Normal	1	Fire

type_2	height_m	weight_kg	total_points	hp	attack
Poison	0.7	6.9	318	45	49
Poison	1.0	13.0	405	60	62
Poison	2.0	100.0	525	80	82
Poison	2.4	155.5	625	80	100
	0.6	8.5	309	39	52
	1.1	19.0	405	58	64

defense	sp_attack	sp_defense	speed	growth_rate	egg_cycles
49	65	65	45	Medium Slow	20
63	80	80	60	Medium Slow	20
83	100	100	80	Medium Slow	20
123	122	120	80	Medium Slow	20
43	60	50	65	Medium Slow	20
58	80	65	80	Medium Slow	20

The target of our classification project is status. We will see how many of each are in our training set:

status	number of pokemon	percentage
Legendary	35	3.787879
Mythical	26	2.813853
Normal	823	89.069264
Sub Legendary	40	4.329004

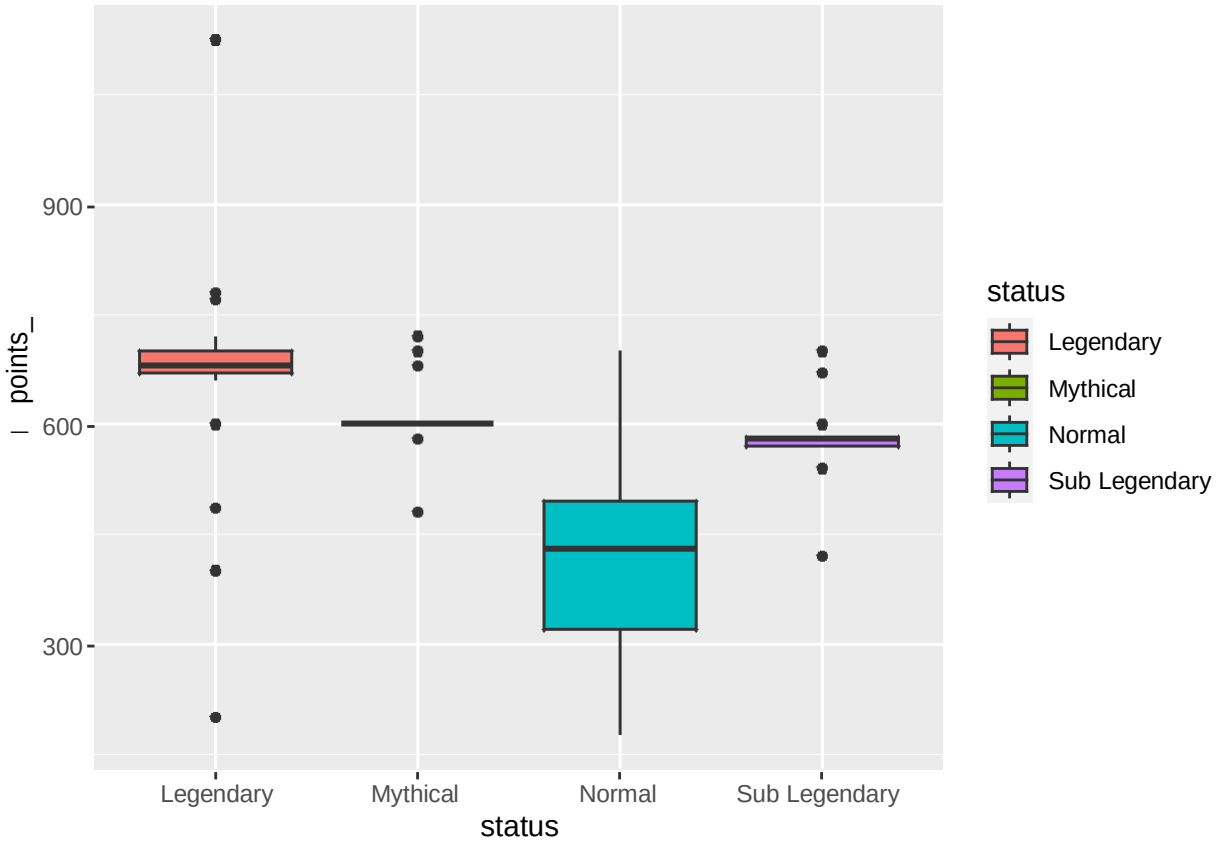
The normal category far outweighs the other categories. Mythical with the lowest number of appearances may be the most difficult to effectively train a model with. The features in the dataset, as can be seen in the head, can be generally split into two groups: descriptive [1:9] and ability [10:21]. To initial explore we will look at the mean value of the combat ability scores of the statuses and see if there is a pattern.

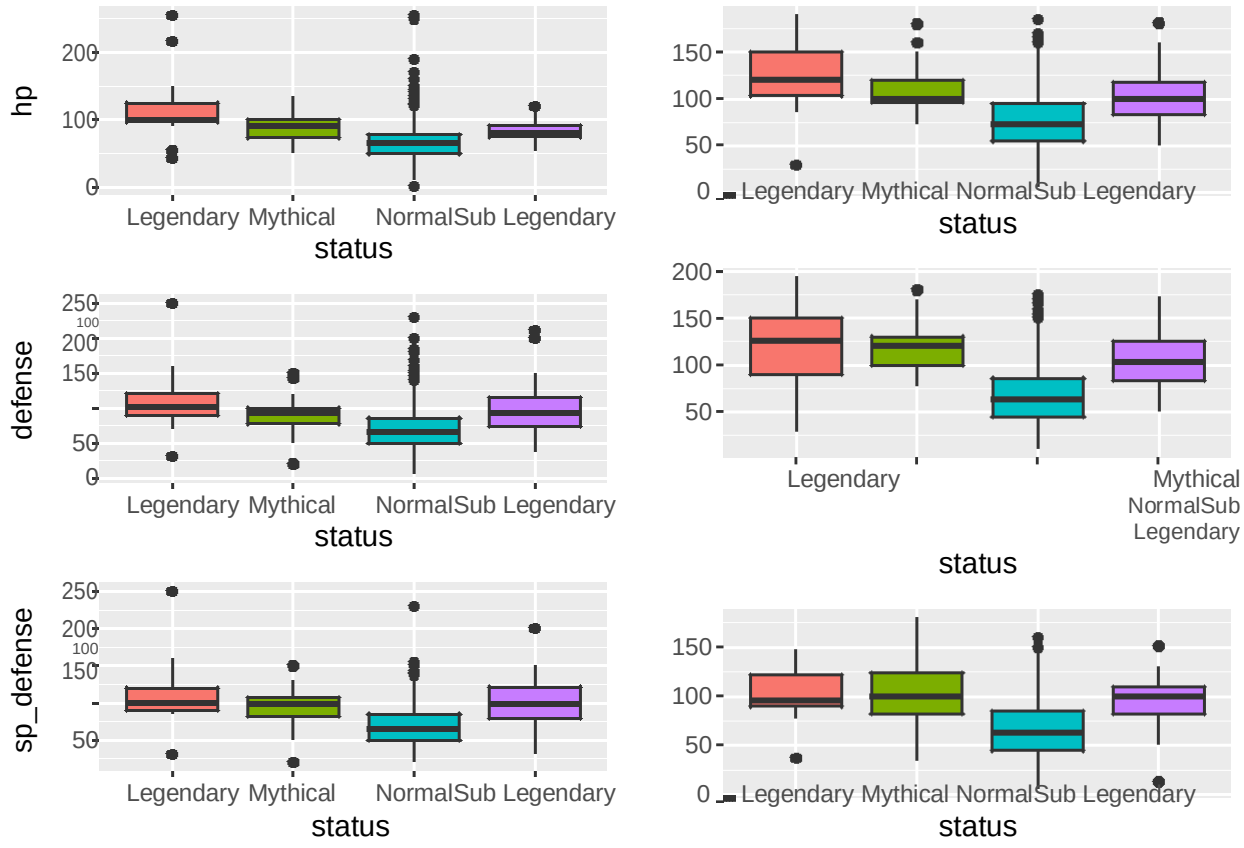
status	Mean total points	Mean hp	Mean attack	Mean defense
Legendary	674.5429	110.51429	123.02857	109.37143
Mythical	605.3846	85.23077	110.65385	92.30769
Normal	414.1312	66.14702	76.08019	71.06197
Sub Legendary	582.0000	84.10000	101.45000	99.22500

status	Mean sp_attack	Mean sp_defense	Mean speed
Legendary	119.45714	111.97143	100.20000
Mythical	118.65385	95.19231	103.34615
Normal	67.60267	68.15188	65.08748
Sub Legendary	104.30000	99.40000	93.52500

From this we can see that the mean for legendary Pokemon have, with the exception of speed, higher mean values than the other statuses, and Normal Pokemon have the lowest, with the mythical and sub legendary sharing the second and third positions across the values.

