Identifying fermionic decay signals of the Higgs boson with classification algorithms

Georgina Mansell and Anthony Stephenson

Contents

Introduction

In 2014, CERN provided a simulated dataset from their ATLAS experiment for use in a programming competition on Kaggle. After the Kaggle Challenge closed, the full dataset was made available here, with the accompanying documentation here. The goal of the challenge is to develop a binary classification model to distinguish signal and background events.

The dataset contains 818,238 samples with 30 features, a class label, and 4 columns of additional information such as sample weight and event ID. Of this, 250,000 samples were provided as training data for the Kaggle Challenge (set "t"), 100,000 for the public leaderboard (set "b"), 450,000 for the private leaderboard (set "v"), and the remaining 18,238 were unused (set "u").

The samples are simulated events from the Large Hadron Collider (LHC). In an event, bunches of protons are accelerated around the LHC in opposite directions and collide. The collision produces hundreds of particles, most of which are unstable and decay into lighter particles such as electrons or photons. Sensors in the LHC measure properties of the surviving particles, and from this the properties of the parent particles can be inferred. Signal events are defined as events where a Higgs boson decays into two tau particles.

In the following markdown notebooks we use methods from SM1 and SC1 to attempt the Kaggle Challenge in R. We chose to follow a similar structure, using the Kaggle training set to train our model, and the private leaderboard set as our hold-out validation set. The public leaderboard and unused datasets are excluded for simplicity.

The notebooks are structured as follows:

1. Data Exploration - an introduction to the dataset, its structure, and our R package `lhc`.
2. SVM - we attempt to use our implementation of SVM algorithms to train a classifier, though we are restricted to train on a small fraction of the data.
3. Logistic Regression - we use our implementation of logistic regression within an OOP framework to explore models with varying parameters and feature engineering.
4. Results - we compare our models and use the best performing model on our validation set.

The package we have developed for assessment is called `lhc` and is available on github here. The functions are divided thematically into the following files:

- utility_funcs.r
- objects.r
- plot_funcs.r
- lr_funcs.r
- kernel_funcs.r
- svm_funcs.r
- project_funcs.r

Data Exploration

Load packages

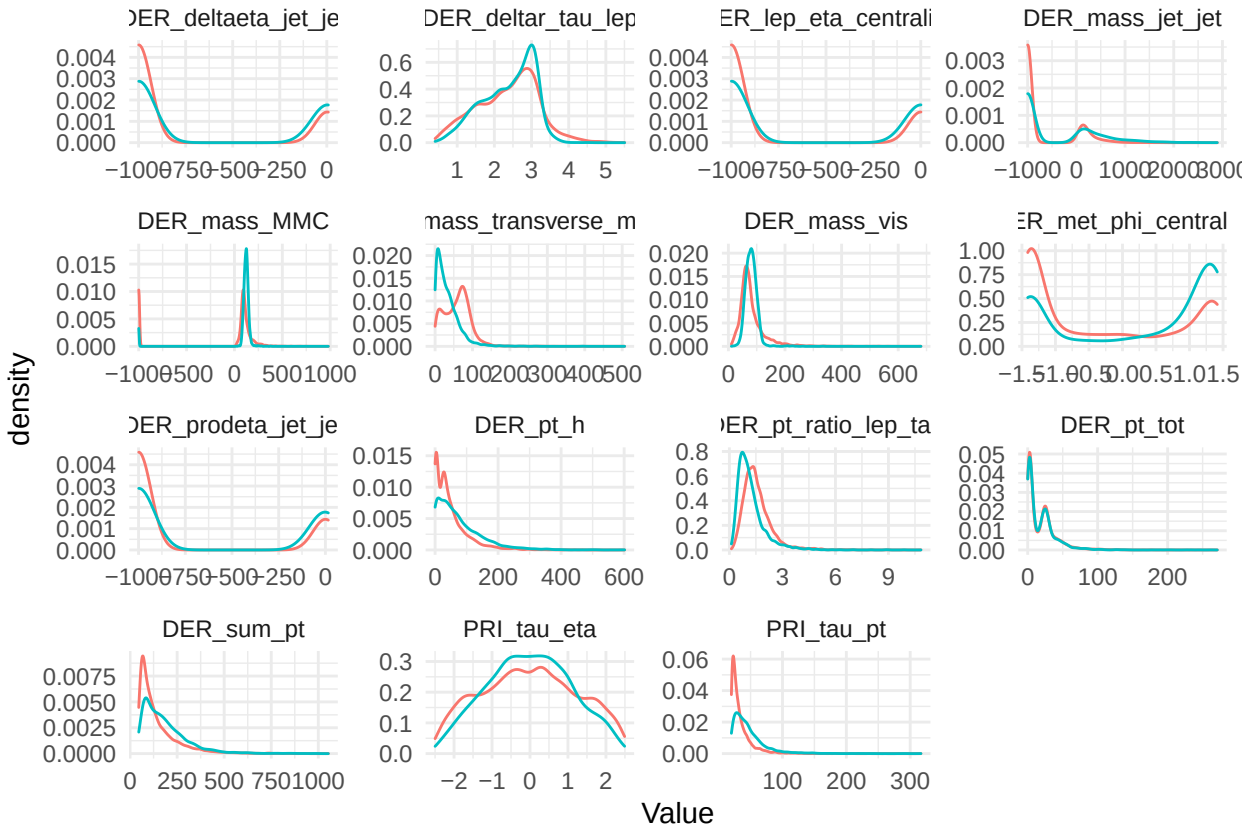
```
devtools::install_github("ant-stephenson/lhc")
```

```
##      checking for file '/tmp/Rtmpdzzl9M/remotes445ec44091c55/ant-stephenson-lhc-47a6e05/DESCRIPTION'
##   -  preparing 'lhc':
##    checking DESCRIPTION meta-information ...  v  checking DESCRIPTION meta-information
##   -  checking for LF line-endings in source and make files and shell scripts
##   -  checking for empty or unneeded directories
## -  building 'lhc_0.1.0.tar.gz'
##
##
```

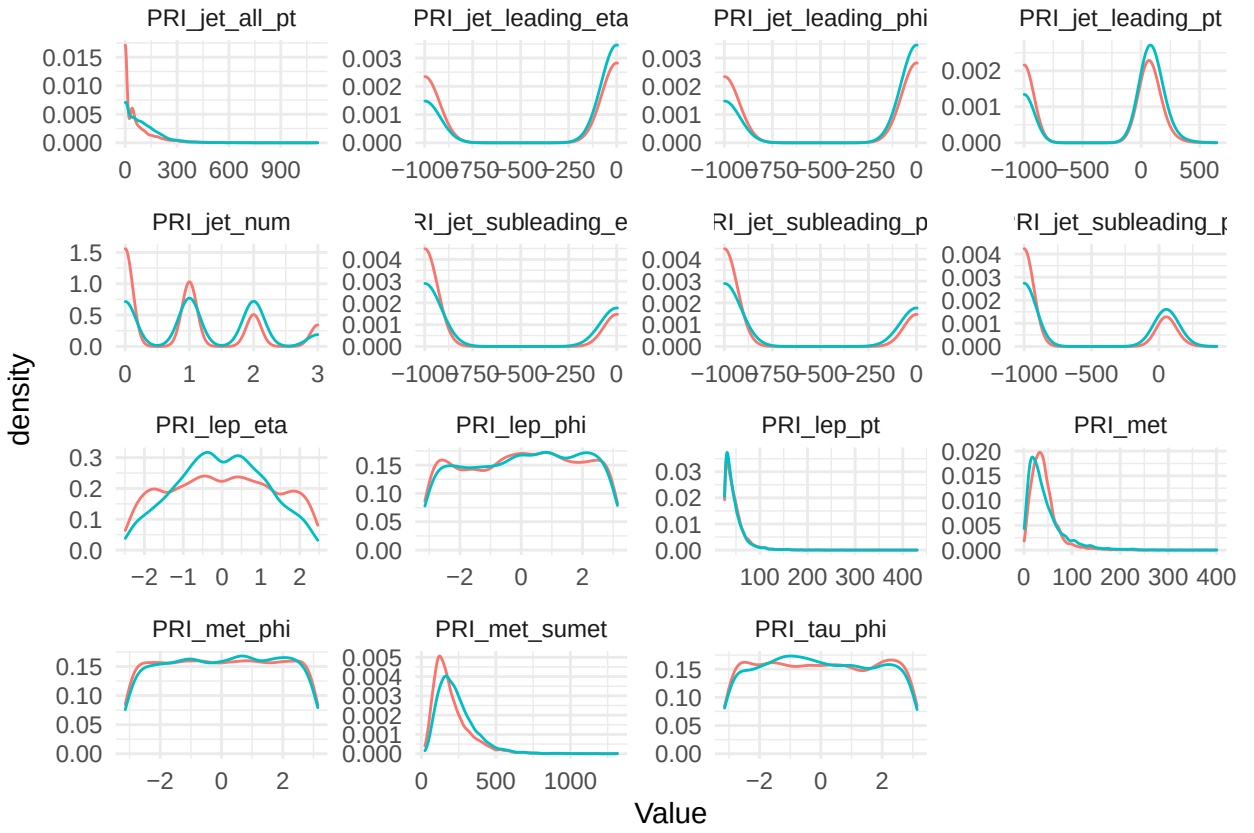
```
library(lhc)
library(dplyr)
library(ggplot2)
library(tidyr)
library(kableExtra)
```

Load dataset

```
#get the filepath of the dataset
filepath <- list.files(path="~", pattern="atlas-higgs-challenge-2014-v2.csv",
                       full.names=T, recursive=T)

#fast load the raw data
raw_data <- data.table::fread(filepath)

#Split data into X (just variables) and additional info
all_info <- as.data.frame(raw_data[, c("KaggleSet", "KaggleWeight", "Weight", "Label")])
all_X <- as.matrix(raw_data[, -c("EventId", "KaggleSet", "KaggleWeight", "Weight", "Label")])

#assign event id as row names so we can check the two stay matched
rownames(all_info) <- rownames(all_X) <- raw_data$EventId

#add a column with numerical coding of class label (0,1)
all_info$Y <- as.numeric(all_info$Label == "s")

#Select the training set
X <- all_X[raw_data$KaggleSet=="t",]
info <- all_info[raw_data$KaggleSet=="t",]

#View the variables
plot_distributions(X, variables = colnames(X)[1:15], labels = info$Label)
```



```
plot_distributions(X, variables = colnames(X)[16:30], labels = info$Label)
```



By looking at the distributions of the data we can see that the two classes appear fairly similar and so we might expect a reasonably complex model will be needed to separate them. All of the variables are continuous except for `PRI_jet_num` which is discrete (0, 1, 2, 3). For some of the variables there are peaks at -999. This is because -999 has been used to indicate undefined values which cannot be computed for a physical reason.

Performance Metrics

The simulated dataset has been generated so that the number of signal and background events are roughly equal. However in reality, there are far more background events than signal, and so the samples have also been given importance weightings. The weightings within each class sum to the actual expected number of signal and background events, $\sum_{i;y_i=s} w_i = N_s$, and $\sum_{i;y_i=b} w_i = N_b$. Whenever we take a training and test set from our total training data, we need to ensure the weightings are rescaled.