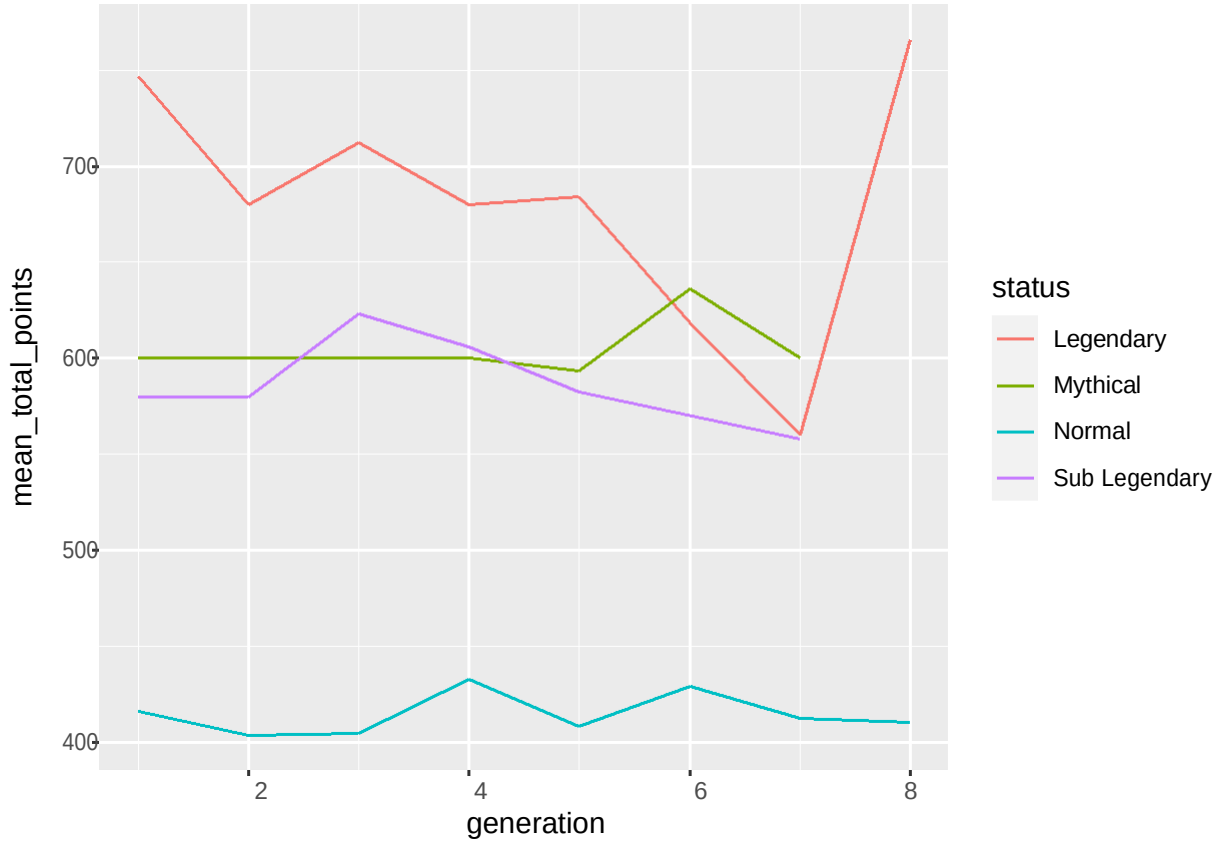
To better visualise this we will look at some graphs.





The graphs show the data from the table more clearly but highlight some outliers in the data, particularly in the normal and legendary status categories. The differentiation between the various ability scores however shows that together they may be good predictors of status.

taking this further we will look at how these have changed over the generations of Pokemon. We will focus on the total points value and separate by the generations values.

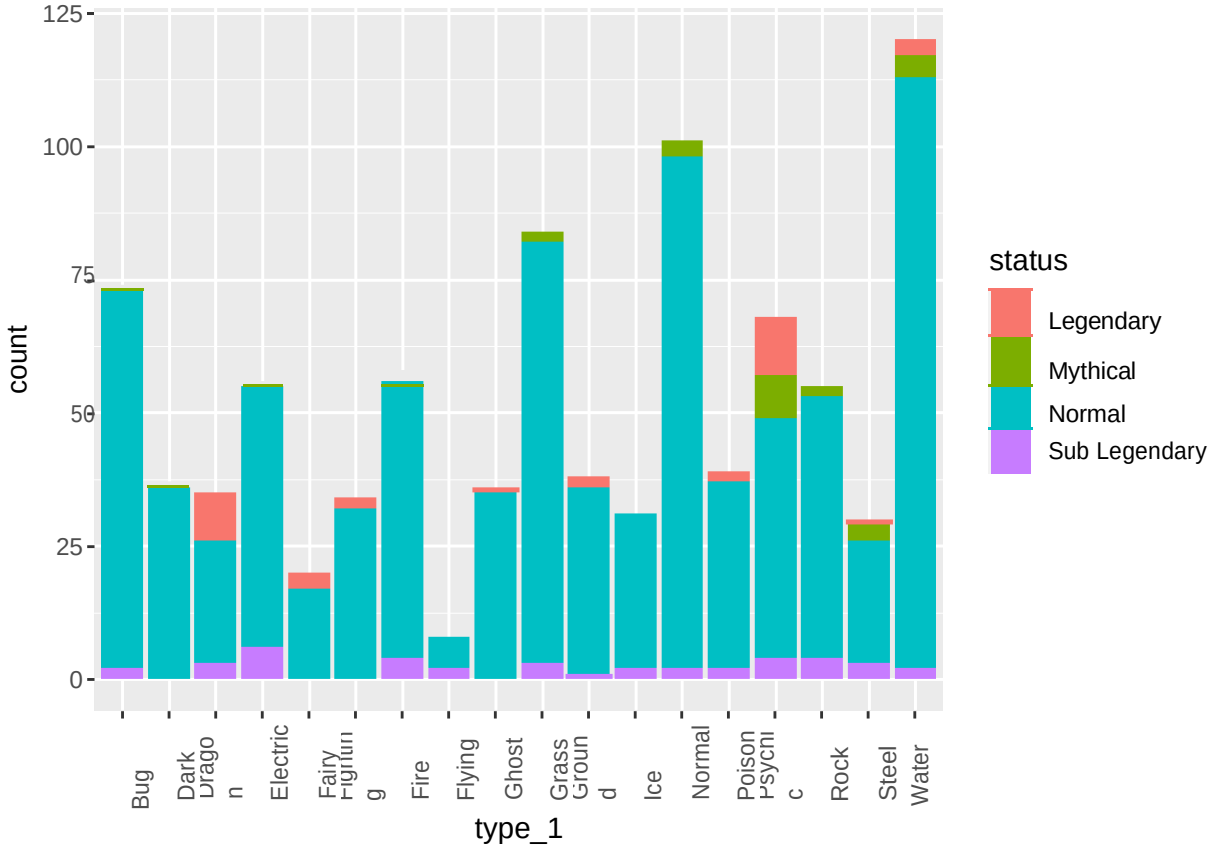


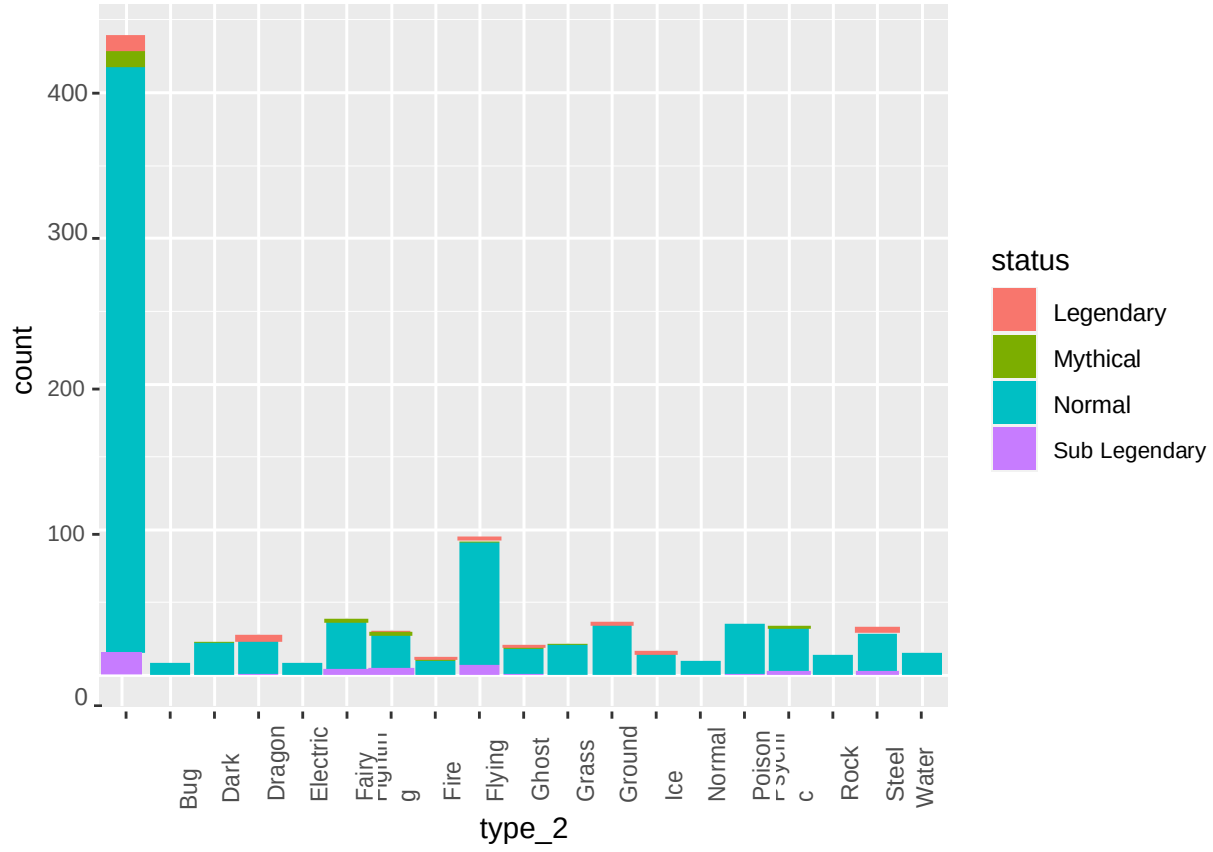
We can see that normal status is fairly consistent, the others statuses dip in gen 7, and the mean total is distinct for each generation. As such this could be a useful feature along with the stats to select which status more effectively.