To assess the accuracy of a binary classification model, a standard approach is to plot a ROC (Receiver Operating Characteristic) curve and calculate its AUC (Area Under the Curve). To create the plot, you take the output of a probabilistic model and plot the true positive rate against the false positive rate as the decision threshold is varied. Using this approach here will tell us about the model's accuracy in terms of number of samples in the dataset, but it will not take into account the sample weights. Note that this approach cannot be used for non-probabilistic models such as SVM, as there is not a continuous output that different thresholds can be applied to.

The objective of the Kaggle Challenge is instead to maximise the AMS (Approximate Median discovery Significance) metric, which is defined as:

$$\text{AMS} = \sqrt{2\left(\left(s+b+b_{reg}\right)\ln\left(1+\frac{s}{b+b_{reg}}\right)-s\right)}$$

where s is the sum of the sample weights of true positives $s = \sum_{i;y_i=s,\hat{y}_i=s} w_i$, and b is the sum of sample weights of false positives $b = \sum_{i;y_i=b,\hat{y}_i=s} w_i$, and $b_{reg} = 10$. For probabilistic models we can look at how the AMS changes as different decision thresholds are applied, and we can use it to select the most appropriate threshold.

In our package `lhc`, we have defined two reference class objects to store data related to the ROC and AMS performance measures `ROC_curve` and `AMS_data`. We have also defined plotting functions to plot multiple ROC curves or AMS metrics on the same axes in order to visualise the variance of models during cross validation.

```
#when we take a subset of samples, weights need to be renormalised
#so the sum of signal weights = Ns and sum of background weights = Nb
Ns <- sum(info$KaggleWeight[info$Label=="s"])
  #or sum(all_info$Weight[all_info$Label=="s"]) as the two are equal
Nb <- sum(info$KaggleWeight[info$Label=="b"])
  #or sum(all_info$Weight[all_info$Label=="b"])

#these values are set as the default scaling factors in our function ams_metric
#which is called within the AMS_data class
print(c(Ns, Nb))
```

```
## [1]    691.9886 410999.8473
```

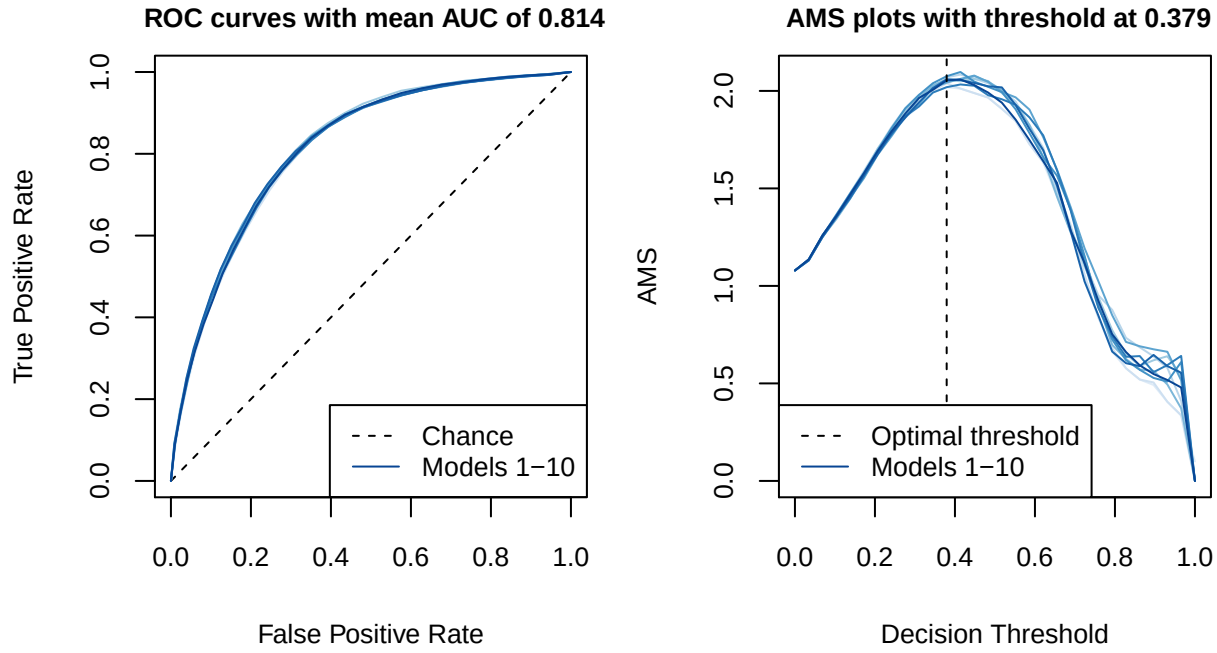
Baseline Model

In our package, we have implemented logistic regression with iteratively weighted least squares (IWLS) in the function `logistic_reg`, and we have created an associated object class `logistic_model`.

As a first pass, we perform a simple logistic regression on the training set with k-fold cross validation, and view the ROC curves and AMS data for each fold.

```
# get an index for CV groups
k <- 10
kI <- partition_data(n=nrow(X), k=k, random=T)

#create lists to hold the k models, roc and ams data
models <- vector("list", k)
rocs <- vector("list", k)
amss <- vector("list", k)

#for each fold, subset the training and test data
for(i in 1:k){
  X_train <- X[kI != i,]
  y_train <- info[kI != i, "Y"]

  X_test <- X[kI == i,]
  y_test <- info[kI == i, "Y"]
  w_test <- info[kI == i, "Weight"]

  #fit a logistic regression model to the CV training data
  models[[i]] <- logistic_model$new(X=X_train, y=y_train)

  #use it to predict the classifications of the test data
  prob <- models[[i]]$predict(X_test)

  #store roc and ams data
```

```
  rocs[[i]] <- ROC_curve$new(y_test, prob)
  amss[[i]] <- AMS_data$new(y_test, prob, w_test)
}

par(mfrow=c(1,2))
par(mar=c(4,4,2,1))
plot_rocs(rocs, scale=0.8)
plot_amss(amss, scale=0.8)
```



We can see the logistic regression model has low variance between folds and a reasonable average AUC.

On the left of the AMS graph (t=0) all samples are classified at signal events, and so the sum of true positive weights, s, is at a maximum and the sum of false positive weights, b, is at a minimum. On the right of the graph (t=1) all samples are classified as background events, and so `s=b=0` and AMS = 0. The best decision threshold for this initial model is around t=0.4.

Missing data

Currently the undefined values are still coded as -999. Since the missing values have a physical meaning, we may expect there to be some structure to the missing values.

```
#creating a copy of X just coding if the value is missing or non missing
missing <- matrix(as.numeric(X == -999), ncol=ncol(X))
colnames(missing) <- colnames(X)

#considering just the columns that contain missing data
missing <- missing[,colSums(missing) != 0]

#using dplyr to group the different types of row
missing_pattern <- as_tibble(missing) %>%
  group_by_all() %>%
  count() %>%
  ungroup()
```

```r
#display the table over two rows with kable
display_table <- function(data){
  n <- ncol(data)/6
  for(i in 1:n){
    print(kable(data[(6*i-5):(6*i)], booktabs=T)  %>%
      kable_styling(latex_options="scale_down"))
  }
}
display_table(missing_pattern)
```

DER_mass_MMC	DER_deltaeta_jet_jet	DER_mass_jet_jet	DER_prodeta_jet_jet	DER_lep_eta_centrality	PRI_jet_leading_pt
0	0	0	0	0	0
0	1	1	1	1	0
0	1	1	1	1	1
1	0	0	0	0	0
1	1	1	1	1	0
1	1	1	1	1	1

PRI_jet_leading_eta	PRI_jet_leading_phi	PRI_jet_subleading_pt	PRI_jet_subleading_eta	PRI_jet_subleading_phi	n
0	0	0	0	0	68114
0	0	1	1	1	69982
1	1	1	1	1	73790
0	0	0	0	0	4429
0	0	1	1	1	7562
1	1	1	1	1	26123

By looking for patterns of missing data, we identify 6 different types of sample. The first types and the last 3 types are identical other than the variable `DER_mass_MMC` being missing or not missing. By looking at the variables names, the main structure appears to be related to jets (a narrow cone of hadrons).

```r
#looking at how the missing patterns relate to the categorical variable
missing <- cbind(X[,"PRI_jet_num"], missing)
colnames(missing)[1] <- "PRI_jet_num"

#ignoring DER_mass_MMC and grouping again
missing_pattern <- as_tibble(missing) %>%
  select(-DER_mass_MMC) %>%
  group_by_all() %>%
  count() %>%
  ungroup()

display_table(missing_pattern)
```

PRI_jet_num	DER_deltaeta_jet_jet	DER_mass_jet_jet	DER_prodeta_jet_jet	DER_lep_eta_centrality	PRI_jet_leading_pt
0	1	1	1	1	1
1	1	1	1	1	0
2	0	0	0	0	0
3	0	0	0	0	0

PRI_jet_leading_eta	PRI_jet_leading_phi	PRI_jet_subleading_pt	PRI_jet_subleading_eta	PRI_jet_subleading_phi	n
1	1	1	1	1	99913
0	0	1	1	1	77544
0	0	0	0	0	50379
0	0	0	0	0	22164

So we have found the following pattern:

- when jet_num = 0 the following variables are undefined

```
missing_pattern[,1] <- NULL
colnames(missing_pattern)[missing_pattern[1,] == 1]
```

```
##  [1] "DER_deltaeta_jet_jet"   "DER_mass_jet_jet"       "DER_prodeta_jet_jet"
##  [4] "DER_lep_eta_centrality" "PRI_jet_leading_pt"     "PRI_jet_leading_eta"
##  [7] "PRI_jet_leading_phi"    "PRI_jet_subleading_pt"  "PRI_jet_subleading_eta"
## [10] "PRI_jet_subleading_phi"
```

- when jet_num = 1 the following variables are undefined

```
colnames(missing_pattern)[missing_pattern[2,] == 1]
```

```
## [1] "DER_deltaeta_jet_jet"   "DER_mass_jet_jet"       "DER_prodeta_jet_jet"
## [4] "DER_lep_eta_centrality" "PRI_jet_subleading_pt"  "PRI_jet_subleading_eta"
## [7] "PRI_jet_subleading_phi"
```

- and when jet_num = 2 or 3, there are no undefined variables.

This pattern makes physical sense, because if there are 0 jets, all the jet variables are missing, and if there is 1 jet all the leading jet variables are there and the subleading jet variables are missing.