Moving to a descriptive feature, we will look at type number, type one and type 2.

status	type_number	n()
Legendary	1	11
Legendary	2	24
Mythical	1	11
Mythical	2	15
Normal	1	402
Normal	2	421
Sub Legendary	1	15
Sub Legendary	2	25

The numbers are fairly evenly split across the status values. we will therefore look at the specifics of type 1 and type 2.

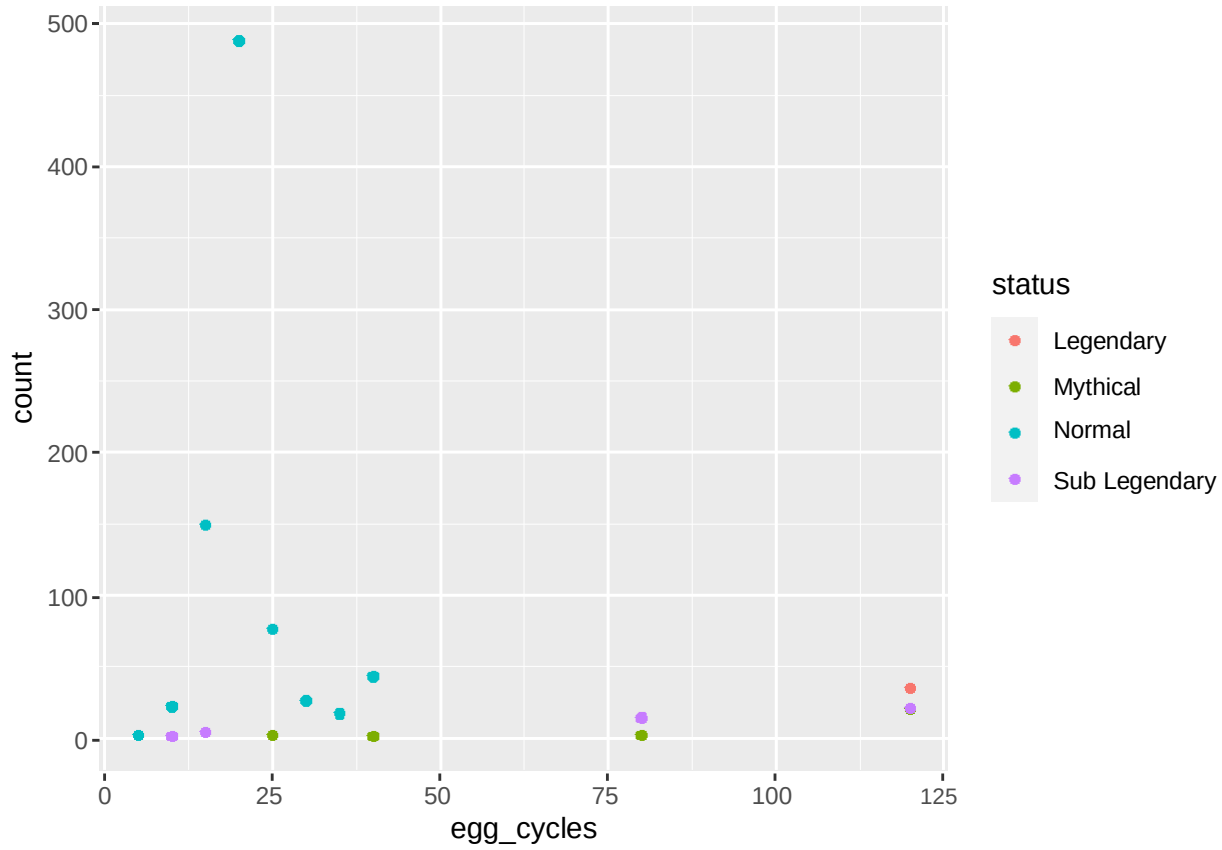

125
100
75
50
25
0
count
Bug
Dark
Dragon
Electric
Fairy
Fighting
Fire
Flying
Ghost
Grass
Ground
Ice
Normal
Poison
Psychic
Rock
Steel
Water
type_1
status
Legendary
Mythical
Normal
Sub Legendary



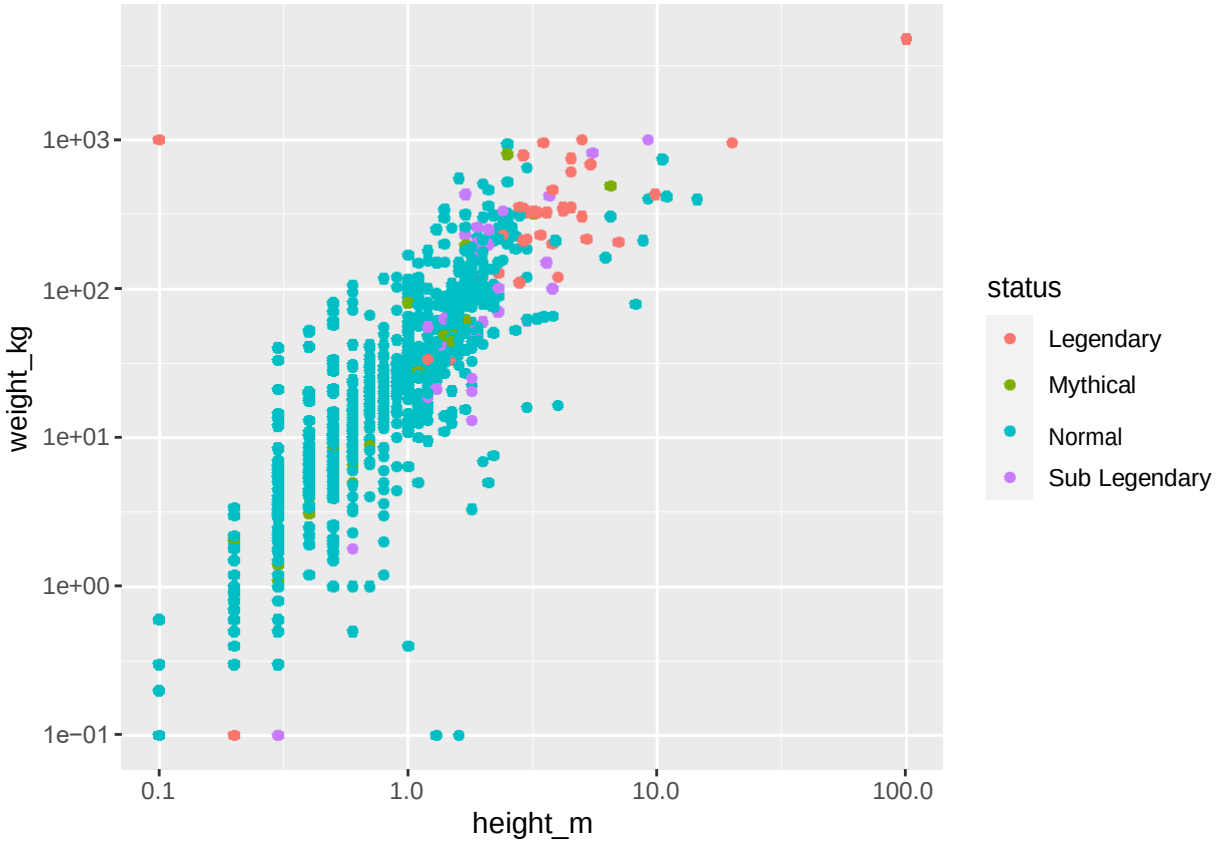
The specifics show no standout unique elements, however several of the types are only utilised with two status groupings. These therefore may be a good discriminative feature for splitting the status groups if, for example the points data cannot effectively pinpoint the status. Since the type number data offers no value over the more descriptive, we will remove this from our datasets.

```
trainset$type_number <- NULL
testset$type_number <- NULL
```

The next feature we will look at is egg cycles. From Bulbapedia we can see that an egg cycle are the amount of times a specific number of in game steps must be taken before an egg of the Pokemon hatches. Looking at the following graph we can see that higher egg cycle are reserved for non normal status Pokemon. As such this feature could be a powerful initial separator in, for example, a decision tree model.

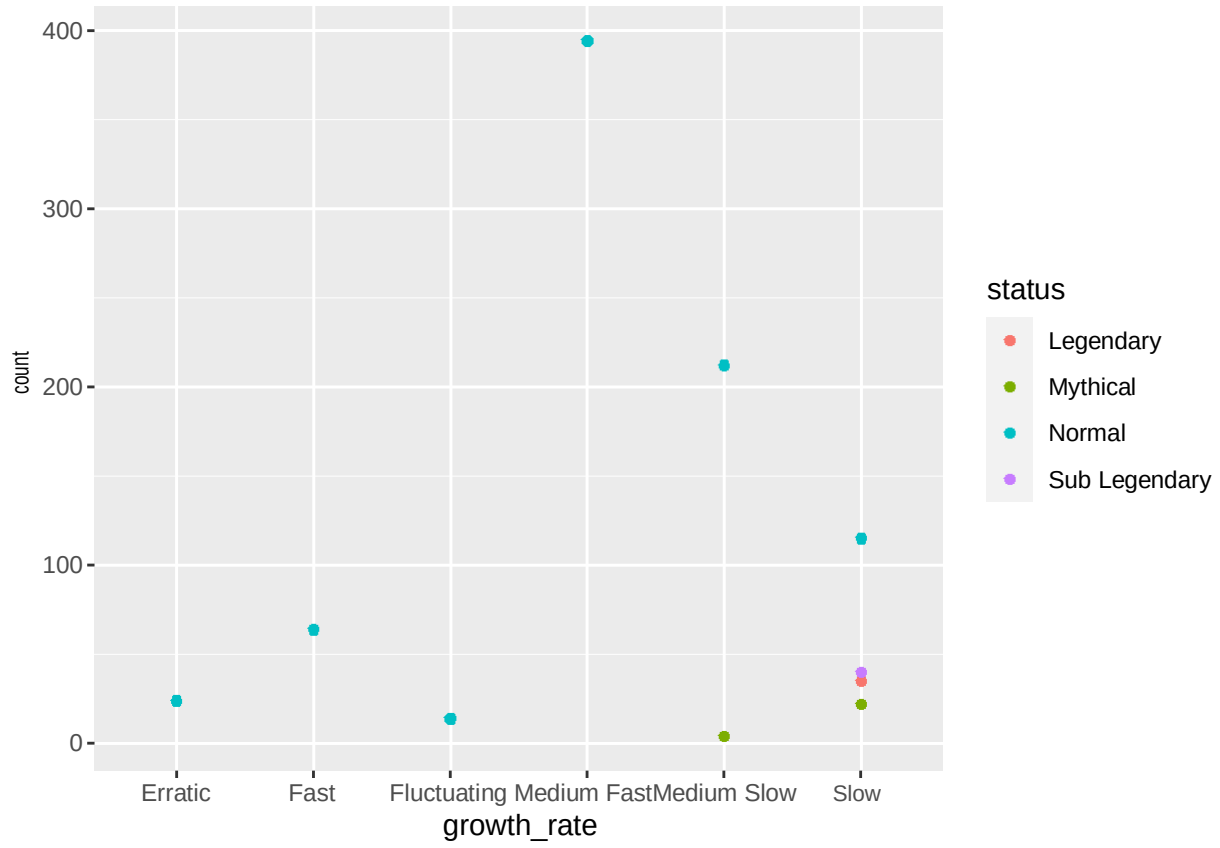


Height and weight are descriptive features of Pokemon. Across the games Pokemon come in all shapes and sizes. We will plot these against each other and color by status to see if there is a pattern.

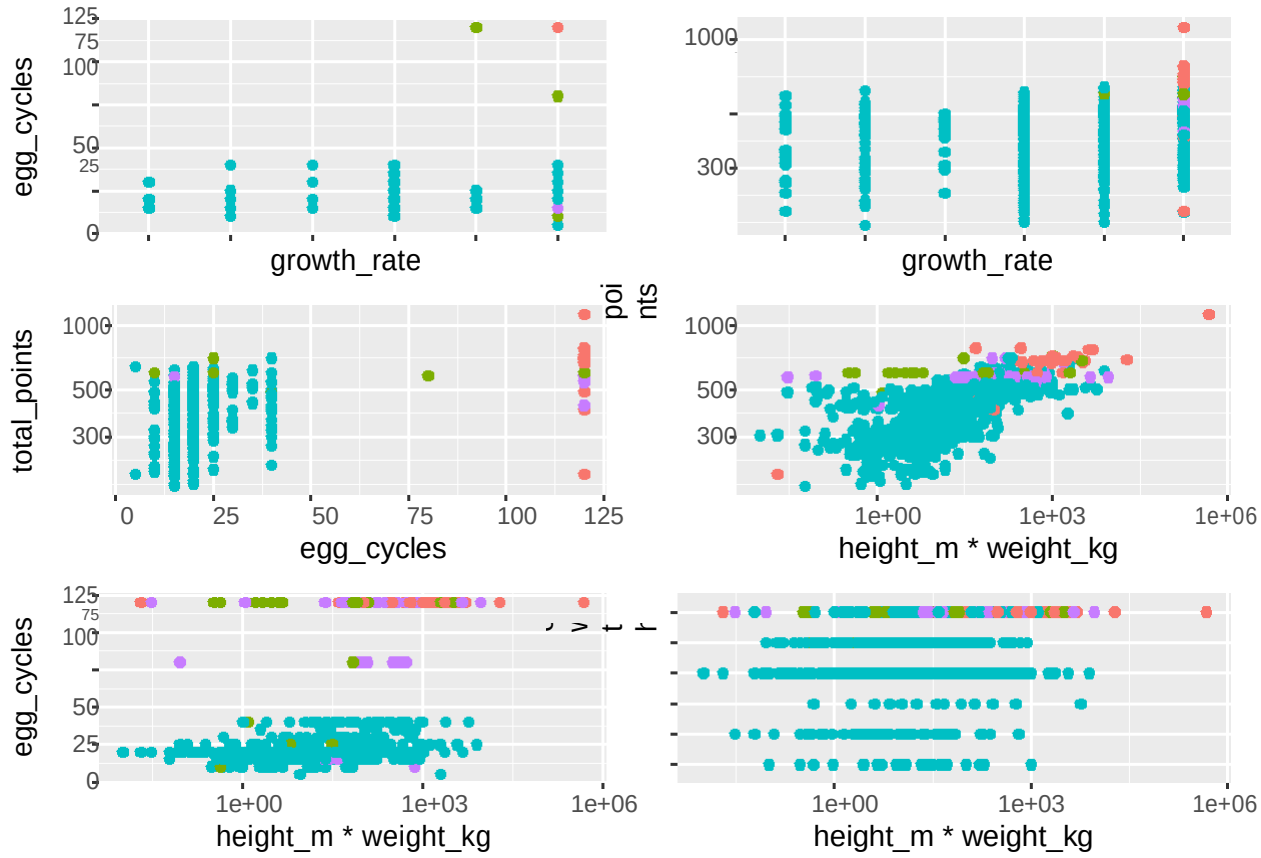


From this we can see that mythical status Pokemon are spread through the range, however there appears to be visible groupings for normal, sub legendary and legendary statuses.

The final feature we will look at independently is growth rate. Growth rate is the amount of experience that a Pokemon requires to reach level 100 in games. The following graph shows that, similar to egg cycles, non-normal Pokemon are reserved the slow and medium-slow categories.



From looking at the features independently we can see that there is potential predictive power in them. Egg cycles and growth rate for example can split effectively narrow the status groups being looked at, and the points scores then differentiate more between the resulting groups. As a final piece of exploratory analysis we will look at the following graphs which combine these key features and highlight further the groupings and correlations.



Modelling selection

This is a multiple classification task. We will explore three methods for modelling;, KNN, Random Forest(rf) and Support vector Machine(svm). The selection of these is informed by two factors 1) they are all capable of multiple classification, 2) they are from different 'families' of algorithm. KNN was chosen to display the capabilities taught in the PH125.9x course and will be tuned by k, the number of neighbours considered, RF will be used for the same reason and as it was shown to be the top performing general use case algorithm by Delgado(2014), second by Caruana(20XX), and best (in binary classifications it must be stated), by Wainer et. al (2016). The RF model will be tuned using mtry - number of random features considered at each split - and ntree. The Choice of SVM was to push myself to learn about a different method to those on the course and because in the aforementioned papers SVM were highly rated at completing this type of task. The SVM will be tried with a liner and radial kernal, and tuned using cost.

To enable knn and SVM we need to normalise the data and account for factors. This is because different scales can warp the algorithm. We will therefore perform one-hot encoding, to create unique variables of 1 or 0 for each factor and normalise the data. These values from the training data will then be used to account for the factors on the test set, and the min/max of the training data used to normalise the numeric values. This will prevent inadvertent improvement of the training data from the test set by accounting for factors and normalising individually or as a whole dataset.