Groups

It appears that splitting out the data based on patterns of missing data may be appropriate. Each of the subsets will contain similar types of event and we can remove the missing variables within each group. If we split the data into the 6 different missing data patterns, some of the groups where `DER_mass_MMC` is missing are very small (only 4,000 out of 250,000 samples). Therefore to keep our training groups sufficiently large, we will split the data into 3 groups based on `PRI_jet_num`, and aim to select 3 different models.

```
#adding a new column in info which indicates the groupings
info$Group <- factor(X[,"PRI_jet_num"],
                     levels=c(0, 1, 2, 3),
                     labels=c("j=0", "j=1", "j=2+", "j=2+"))

G <- nlevels(info$Group)
groups <- levels(info$Group)

#define which columns we can remove from each subset (now constants, sd=0)
features_to_rm <- vector("list", 3)
for(g in 1:G){
  features_to_rm[[g]] <- colnames(X)[apply(X[info$Group==groups[g],], 2, sd) == 0]
}

#check how the number of signal/background events are distributed in each group
n_stats <- as.data.frame(unclass(table(info$Group, info$Label)))
n_ratio <- n_stats/rowSums(n_stats)

n_stats <- cbind(n_stats, rowSums(n_stats))
n_stats <- rbind(n_stats, colSums(n_stats))
colnames(n_stats)[3] <- rownames(n_stats)[4] <- "total"

kable(n_ratio, booktabs=T)
```

	b	s
j=0	0.7448580	0.2551420
j=1	0.6426545	0.3573455
j=2+	0.5524723	0.4475277

```
kable(n_stats, booktabs=T)
```

	b	s	total
j=0	74421	25492	99913
j=1	49834	27710	77544
j=2+	40078	32465	72543
total	164333	85667	250000

```
#check how the weights are distributed
w_stats <- info %>%
  group_by(Group) %>%
  summarise(b = sum(Weight[Label=="b"]),
            s = sum(Weight[Label=="s"]),
            .groups ="drop") %>%
  as.data.frame()

rownames(w_stats) <- w_stats[,"Group"]
w_stats <- w_stats[,2:3]
w_ratio <- w_stats / rowSums(w_stats)

w_stats <- cbind(w_stats, rowSums(w_stats))
w_stats <- rbind(w_stats, colSums(w_stats))
colnames(w_stats)[3] <- rownames(w_stats)[4] <- "total"

kable(w_ratio, booktabs=T)
```

	b	s
j=0	0.9986757	0.0013243
j=1	0.9978769	0.0021231
j=2+	0.9965697	0.0034303

```
kable(w_stats, booktabs=T)
```

	b	s	total
j=0	85253.41	113.04725	85366.45
j=1	29315.73	62.37262	29378.10
j=2+	10748.76	36.99816	10785.76
total	125317.90	212.41803	125530.32

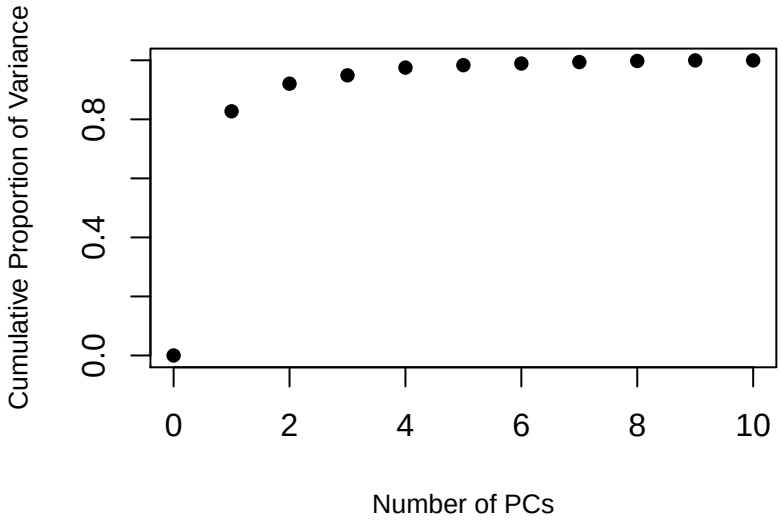
The three groups have varying ratios of signal and background events (approx 1:3, 1:2, and 1:1), though they have similar ratio of sample weights (approx 1:500) and have similar number of samples (70,000 to 100,000).

PCA

A standard approach for visualising a high dimensional dataset is to perform Principle Component Analysis (PCA) and to plot the first few principle components. We can select the number of PCs to visualise by looking at the proportion of variance that they explain.

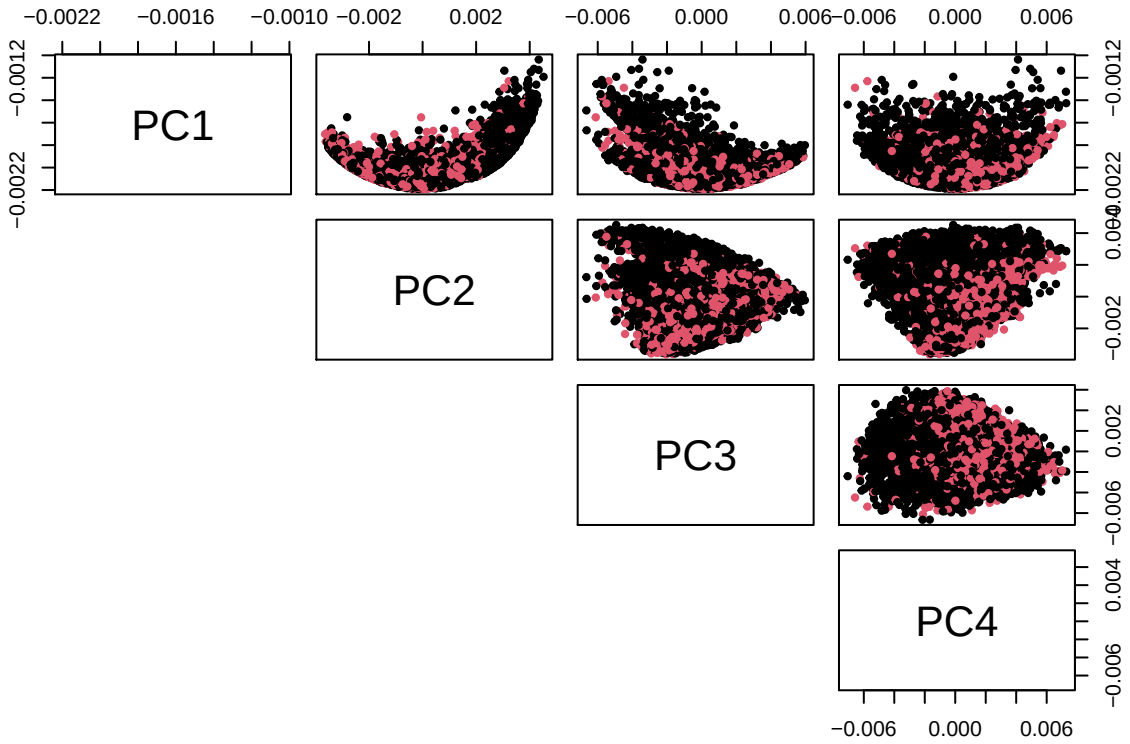
```
#perform pca excluding -999
X_temp <- X
```

```
X_temp[X==-999] <- NA
pca <- prcomp(na.omit(t(X_temp)), scale.=T, center=T)

#plot variance explained over n pcs
var_explained <- c(0, summary(pca)$importance[3,])
plot(0:10, var_explained[1:11], xlab="Number of PCs",
     ylab="Cumulative Proportion of Variance", pch=16, cex.lab=0.8)
```



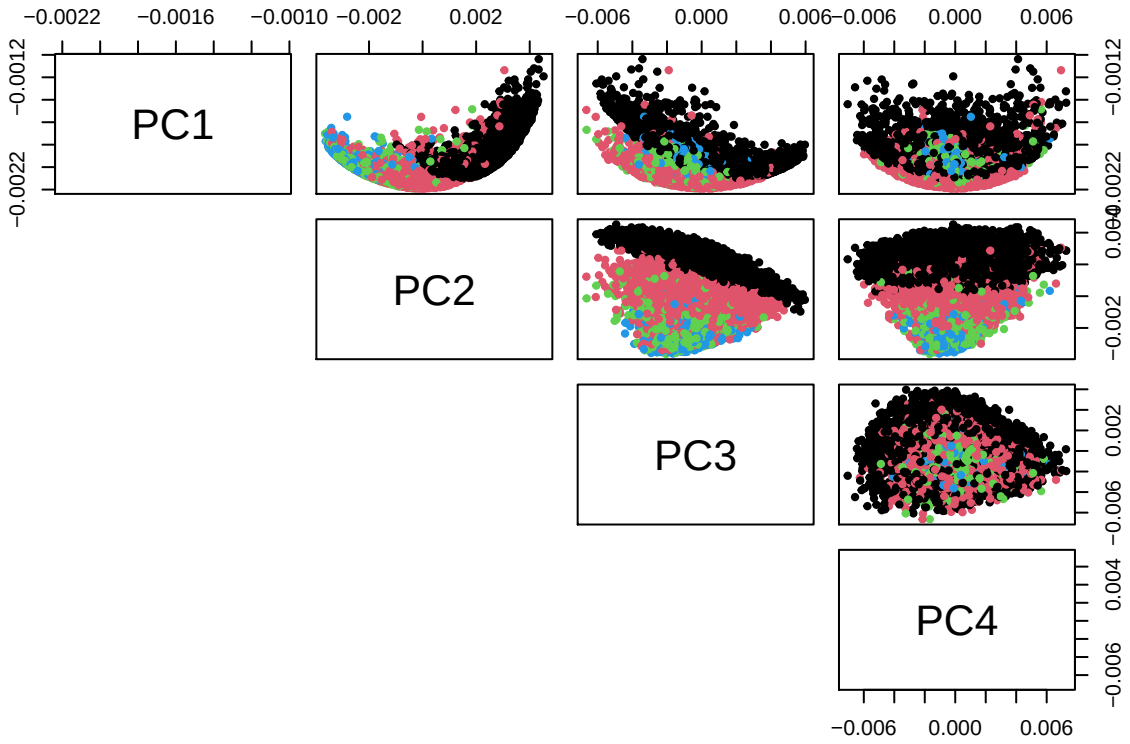
From the plot above we can see that the first 4 principle components explain nearly all the variance of the data. We can plot the transformed data coloured by class label to see if the PCA has separated the classes.

```
#the first 5 pcs explain nearly all the variance of the data
X_transformed <- pca$rotation

#plot the pcs of a random subset of samples
idx <- sample(1:nrow(X), 10000)
pairs(X_transformed[idx, 1:4], lower.panel = NULL, pch=20, col=info[idx, "Y"]+1)
```