Preparing for modelling

One Hot Encoding Our first step is to remove the names and pokedex number columns from the training and test datasets as they will not be used.

```
trainset$pokedex_number <- NULL
testset$pokedex_number <- NULL
trainset$name <- NULL
testset$name <- NULL
```

Next we will use the dummyVars function from the caret package to create dummy variables for our factor columns. This will essentially turn each factor into a binary variable. Then we will apply this to the training set to create a matrix removing the factor columns of the training set and inserting the binary variable columns. We will then convert the matrix to a data frame. Using the predict function we will perform the same actions on the test set.

```
dummies_model <- dummyVars(status ~ ., data=trainset)
normtrainset <- predict(dummies_model, newdata = trainset)
normtrainset <- data.frame(normtrainset)

normtestset <- predict(dummies_model, newdata = testset)
normtestset <- data.frame(normtestset)
```

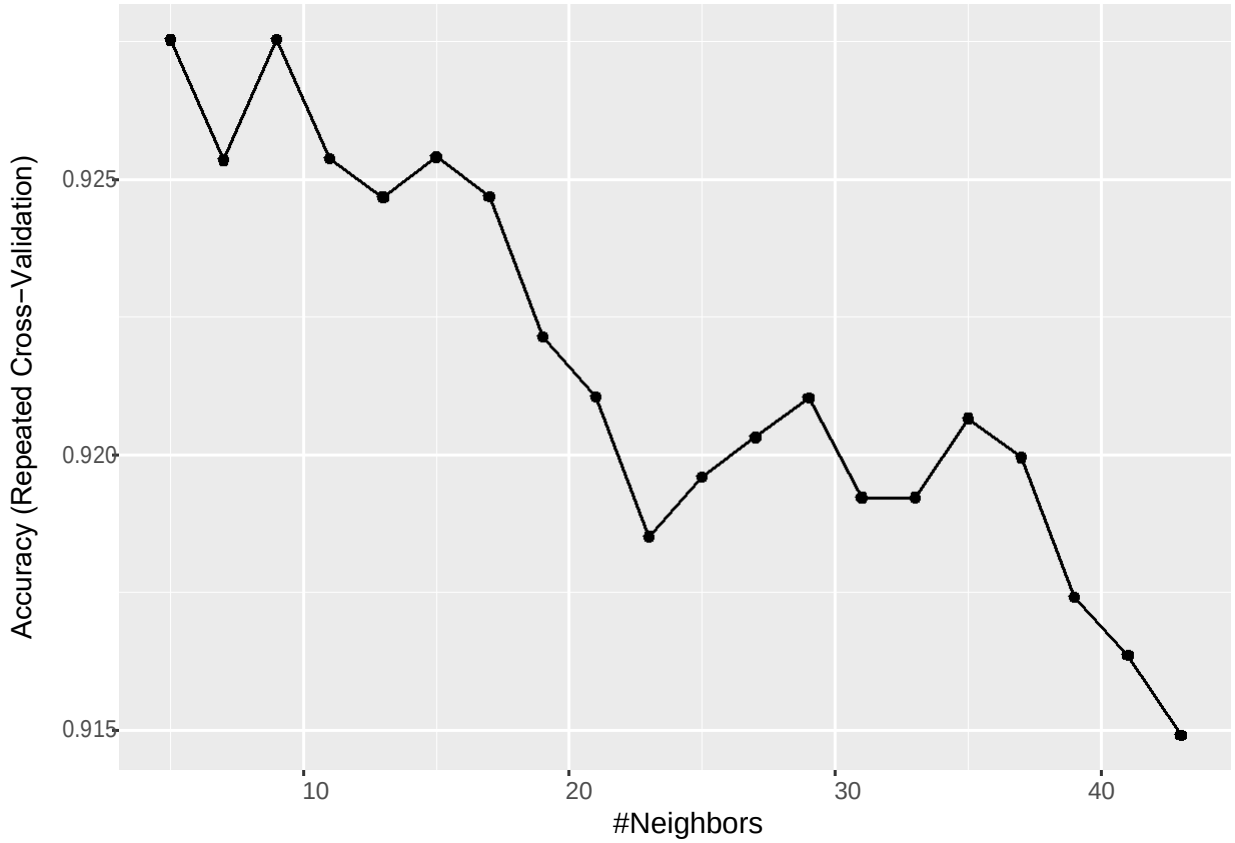
Normalisation *To Normalise the data we will use the preProcess function from the caret package. This works in a similar way to the dummyVars function. For the numeric variables in our dataset a normalisation model will be made using the min and max values. The predict function will then apply this to the training dataset and the test set. They will then be converted back to data frames. We will finally add the original status columns from the training and test datasets to their respective normalised sets.*

```
normalise_model <- preProcess(normtrainset, method='range')
normtrainset <- predict(normalise_model, newdata = normtrainset)
normtrainset <- data.frame(normtrainset)

normtestset <- predict(normalise_model, newdata = normtestset)
normtestset <- data.frame(normtestset)

normtrainset<- normtrainset %>% mutate(status=trainset$status)
normtestset <- normtestset %>% mutate(status=testset$status)
```

Modelling

KNN *For each of the models we produce we will set the seed to 2. This will enable assessors to replicate our results and a fair comparison between the models. A control variable will be used across the models which will specify 10 fold cross validation repeated three times be used to assess the accuracy of the model with the different model variables (k in this instance). For the KNN tuning we will use the tuneLength parameter of the caret train function to select 20 values of k starting with 5 and increasing in increments of 2.*

```
set.seed(2, sample.kind = "Rounding")
control <- trainControl(method="repeatedcv",number=10, repeats = 3)
knn_model <- train(status~., method="knn", data=normtrainset, trControl=control, tuneLength=20)
ggplot(knn_model)
```

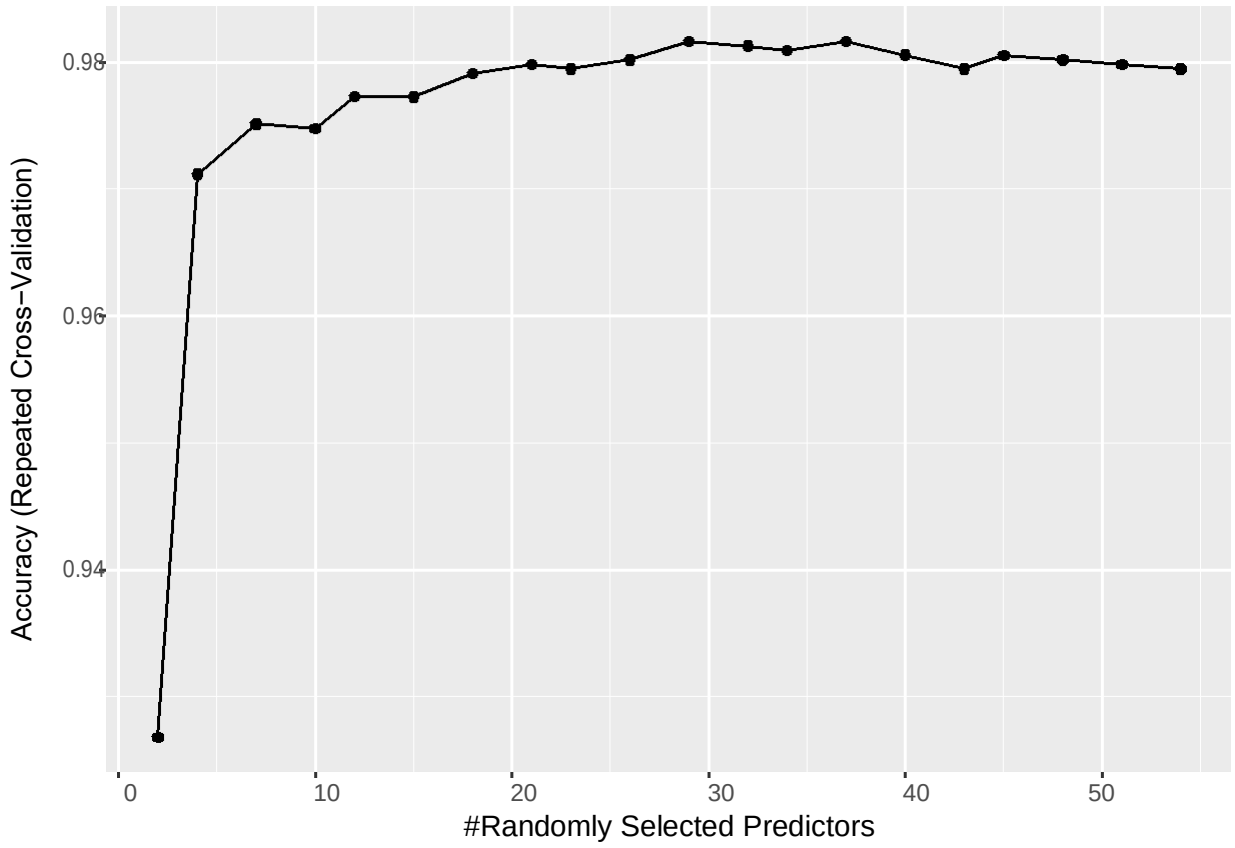



```
knn_acc <- max(knn_model$results$Accuracy)
```