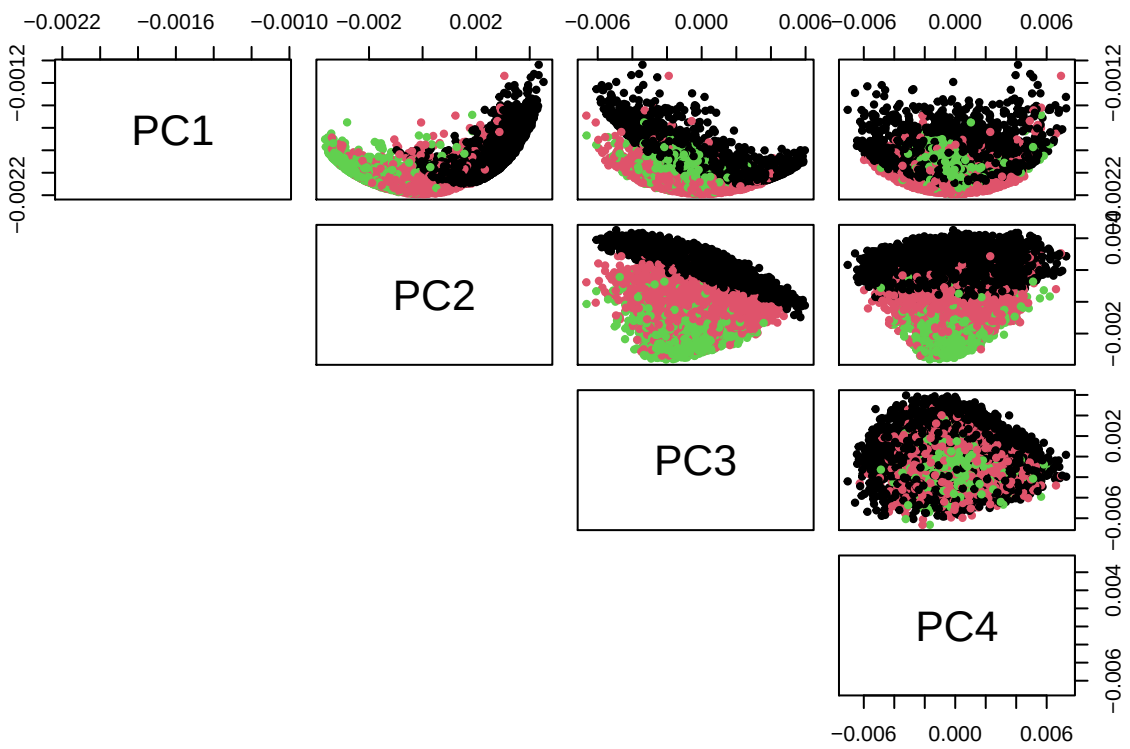


It looks like there may be some separation of classes in PC4. Let's instead colour the points by our new jet number groups.

```
#colour by the jet num
pairs(X_transformed[idx,1:4], lower.panel = NULL, pch=20, col=X[idx, "PRI_jet_num"]+1)
```



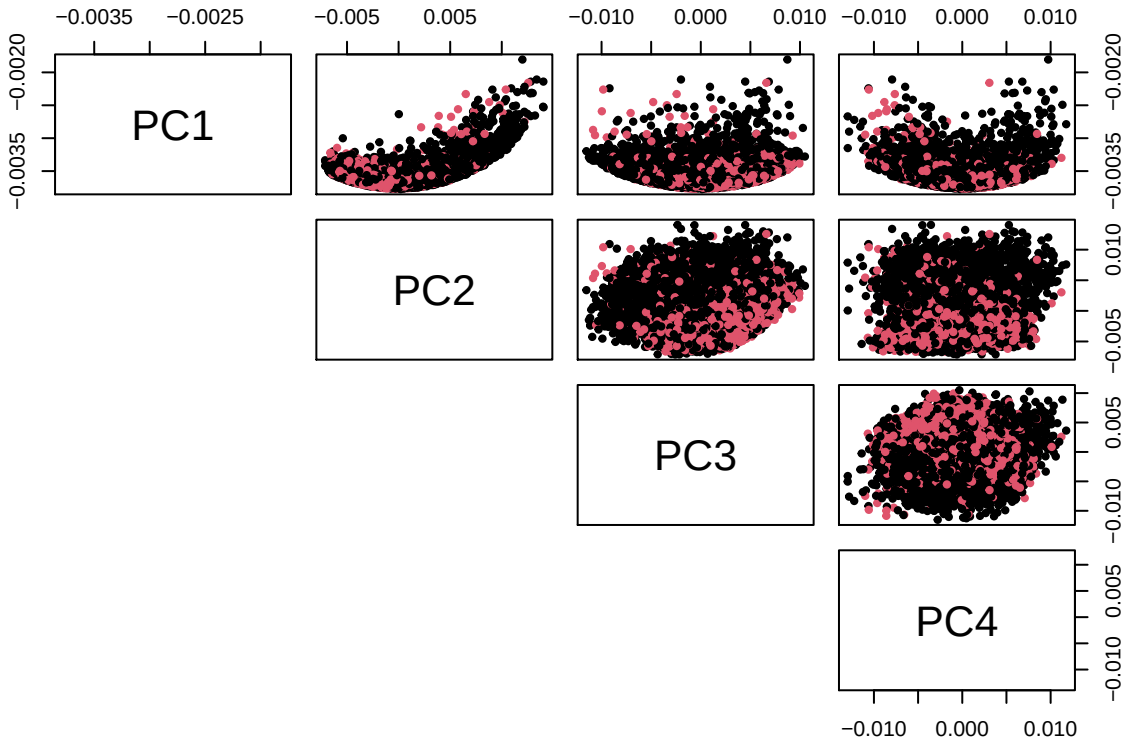

```
#colour by our groups
pairs(X_transformed[idx,1:4], lower.panel = NULL, pch=20, col=as.numeric(info[idx, "Group"]))
```



PC2 appears to separate the samples based on their missing data pattern. Events with 2 or 3 jets look very similar and so having these grouped together as jet=2+ seems appropriate.

Now lets look at a PCA on just one of the groups, which should now be much less effected by missing data. Unfortunately, the samples are still not easily separable.

```
X_temp <- X_temp[info$Group=="j=1", !colnames(X) %in% features_to_rm[[1]]]
info_temp <- info[info$Group=="j=1",]

pca <- prcomp(na.omit(t(X_temp)), scale.=T, center=T)
X_transformed <- pca$rotation

idx <- sample(1:nrow(X_temp), 10000)
pairs(X_transformed[idx,1:4], lower.panel = NULL, pch=20, col=info_temp[idx, "Y"]+1)
```

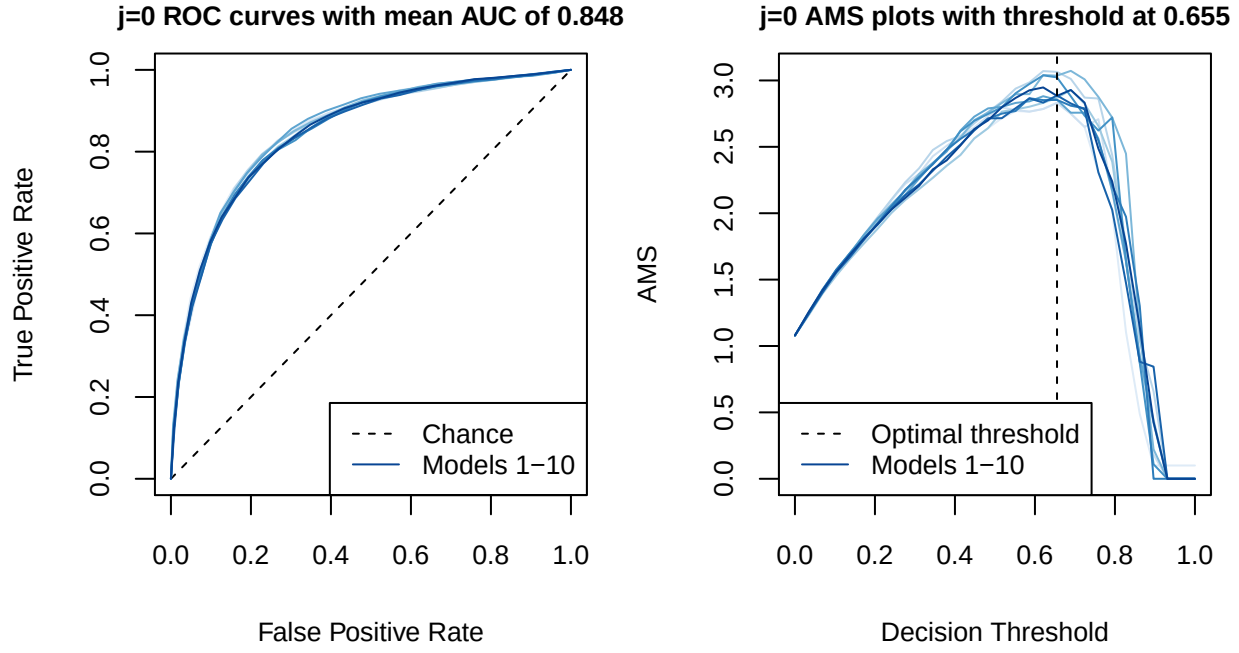


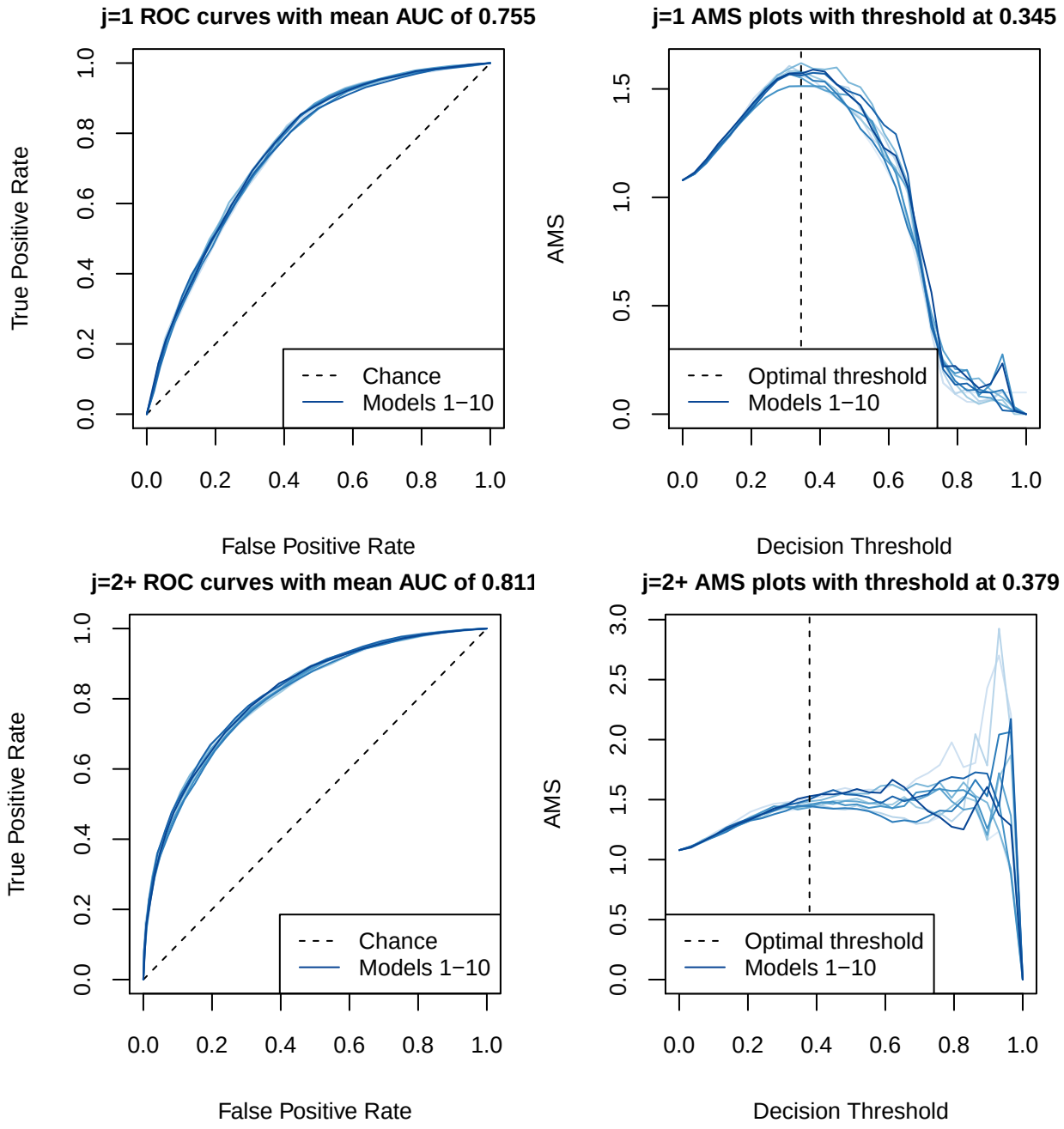
Baseline model per group

Now we have split the data into 3 groups, we'll train a logistic regression model for each group. Again we will use k-fold CV and our reference class objects.

```
par(mfrow=c(1,2))
par(mar=c(4,4,2,1))

for(g in 1:G){
  X_group <- X[info$Group==groups[g], !colnames(X) %in% features_to_rm[[g]]]
  info_group <- info[info$Group==groups[g],]

  # get an index for CV groups
  k <- 10
  kI <- partition_data(n=nrow(X_group), k=k, random=T)

  #create lists to hold the k models and k roc curves
  models <- vector("list", k)
  rocs <- vector("list", k)
  amss <- vector("list", k)

  for(i in 1:k){
    X_train <- X_group[kI != i,]
    y_train <- info_group[kI != i, "Y"]

    X_test <- X_group[kI == i,]
    y_test <- info_group[kI == i, "Y"]
    w_test <- info_group[kI == i, "Weight"]

    #fit a logistic regression model to the CV training data
```

```
    models[[i]] <- logistic_model$new(X=X_train, y=y_train)

    #use it to predict the classifications of the test data
    prob <- models[[i]]$predict(X_test)

    #store roc and ams data
    rocs[[i]] <- ROC_curve$new(y_test, prob)
    amss[[i]] <- AMS_data$new(y_test, prob, w_test)
  }

  plot_rocs(rocs, info=groups[g], scale=0.8)
  plot_amss(amss, info=groups[g], scale=0.8)
}
```





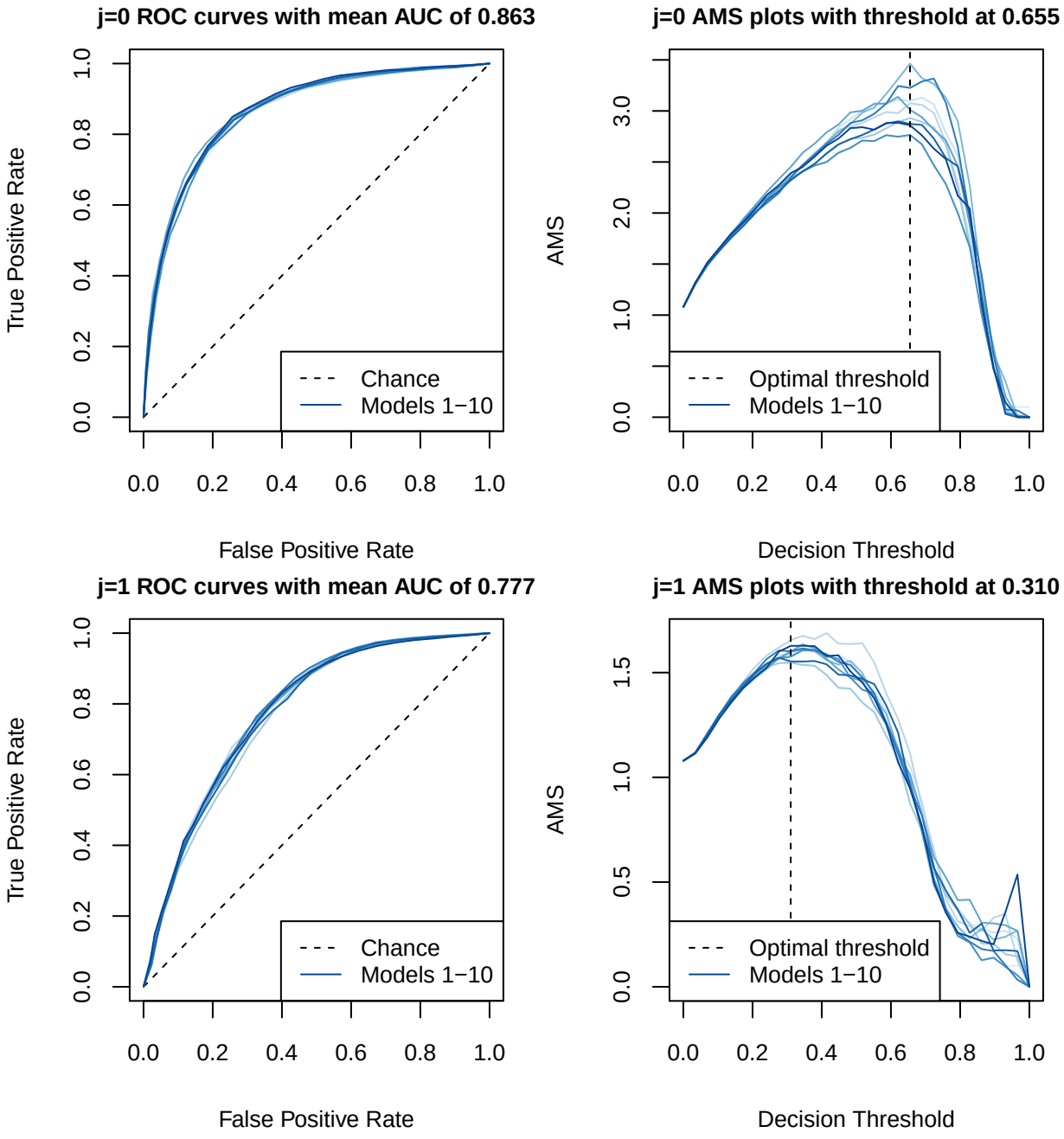
Overall the models perform pretty similarly in terms of AUC, though the j=2 group has performed slightly worse. Interestingly the AMS graphs look quite different and the optimal decision threshold is different for each group. For the j=2+ group the AMS score is extremely noisy after t=0.7.

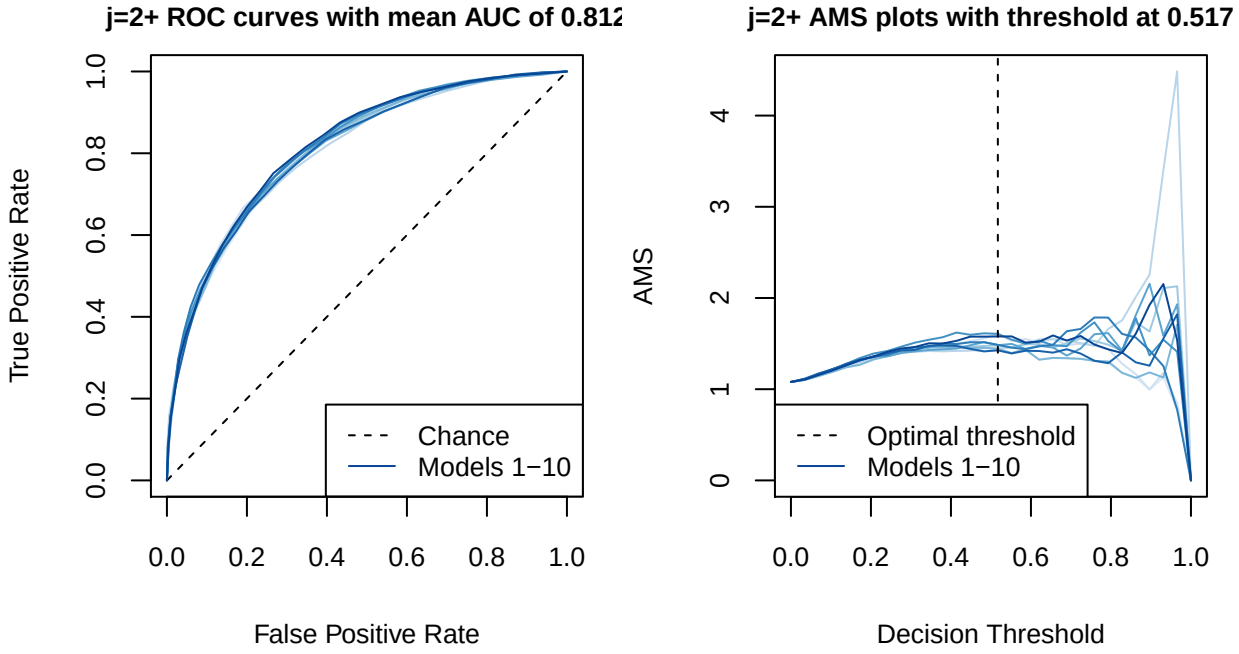
Standardising data

It is typically beneficial in regression problems to standardise the data before training. That is, to centre each variable to have a mean of 0 and standard deviation of 1. Here we'll see how replacing any remaining -999s with NAs (this will just effect the `DER_mass_MMC` column) and standardising the variables using our `scale_dat` function affects performance.

```
par(mfrow=c(1,2))
par(mar=c(4,4,2,1))
```

```
for(g in 1:G){
  X_group <- X[info$Group==groups[g], !colnames(X) %in% features_to_rm[[g]]]
  info_group <- info[info$Group==groups[g],]

  # get an index for CV groups
  k <- 10
  kI <- partition_data(n=nrow(X_group), k=k, random=T)

  #create lists to hold the k models and k roc curves
  models <- vector("list", k)
  rocs <- vector("list", k)
  amss <- vector("list", k)

  for(i in 1:k){
    X_train <- X_group[kI != i,]
    y_train <- info_group[kI != i, "Y"]

    X_test <- X_group[kI == i,]
    y_test <- info_group[kI == i, "Y"]
    w_test <- info_group[kI == i, "Weight"]

    #scale the training data, and scale the test data with the same transformation
    X_train_scaled <- scale_dat(X_train, X_train)
    X_test_scaled <- scale_dat(X_test, X_train)

    #fit a logistic regression model to the CV training data
    model <- logistic_model$new(X=X_train_scaled, y=y_train)

    #use it to predict the classifications of the test data
    prob <- model$predict(X_test_scaled)

    #store roc and ams data
    rocs[[i]] <- ROC_curve$new(y_test, prob)
    amss[[i]] <- AMS_data$new(y_test, prob, w_test)
  }

  plot_rocs(rocs, info=groups[g], scale=0.8)
  plot_amss(amss, info=groups[g], scale=0.8)
}
```


j=0 ROC curves with mean AUC of 0.863
True Positive Rate
False Positive Rate
Chance
Models 1−10
j=0 AMS plots with threshold at 0.655
AMS
Decision Threshold
Optimal threshold
Models 1−10
j=1 ROC curves with mean AUC of 0.777
True Positive Rate
False Positive Rate
Chance
Models 1−10
j=1 AMS plots with threshold at 0.310
AMS
Decision Threshold
Optimal threshold
Models 1−10



We can see that standardising the data has improved performance for all three groups, though the j=2+ group still has noisy AMS. So far we have just used a standard logistic regression which finds a linear boundary and so performance is likely to improve when we add some non-linearity to the models.