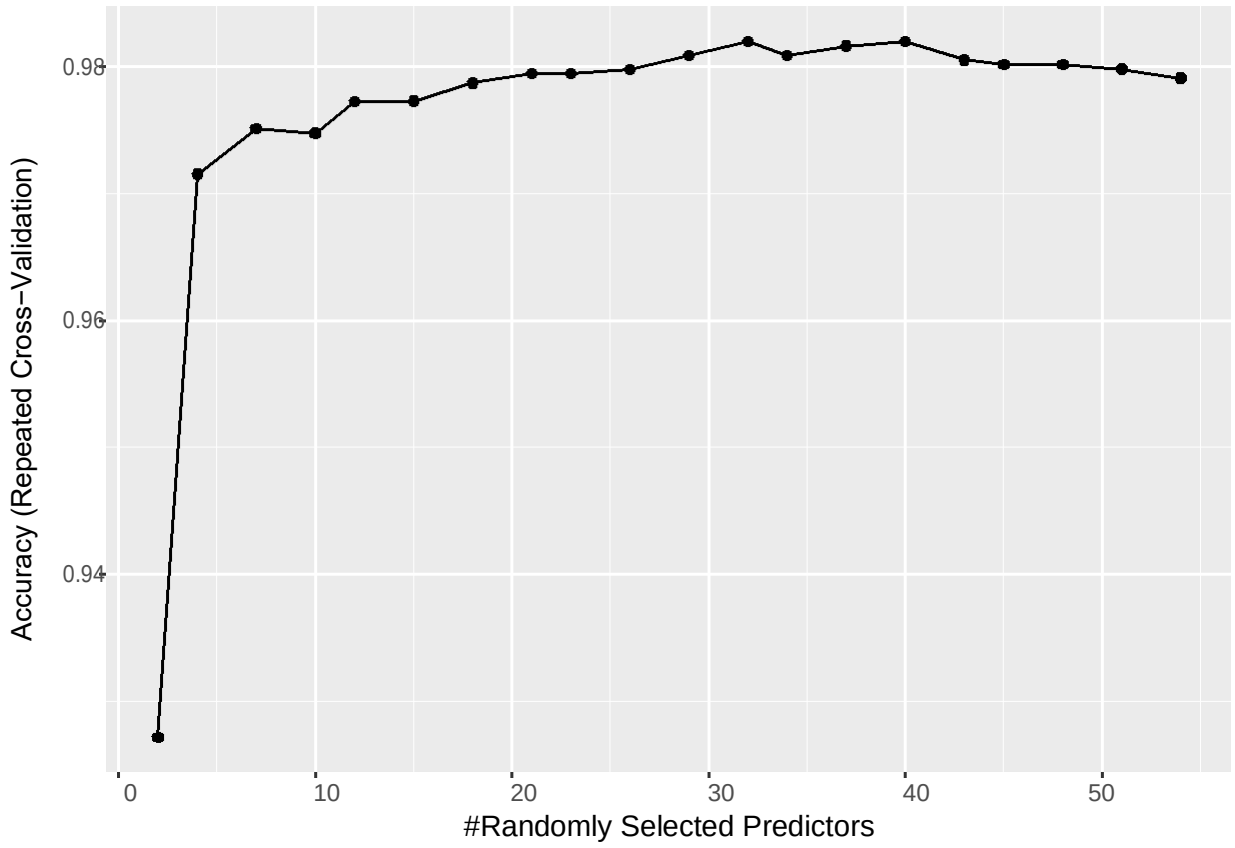
From the graph we can see that the optimal value of k for our model is 9, with an accuracy of 0.9275327.

Random Forest Random forest is a resource intensive modelling algorithm. As stated we will use the same repeated 10 fold cross validation across our models. For our random forest models we will tune mtry - the number of randomly selected variables to be considered at each point in the generated trees. Again we will use the tuneLength parameter to tune across 20 values starting at two and increasing by 3 each time. This will be repeated three times, the second and third iterations setting an ntree value - the number of random trees generated - of 1000 and 2000. The default for the first model is 500. From these we will have all combinations of tuning parameters.

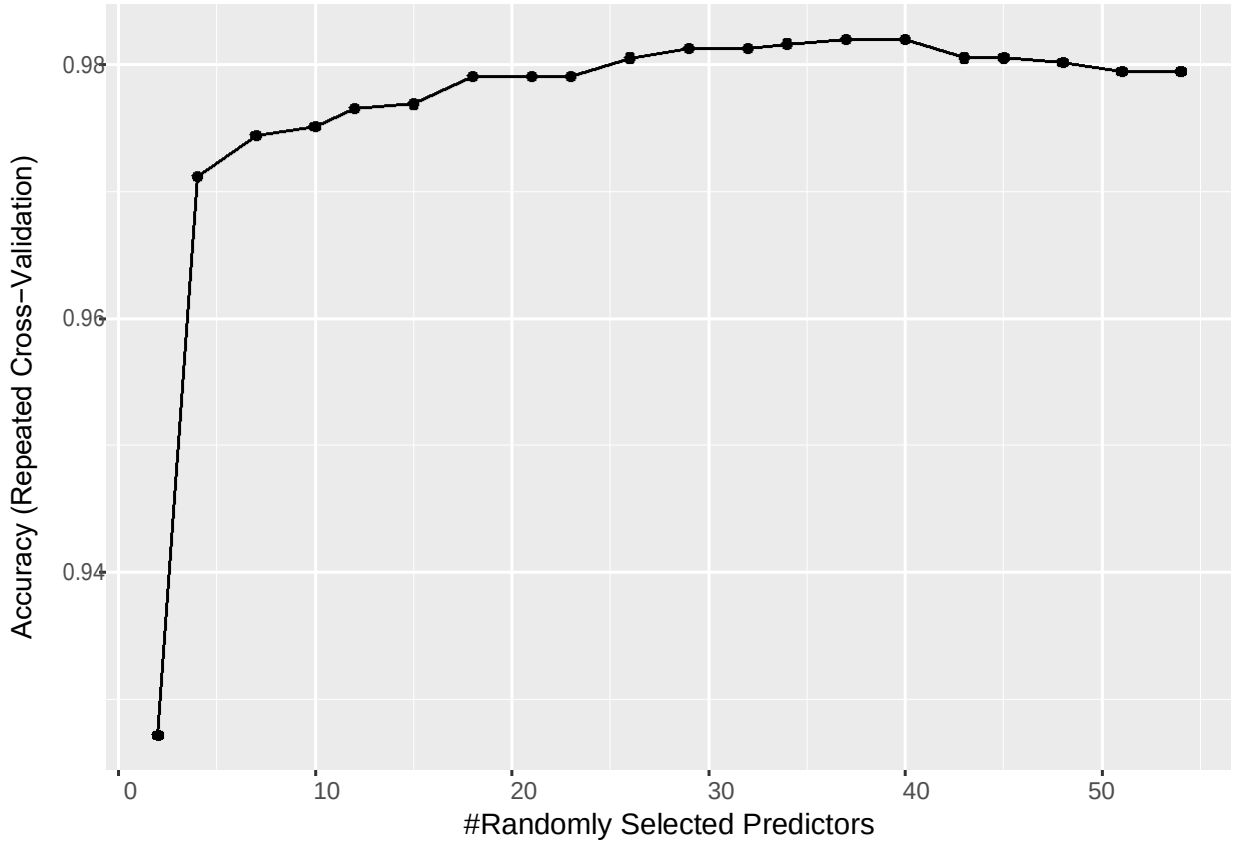
```
set.seed(2, sample.kind = "Rounding")
rf_model <- train(status~., method="rf", data=normtrainset, trControl=control, tuneLength=20)
ggplot(rf_model)
```




```
rf_acc <- max(rf_model$results$Accuracy)

set.seed(2, sample.kind = "Rounding")
rf_model_1k <- train(status~., method="rf", data=normtrainset,ntree=1000, trControl=control, tuneLength=
ggplot(rf_model_1k)
```




```
rf_1k_acc <- max(rf_model_1k$results$Accuracy)

set.seed(2, sample.kind = "Rounding")
rf_model_2k <- train(status~., method="rf", data=normtrainset,ntree=2000, trControl=control, tuneLength=
ggplot(rf_model_2k)
```



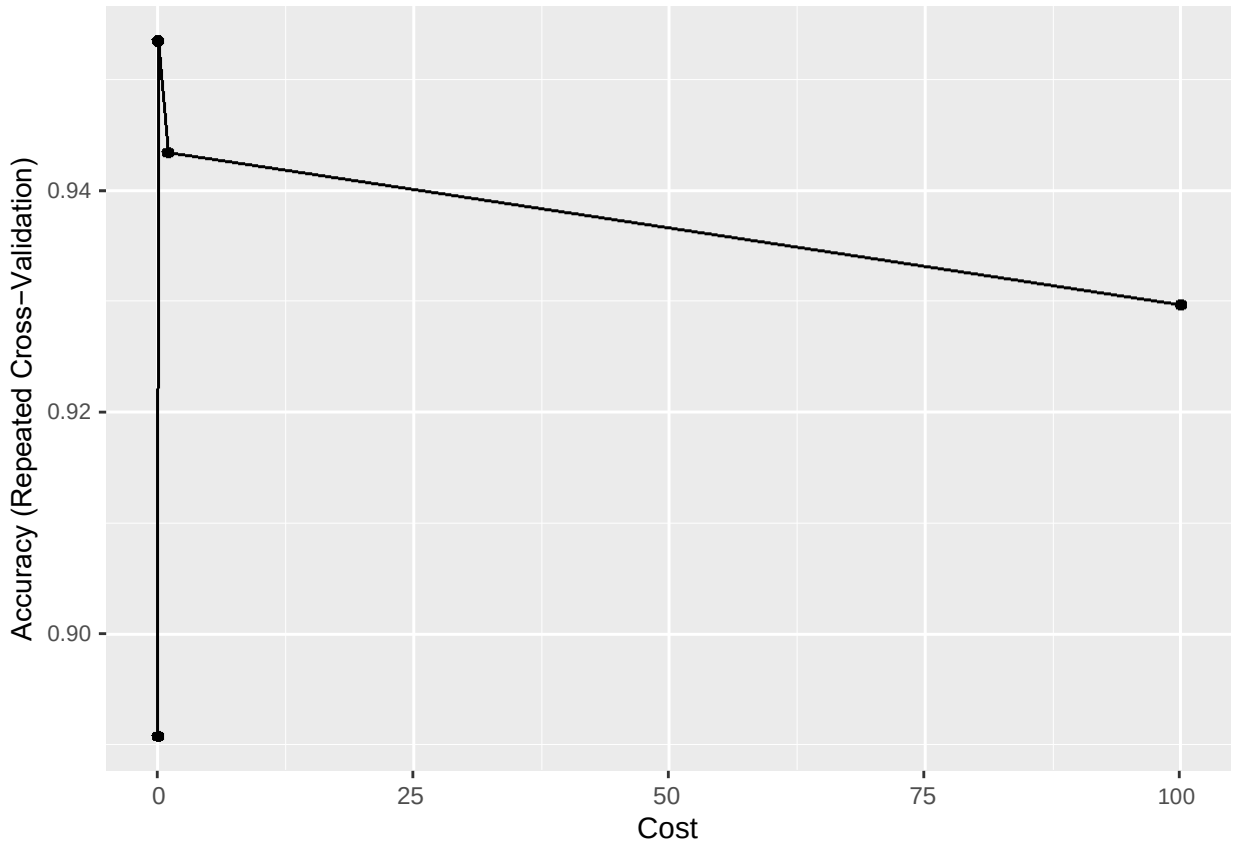

```
rf_2k_acc <- max(rf_model_2k$results$Accuracy)
```