SVM

In the `lhc` package we have implemented two support vector machine (SVM) algorithms: `svm` and `kernel_svm`. In this section of the notebook we attempt to apply these methods to the Higgs boson classification problem.

It is expected that soft-margin SVM will have a similar performance to logistic regression. Kernel SVM could improve performance by transforming the data and creating a non-linear decision boundary. Unfortunately, SVM algorithms do not scale well with n and so we are unlikely to be able to train our models on a significant portion of the training data.

Our SVM functions take as inputs: an $n \times d$ design matrix $\mathbf{X}$, binary class labels $\mathbf{y} \in \{-1, 1\}^n$, a parameter C to controls the relative weighting of maximising the margin vs minimising the slack variables. For `kernel_svm` a kernel function k is also needed.

The functions use the function `solve.QP` from the `quadprog` package to solve the Lagrangian optimisation problem:

$$L(\boldsymbol{\lambda}) = -\frac{\boldsymbol{\lambda}((\mathbf{y}\mathbf{y}^T) \circ (\mathbf{X}\mathbf{X}^T))\boldsymbol{\lambda}}{2} + \langle \mathbf{1}, \boldsymbol{\lambda} \rangle$$

subject to

$$0 \leq \lambda_i \leq C, \qquad \sum_i \lambda_i y_i = 0$$

For `kernel_svm`, $\mathbf{X}\mathbf{X}^T$ is replaced with $\mathbf{K}$, where $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$.

Finally, the functions return a function/trained model which predicts the classes of new samples. For example, the prediction function for the soft-margin SVM takes the form $y_i = \mathbf{w}^T\mathbf{x}_i + w_0$.

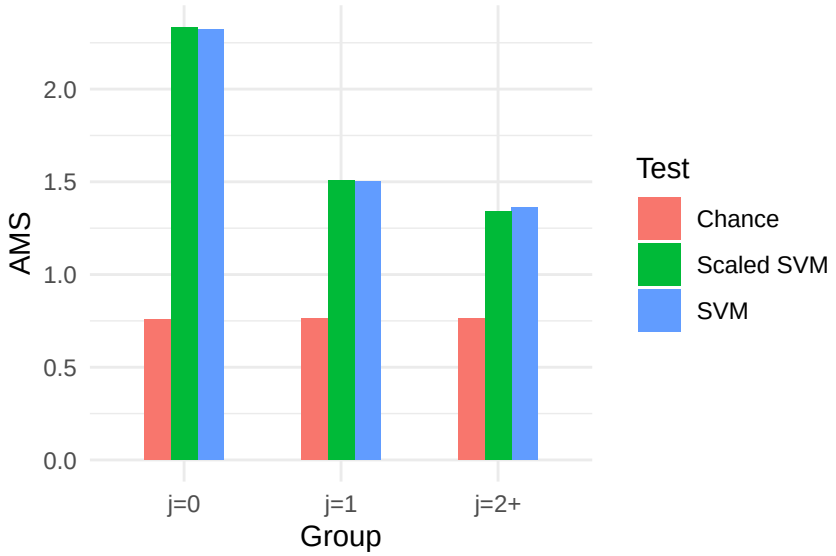
SVM algorithms return the predicted class of samples $\hat{y}_i$ not the probability $P(y_i = s|\mathbf{x}_i)$. This means we cannot use our `ROC_curve` and `AMS_data` object classes which vary a decision threshold, and we so calculate a single AMS value for the test set.

Soft-margin SVM

First we run a first pass of the soft-margin SVM on each of our groups using a small training set. We test how standardising the data before fitting affects performance, and how performance compares to chance (random assignment of classes).

```
#for SVM recode Y as -1 and 1
info$Y[info$Y == 0] <- -1

#Set seed so random samples are reproducible
set.seed(110)

#create a results table to store AMS
results <- as.data.frame(matrix(NA, nrow=G, ncol=4))
results[,1] <- groups
colnames(results) <- c("Group", "SVM", "Scaled SVM", "Chance")

#loop through the groups
n <- 500
for(g in 1:G){
  #select samples from this group
  X_group <- X[info$Group==groups[[g]], !colnames(X) %in% features_to_rm[[g]]]
  info_group <- info[info$Group==groups[[g]], ]

  #randomly select n sample to train, and use the rest as test
  ind <- c(rep(0, n), rep(1, nrow(X_group) - n))
  kI <- sample(ind, nrow(X_group))

  X_train <- X_group[kI != 1,]
  y_train <- info_group[kI != 1, "Y"]

  X_test <- X_group[kI == 1,]
  y_test <- info_group[kI == 1, "Y"]
  w_test <- info_group[kI == 1, "Weight"]

  #train a model on unscaled data
  model <- svm(X_train, y_train, C=1)

  #use the resulting function to predict the classes of the training samples
  y_pred <- model(X_test)

  #calculate AMS
  results[g, 2] <- ams_metric(y_test, y_pred, w_test)

  #scale the training data, and scale the test data with the same transformation
  X_test_scaled <- scale_dat(X_test, X_train, add.intercept = F)
  X_train_scaled <- scale_dat(X_train, X_train, add.intercept = F)

  #train a model on scaled data and calculate AMS
  model <- svm(X_train_scaled, y_train, C=1)
  y_pred_scaled <- model(X_test_scaled)
  results[g, 3] <- ams_metric(y_test, y_pred_scaled, w_test)

  #randomly assign classes and calculate AMS
  y_rand <- sample(c(-1, 1), length(y_test), replace=T)
```

```
  results[g, 4] <- ams_metric(y_test, y_rand, w_test)
}

results %>%
  pivot_longer(cols=-"Group", names_to="Test", values_to="AMS") %>%
  ggplot(aes(x=Group, y=AMS, fill=Test)) +
    geom_bar(position="dodge", stat = "identity", width=0.5) +
    theme_minimal()
```



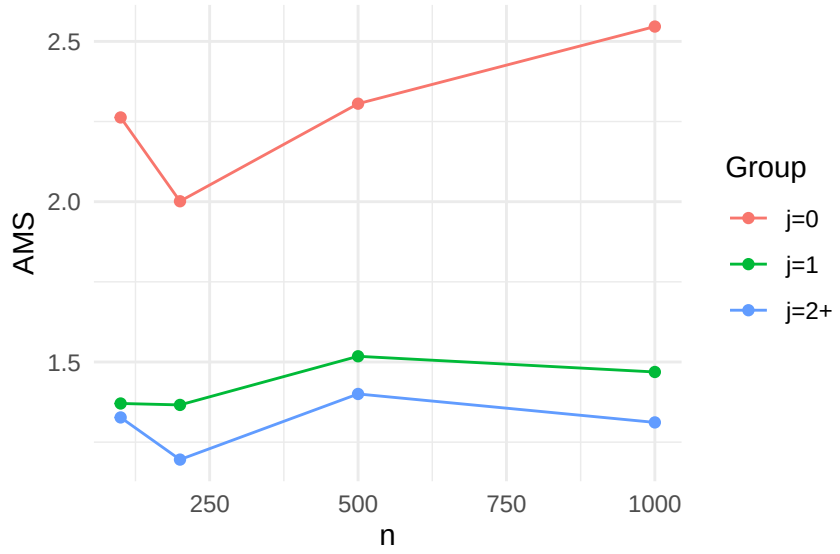
Interestingly, the SVM models trained on only 500 samples perform almost as well as the logistic regression. It's likely that this is around the best performance possible with a linear boundary. Scaling the data before training appears to have a negligible impact.

Training size

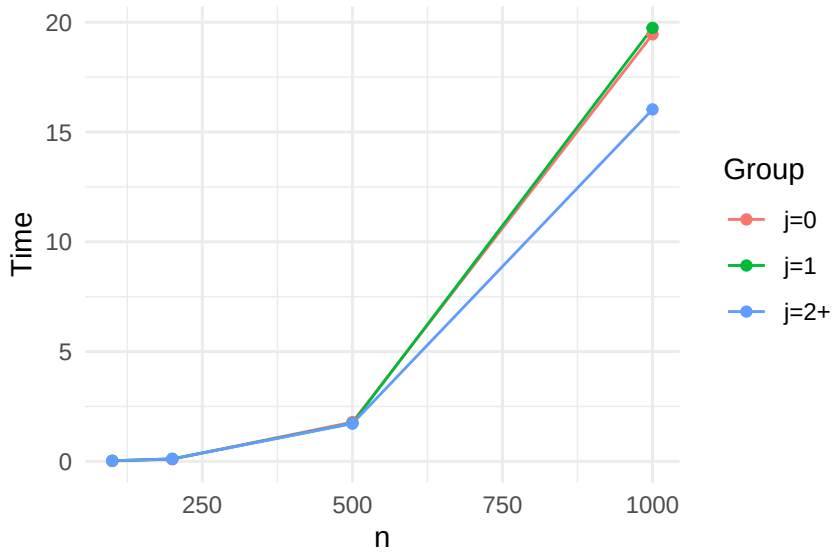
How does increasing n affect time and performance?

```
#create a results table to store AMS
sizes <- c(100, 200, 500, 1000)
results <- as.data.frame(matrix(NA, nrow=G, ncol=length(sizes)+1))
results[,1] <- groups
colnames(results) <- c("Group", sizes)

#and a table to store the times
times <- results

for(i in 1:length(sizes)){
  n <- sizes[i]
  for(g in 1:G){
    X_group <- X[info$Group==groups[[g]], !colnames(X) %in% features_to_rm[[g]]]
    info_group <- info[info$Group==groups[[g]], ]

    ind <- c(rep(0, n), rep(1, nrow(X_group)-n))
    kI <- sample(ind, nrow(X_group))

    X_train <- X_group[kI == 0,]
    y_train <- info_group[kI == 0, "Y"]
```

```
    X_test <- X_group[kI == 1,]
    y_test <- info_group[kI == 1, "Y"]
    w_test <- info_group[kI == 1, "Weight"]

    time_check<- function(){
      model <- svm(X_train, y_train, C=1)
      y_pred <- model(X_test)
      return(y_pred)
    }

    times[g, i+1] <- system.time(y_pred <- time_check())["elapsed"]

    results[g, i+1] <- ams_metric(y_test, y_pred, w_test)
  }
}

results %>%
  pivot_longer(cols=-"Group", names_to="n", values_to="AMS") %>%
  mutate(n=as.numeric(n)) %>%
  ggplot(aes(x=n, y=AMS, colour=Group)) +
    geom_line() +
    geom_point() +
    theme_minimal()
```




```
times %>%
  pivot_longer(cols=-"Group", names_to="n", values_to="Time") %>%
  mutate(n=as.numeric(n)) %>%
  ggplot(aes(x=n, y=Time, colour=Group)) +
    geom_line() +
    geom_point() +
    theme_minimal()
```



We can see that performance (AMS score) does not change much when the sample size is doubled from 500 to 1000, but the training time increases with approximately $\mathcal{O}(n^3)$. The group with no jets seems to perform consistently better than the other two groups.