Of our tuned random forest models, the best cross validated accuracy came from the 1000 ntree model with mtry of 32. The accuracy of 0.9819924 was just 1.6851573×10^{-7} *greater than the 2000 ntree model.*

SVM *We will try the SVM algorithm with two different Kernels, and tune for cost. SVM works essentially by creating a decision boundary in a higher dimensionality than the data. Different kernels enable this decision boundary to be non linear as standard. We will try a linear and a radial kernel. The cost training parameter is a miss-classification penalty which can be applied to better decide a decision boundary.*

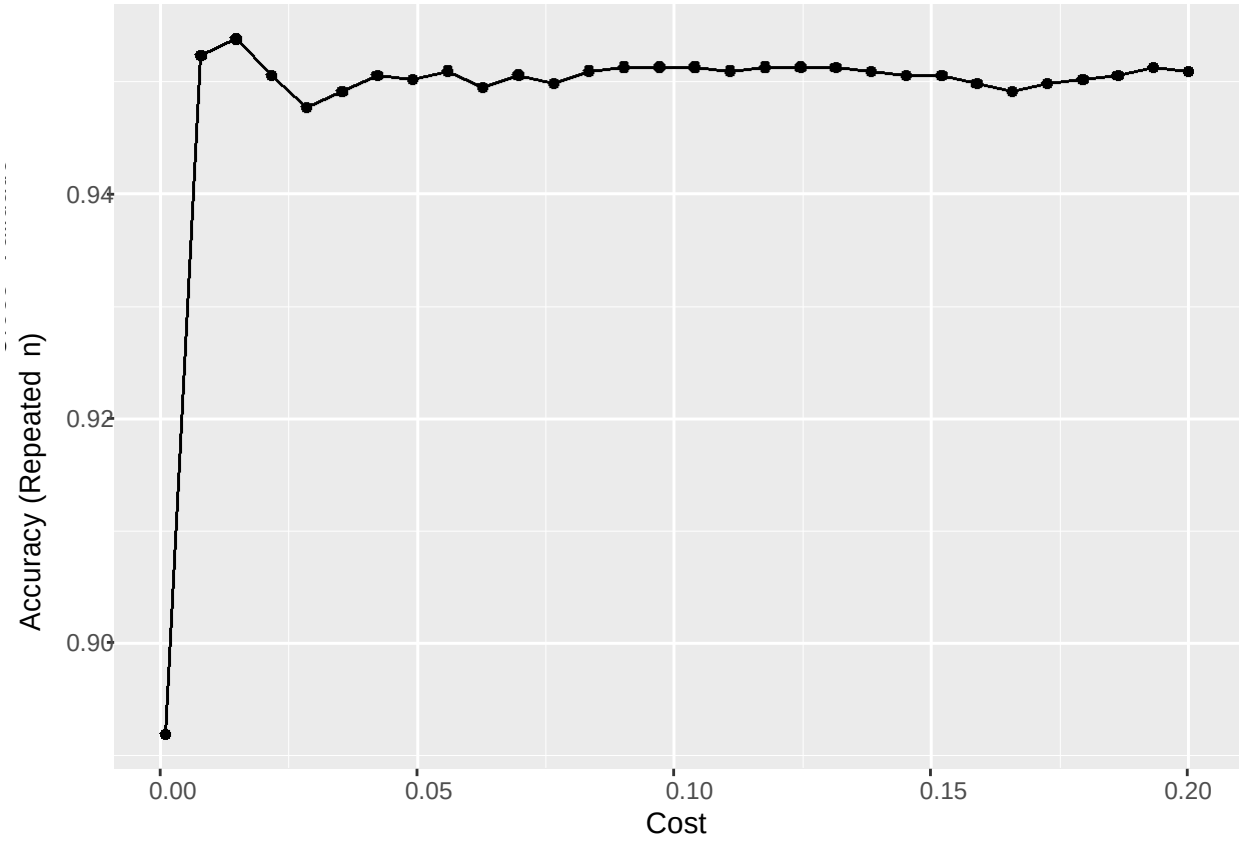
To tune the cost we will being by tuning from 0.0001 to 100, an initial value drawn from reading stacked overflow content regarding SVMs. Following this we will tune more precisely.

```
set.seed(2, sample.kind = "Rounding")
svm_l_model <- train(status~.,
                     method="svmLinear",
                     data=normtrainset,
                     trControl=control,
                     tuneGrid=data.frame(C=(0.0001 * 10^(seq(0,6,2)))))
ggplot(svm_l_model)
```



From this we can see that a cost value of 0.01 giving and accuracy of 0.9534692 was our best initial model, we will now refine that and tune for 30 values between 0.001 and 0.2 to find our best linear model.

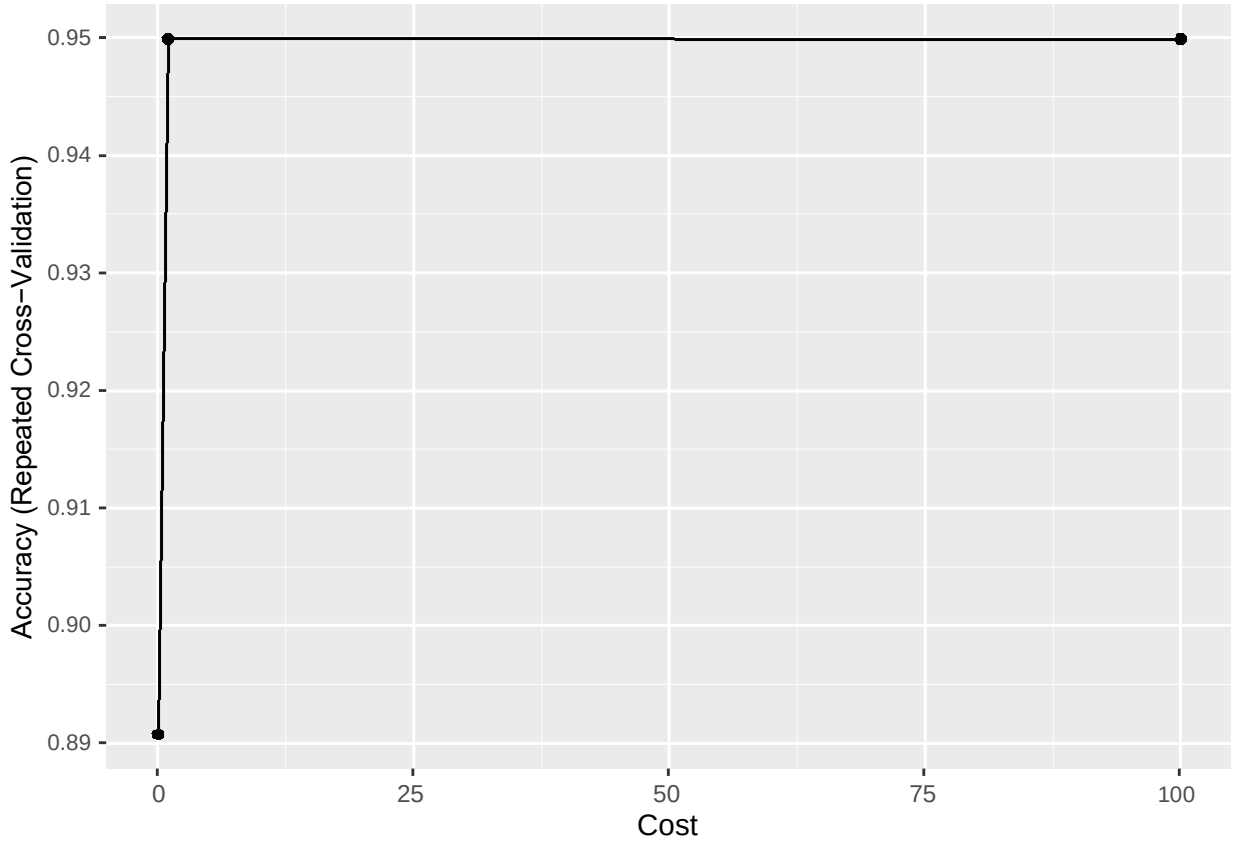
```
set.seed(2, sample.kind = "Rounding")
svm_l_model <- train(status~.,
                     method="svmLinear",
                     data=normtrainset,
                     trControl=control, tuneGrid=data.frame(C=seq(0.001,0.2,length=30)))
ggplot(svm_l_model)
```