Ensemble SVM

Ensemble SVM is a technique where multiple SVM models are trained on subsets of the training data. To classify new samples, the majority vote from the models is taken. Dividing a dataset of n samples over m models reduces the computational complexity of an $\mathcal{O}(n^2)$ algorithm to $\mathcal{O}(n^2/m)$.

```
m <- 10
n <- 500

results <- as.data.frame(matrix(NA, nrow=G, ncol=m+2))
results[,1] <- groups
colnames(results) <- c("Group", paste("Model", 1:m), "Ensemble")

#use a different parameter C for each model
params <- sample(1:50, m)

for(g in 1:G){
  X_group <- X[info$Group==groups[[g]], !colnames(X) %in% features_to_rm[[g]]]
  info_group <- info[info$Group==groups[[g]], ]

  ind <- c(rep(1:m, n), rep(0, nrow(X_group)-m*n))
  kI <- sample(ind, nrow(X_group))

  X_test <- X_group[kI==0, ]
  y_test <- info_group[kI==0, "Y"]
  w_test <- info_group[kI==0, "Weight"]

  #store test predictions from m models
  y_predictions <- matrix(NA, ncol=m, nrow=nrow(X_test))

  #for each of the k groups, fit a svm, and calc ams
  for(i in 1:m){
    X_train <- X_group[kI == i,]
```

```
    y_train <- info_group[kI == i, "Y"]

    model <- svm(X_train, y_train, C=params[i])
    y_pred <- model(X_test)
    y_predictions[,i] <- y_pred

    results[g, i+1] <- ams_metric(y_test, y_pred, w_test)
  }

  #now find the ams from majority vote
  y_pred <- sign(rowMeans(y_predictions, na.rm=T))
  results[g, m+2] <- ams_metric(y_test, y_pred, w_test)
}


plot_data <- results %>%
  pivot_longer(cols=-"Group", names_to="Model", values_to="AMS") %>%
  mutate(Ensemble = grepl("Ensemble", Model))

p <- ggplot(plot_data[!plot_data$Ensemble,], aes(x=Group, y=AMS, colour=Group)) +
  geom_boxplot() +
  geom_point(data=plot_data[plot_data$Ensemble,], aes(fill=Ensemble), colour="black") +
  theme_minimal()

pdf("../doc/figs/Ensemble_SVM.pdf", width=5, height=4)
p
dev.off()
```

```
## pdf
##   2
```

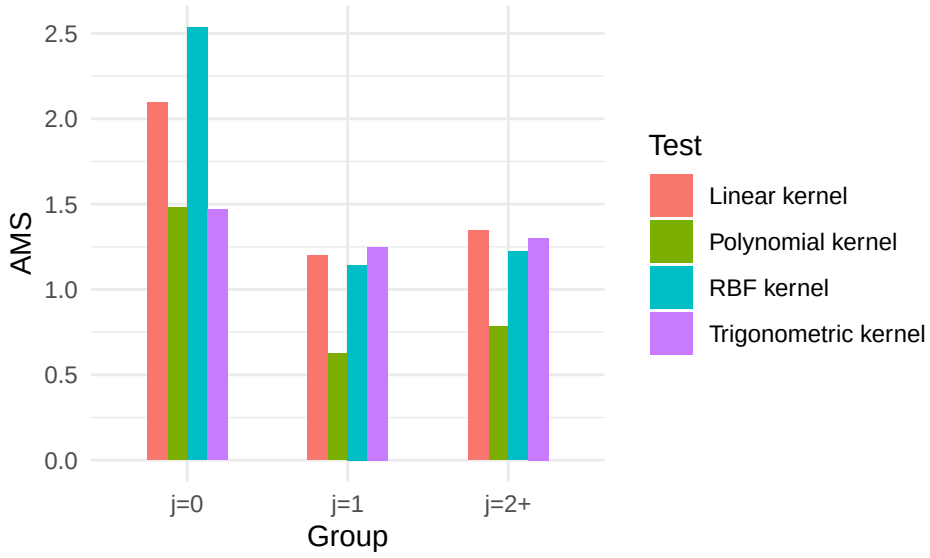
The ensemble SVM outperforms the individual models and and scales better with n, however the improvements in AMS are fairly modest.

Kernel SVM

Now we test out our implementation of kernel SVM, with a selection of different kernels on each group. Unfortunately our implementation scales badly with n and so only small training and test sets can be used (predictions are also computational expensive).

```
train_n <- 200
test_n <- 2000

kernels <- list(
  "Polynomial kernel" = tuned_kernel(poly_kernel, b=2),
  "RBF kernel" = tuned_kernel(rbf_kernel, sigma=10),
  "Trigonometric kernel" = tuned_kernel(trig_kernel, b=5),
  "Linear kernel" = tuned_kernel(lin_kernel)
)

results <- as.data.frame(matrix(NA, nrow=G, ncol=length(kernels)+1))
results[,1] <- groups
colnames(results) <- c("Group", names(kernels))

for(g in 1:G){
```

```
  X_group <- X[info$Group==groups[[g]], !colnames(X) %in% features_to_rm[[g]]]
  info_group <- info[info$Group==groups[[g]], ]

  for(k in 1:length(kernels)){
    #create index training samples = 1, test samples = 2, other = 0
    ind <- c(rep(1, train_n), rep(2, test_n), rep(0, nrow(X_group)-train_n-test_n))
    kI <- sample(ind, nrow(X_group))

    X_train <- X_group[kI==1, ]
    y_train <- info_group[kI==1, "Y"]

    X_test <- X_group[kI==2, ]
    y_test <- info_group[kI==2, "Y"]
    w_test <- info_group[kI==2, "Weight"]

    #scale data so values in poly kernel aren't too large
    X_test_scaled <- scale_dat(X_test, X_train, add.intercept = F)
    X_train_scaled <- scale_dat(X_train, X_train, add.intercept = F)

    ckernel <- kernels[[k]]
    model <- kernel_svm(X_train_scaled, y_train, C=1, ckernel)
    y_pred <- model(X_test_scaled)
    results[g, k+1] <- ams_metric(y_test, y_pred, w_test)
  }
}

results %>%
  pivot_longer(cols=-"Group", names_to="Test", values_to="AMS") %>%
  ggplot(aes(x=Group, y=AMS, fill=Test)) +
    geom_bar(position="dodge", stat = "identity", width=0.5) +
    theme_minimal()
```



The RBF, trigonometric and polynomial kernels perform quite similarly in this trial. To improve performance we could do a grid search for the optimal hyperparameters with cross validation, however we are restricted by the time complexity of the kernel SVM function and so we instead focus on feature engineering the logistic regression model.

Analyse Experiments

As mentioned previously, we ran a series of experiments running a logistic regression model with varying model parameters. The experiments were run in a separate script `RunModels.R`, and the output of the experiments are saved in the `output` folder of the git repository.

Import results

```
filepath <- path_join(c(dirname(getwd()), "/output/results_6.csv"))
exp_data <- read.csv(filepath) %>% select(-X, -C, -K)
```

Experiments were run over all combinations of the following parameters:

```
#the number of rbf features
(n_rbfs <- unique(exp_data$n_rbf))
```

```
## [1] 0 2 4 1 3
```

```
#the l2 regularisation parameter
(lambdas <- unique(exp_data$lambda))
```

```
## [1]  0.000100000  0.044721360 20.000000000  0.004641589  0.215443469
## [6] 10.000000000
```

```
#and the degree of polynomial transform (with no interaction terms)
(poly_orders <- unique(exp_data$poly))
```

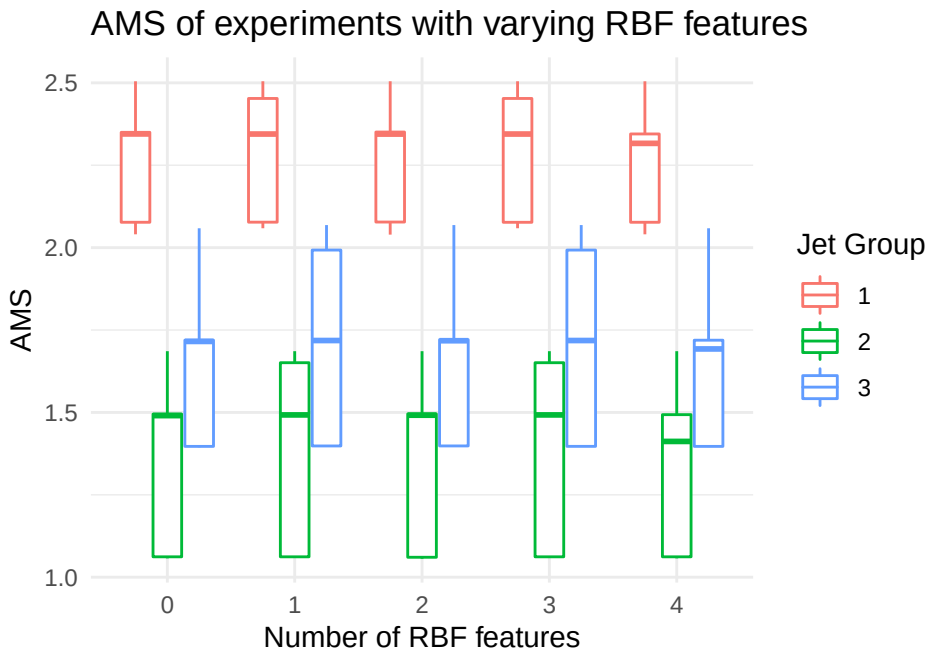
```
## [1] 1 2 3
```

Since we want a model with high performance but low variance, we measured the median absolute deviation (mad) of the AMS and AUC metrics over the k-folds. Now we can look to maximise the mean(AMS)/mad(AMS).