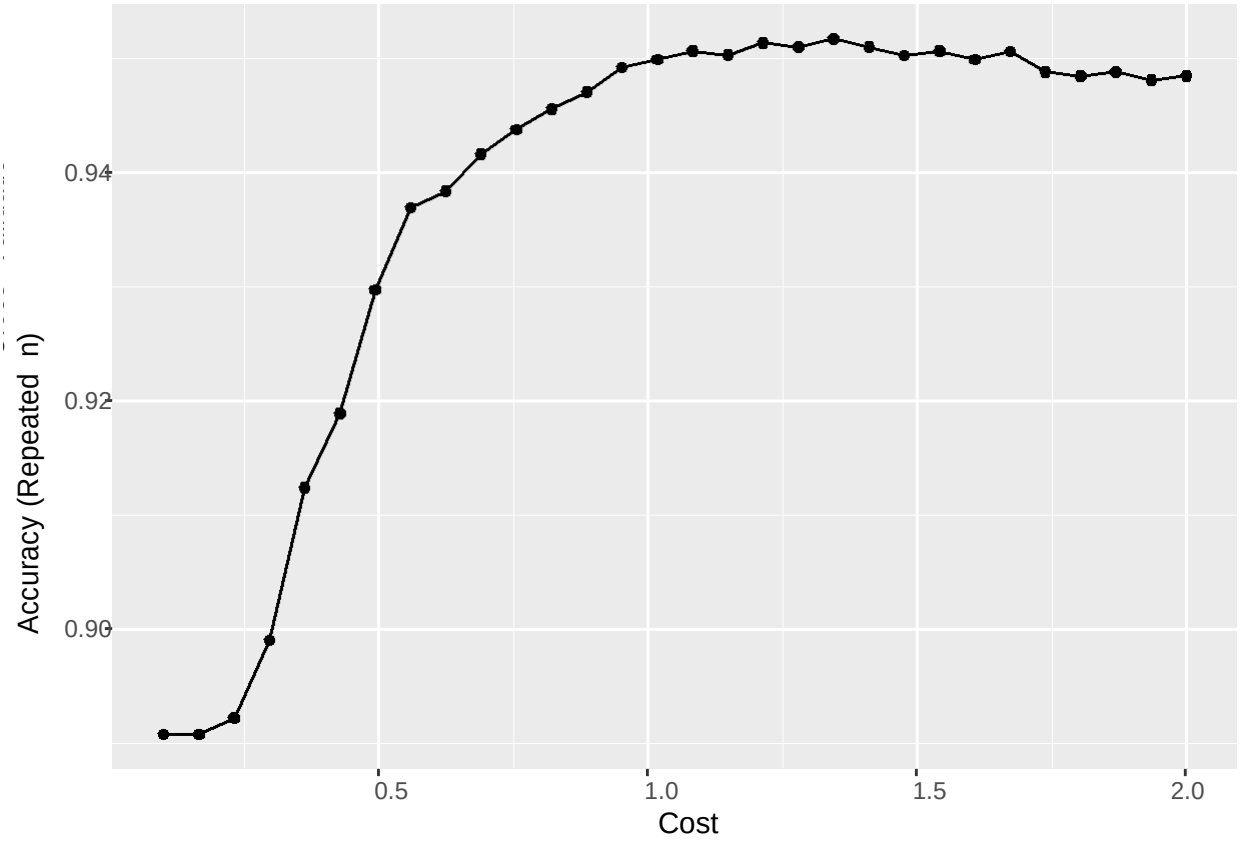

```
svm_l_acc <- max(svm_l_model$results$Accuracy)
```

Our final tuning improves on the model and uses a cost value of 0.0147241379310345 to present an accuracy of 0.9538431. This process will be repeated using the radial kernal.

```
set.seed(2, sample.kind = "Rounding")
svm_r_model <- train(status~.,
                     method="svmRadialCost",
                     data=normtrainset,
                     trControl=control,
                     tuneGrid=data.frame(C=(0.0001 * 10^(seq(0,6,2)))))
ggplot(svm_r_model)
```




```
set.seed(2, sample.kind = "Rounding")
svm_r_model <- train(status~.,
                     method="svmRadialCost",
                     data=normtrainset,
                     trControl=control, tuneGrid=data.frame(C=seq(0.1,2,length=30)))

ggplot(svm_r_model)
```




```
svm_r_acc <- max(svm_r_model$results$Accuracy)
```

Here we see our best result from the radial kernel gives an accuracy of 0.951677 using a cost of 1.3448275862069.

Model Comparison

model	accuracy
KNN	0.9275327
Random Forest	0.9816261
Random Forest 1k	0.9819924
Random Forest 2k	0.9819922
SVM Linear	0.9538431
SVM Radial	0.9516770

From our models we can see that random forest models performed best, then the SVM, then KNN. This order follows the conclusions of Delgado(2014), Caruana(20XX), and Wainer et. al (2016) which were used in the selection of algorithms for this project.

Of the random forest models, the 1000 ntree model performed the best with marginal gains over the 2000 ntree model. Looking into this model more we can see that the top 5 most important variables are egg cycles, total points, height, weight and hp. This reflects some of the patterns seen in the exploratory graphs and the grouping and status separation available through these features.

```
##  rf variable importance
##
##     only 20 most important variables shown (out of 54)
##
##                          Overall
## egg_cycles             100.0000
## total_points            53.9838
## height_m                13.5636
## weight_kg                8.7535
## hp                       6.8846
## growth_rate.Slow         6.1594
## sp_attack                3.9917
## attack                   3.1402
## type_1.Normal            2.6652
## speed                    2.5958
## sp_defense               2.5450
## defense                  1.7182
## generation               1.6114
## type_2.Fairy             1.3287
## type_1.Water             1.3030
## type_1.Dragon            1.2069
## type_2.Psychic           0.7399
## type_1.Ground            0.6255
## type_1.Psychic           0.4869
## type_2.Flying            0.4338
```

As it was our best performing model when assessed using cross validation, it is the model we will recommend to be tested.

Results

To assess the effectiveness of our model we will use it to predict the status of Pokemon in the test set. A confusion matrix will then be to show the overall accuracy and investigate specifics of the model performance.

```
set.seed(3, sample.kind = "Rounding")
```

```
##  Warning in set.seed(3, sample.kind = "Rounding"): non-uniform 'Rounding' sampler
##  used
```

```
final_predictions <- predict(rf_model_1k, normtestset)
confusionMatrix(data = final_predictions, reference = testset$status)
```

```
##  Confusion Matrix and Statistics
##
##                     Reference
## Prediction            Legendary Mythical Normal Sub Legendary
##     Legendary                 4        1      0             0
##     Mythical                  0        1      0             0
##     Normal                    0        0     92             0
##     Sub Legendary             0        1      0             5
##
## Overall Statistics
```

```
##
##                          Accuracy : 0.9808
##                            95% CI : (0.9323, 0.9977)
##       No Information Rate : 0.8846
##       P-Value [Acc > NIR] : 0.0003065
##
##                             Kappa : 0.9095
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                                Class: Legendary Class: Mythical Class: Normal
## Sensitivity                             1.00000         0.333333        1.0000
## Specificity                             0.99000         1.000000        1.0000
## Pos Pred Value                          0.80000         1.000000        1.0000
## Neg Pred Value                          1.00000         0.980583        1.0000
## Prevalence                              0.03846         0.028846        0.8846
## Detection Rate                          0.03846         0.009615        0.8846
## Detection Prevalence                    0.04808         0.009615        0.8846
## Balanced Accuracy                       0.99500         0.666667        1.0000
##                                Class: Sub Legendary
## Sensitivity                                 1.00000
## Specificity                                 0.98990
## Pos Pred Value                              0.83333
## Neg Pred Value                              1.00000
## Prevalence                                  0.04808
## Detection Rate                              0.04808
## Detection Prevalence                        0.05769
## Balanced Accuracy                           0.99495
```

Our model performs well with an overall accuracy of 0.9807692. Of the predictions made the incorrect ones were all Mythical status Pokemon. As we saw through the data exploration, this was the status group with the fewest entries in the training dataset. The Mythical and Sub legendary statuses were also the ones which had less distinct groupings throughout the exploration.

Conclusion

This project aimed to develop a model to classify a Pokemon into the correct status group based upon a set of features. Based upon the multiple classification requirement and informed by literature, KNN, Random forest and SVM models were tuned to accomplish this. Our Random forest model utilisting 1000 trees and considering 32 random features at each junction performed best in cross validation tuning. When applied to the test set for a final result it achieved an overall accuracy of 0.9807692.

Further research could build on this by applying the model to the next generation of Pokemon to gauge performance, and then investigate improvements. Alternatively a wider range of algorithms could be applied to the existing training set, to further investigate the impact of different families of algorithm. Following the reasoning behind the selection for the algorithms tried here, we would recommend appraisal of a wider range of SVM kernels, neural networks and C50 ensembles thereby continuing the general order of the findings from Delgado(2014).

Bibliography

Caruana, R. (2006). An Empirical Comparison of Supervised Learning Algorithms Rich. Icml, 161–168. https://doi.org/10.1145/1143844.1143865

Fernández-Delgado, M., Cernadas, E., Barro, S., & Amorim, D. (2014). Do we need hundreds of classifiers to solve real world classification problems? Journal of Machine Learning Research, 15, 3133–3181. https: //doi.org/10.1117/1.JRS.11.015020

Wainer, J. (2016). Comparison of 14 different families of classification algorithms on 115 binary datasets. 2014. http://arxiv.org/abs/1606.00930