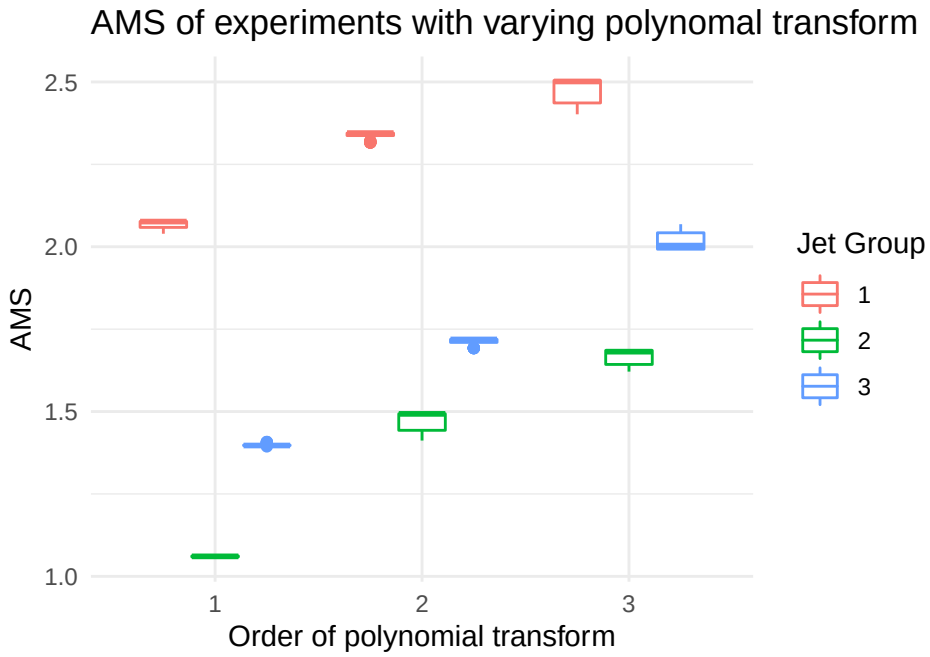
```
# scaling metrics by MAD(metric) to get a handle on variation
exp_data$scaled.auc <- exp_data[,"auc"]/exp_data[,"mad.auc."]
exp_data$scaled.ams <- exp_data[,"ams"]/exp_data[,"mad.ams."]
```

Plot metrics

```
#convert columns to factors for boxplots
exp_data$lambda <- signif(exp_data$lambda,3)
cols <- c("G", "lambda", "poly", "n_rbf")
exp_data[cols] <- lapply(exp_data[,cols], factor)

(p1 <- exp_data %>%
  ggplot(aes(x=n_rbf, y=ams, colour=G)) +
    geom_boxplot() +
    theme_minimal() +
    labs(x="Number of RBF features", y="AMS",
         title="AMS of experiments with varying RBF features",
         colour="Jet Group"))
```

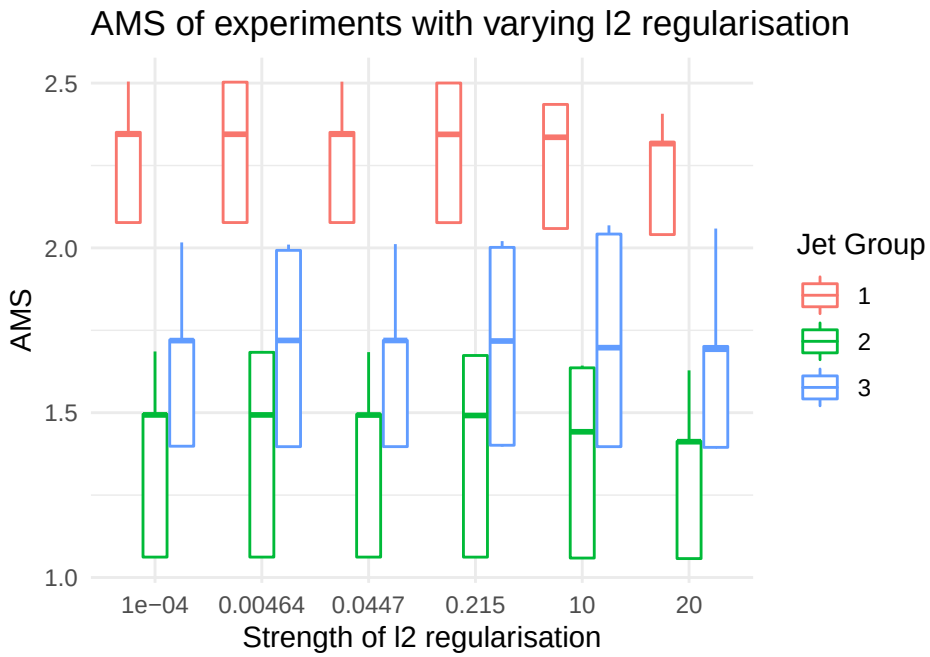



```
(p2 <- exp_data %>%
  ggplot(aes(x=poly, y=ams, colour=G)) +
    geom_boxplot() +
    theme_minimal() +
    labs(x="Order of polynomial transform", y="AMS",
         title="AMS of experiments with varying polynomal transform",
         colour="Jet Group"))
```




```
(p3 <- exp_data %>%
  ggplot(aes(x=lambda, y=ams, colour=G)) +
    geom_boxplot() +
    theme_minimal() +
```

```
    labs(x="Strength of l2 regularisation", y="AMS",
         title="AMS of experiments with varying l2 regularisation",
         colour="Jet Group"))
```



Record results

Print results as a LaTeX table that can be imported into a report easily. Not too surprisingly we see different models appear at the top of our list depending on how we decide to choose the best model. Since our goal is ultimately to maximise AMS, it makes sense that this (rather than AUC) should be the metric we choose, despite the higher variance it displays. To try and mitigate this variance, we can choose to sort by a scaled version of AMS, where we take the mean absolute deviation across the folds and scale by that, to try and optimise for the best result with the least variation and therefore hopefully the best generalisation.

The top models appears to have the following set of parameters: $n_{rbf} = 2$, $\lambda = 1e-4$ and the inclusion of up to 3rd order polynomial terms.

```
library(Hmisc)
#For each group find the experiment with the highest ams
output <- NULL
for(i in 1:3){
  exp_sub <- filter(exp_data, G==i) %>%
    arrange(desc(ams))

  output <- rbind(output, exp_sub[1,])
}

#filter columns
output <- select(output,
                 G, n_rbf, lambda, poly, auc, ams, mad.ams., scaled.ams)
output[,5:8] <- round(output[,5:8], 3)

# make column names latex friendly
colnames(output) <- sub("(\\w+)\\.(\\w+)\\.?", "\\$\\1_\\{\\2\\}\\$", colnames(output))
```

Table 1: Results table of the top 5 experiments

output	n_{rbf}	lambda	poly	auc	mad_{auc}	ams	mad_{ams}	$scaled_{auc}$	$scaled_{ams}$
1	3	0.000	3	0.876	0.009	2.507	0.118	100.399	21.334
2	0	0.001	3	0.876	0.007	2.508	0.127	130.466	19.742
3	5	0.000	2	0.869	0.010	2.386	0.121	87.225	19.695
4	1	0.000	2	0.869	0.010	2.386	0.122	87.087	19.629
5	2	0.000	2	0.869	0.010	2.386	0.122	87.087	19.629

```
colnames(output) <- sub("n\\_rbf", "\\$n_{rbf}\\$", colnames(output))

# Generate LaTeX table
latex(output, file=path_join(c(dirname(getwd()), "doc/results_table_groups.tex")), caption="Results tabl
```

Logistic Regression and Feature Engineering

In an effort to include higher order features we decided to implement feature augmentation in the form of polynomial transformations and RBF centroid features. We created a method in which we could vary the features easily with a few parameters so that we could run a series of experiments, with 10-fold cross-validation. Then we can compare the performance of the models to obtain a "best" model to apply to our hold-out validation set (the private leaderboard set "v").

From the exploratory data analysis we saw that the data can be viewed as having been generated from different underlying processes, based on the structure of the missing data. Although we considered building independent models for all 6 of the missing data patterns, we found that some of the groups were too small to build reliable models. As a result we opted to divide only by jet number (rather than Higgs mass as well) and remove the remaining missing columns as required.

A set of functions (project_funcs.r) are included in the `lhc` package to streamline the pipeline for the model experiments. Example functions are `import_data` to load and pre-process the Kaggle dataset, and `idx_jet_cat` which indexes samples from the jet number groups.

Initialise script

Set script parameters:

```
set.seed(22)

# Regularisation (L2) parameter [global]
lambda <- c(0.00464, 1e-4, 10)
# constraint parameter for L1 Logistic regression [global]
C <- 1
# number of jet/Higgs mass groups to build different models for
G <- 3
poly_order <- c(3, 3, 3)
n_rbf <- c(1, 0, 1)

# choose model. Interchangeable so long as we have implemented polymorphism
# across the model classes so that they share a common interface. Use "partial"
# to call with optional parameters to enforce an interface of f(X,y)
# L2 logistic regression
model_init <- c(partial(logistic_model$new, lambda=lambda[1]), partial(logistic_model$new, lambda=lambda

# Constrained logistic regression (using CVXR)
```

```
# model_init <- partial(logistic_l1_model$new, C=C)

# pick training and validation sets
train_label = c("t")
val_label <- c("v")

# bool to choose whether to save outputs
if (!("do_save_outputs" %in% ls())) {
  do_save_outputs <- TRUE
}
# dir to save figures
fig_dir <- path_join(c(dirname(getwd()), "doc/figs/"))
```

Load data

```
filepath <- "../../atlas-higgs-challenge-2014-v2.csv"
data <- import_data(filepath)
```

Assign variables corresponding to the training set.

```
train_idx <- get_subset_idx(data$kaggle_s, train_label)
X <- data$X[train_idx, ]
y <- data$y[train_idx]
kaggle_w <- data$kaggle_w[train_idx]
w <- data$kaggle_w[train_idx]
nj <- data$nj[train_idx]
e_id <- data$e_id[train_idx]
```

Assign variables corresponding to the validation set.

```
# public leaderboard set
val_idx <- get_subset_idx(data$kaggle_s, val_label)
Xv <- data$X[val_idx, ]
yv<- data$y[val_idx]
wv <- data$kaggle_w[val_idx]
njv <- data$nj[val_idx]
```

Feature engineering

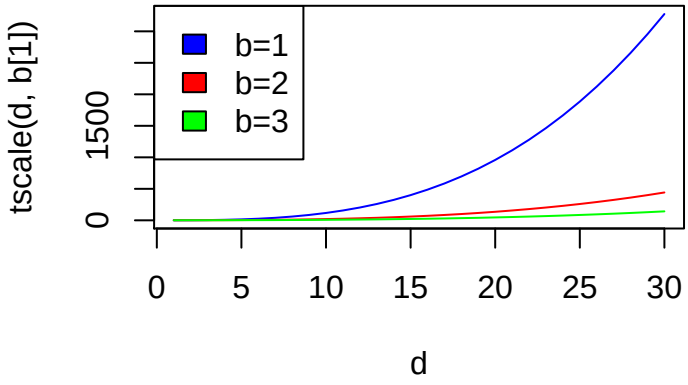
Apply some feature engineering

```
# modify features
X <- reduce_features(X)
X <- invert_angle_sign(X)
Xv <- reduce_features(Xv)
Xv <- invert_angle_sign(Xv)

# if (poly_order > 1) {
#   X <- poly_transform(X, poly_order)
#   Xv <- poly_transform(Xv, poly_order)
# }

# ensure X and Xv have the same columns
# cols2keep <- intersect(colnames(X), colnames(Xv))
# Xv <- Xv[, cols2keep]
```

```
# X <- X[, cols2keep]
```

Should we include interaction terms? Plot computation scaling as a function of number of initial features d and order of polynomial terms of those to include, if we were to include all pairwise interactions.

```
nint <- function(d) {
  exp(lgamma(d+1) - log(2) - lgamma(d-1))
}
tscale <- function(d,b) {
  dI <- nint(d)
  1 + dI^3/(b*d)^3 + dI^2/(b*d)^2 + dI/(b*d)
}
d <- 1:30
b <- 1:3
plot(d, tscale(d,b[1]), type="l", col="blue")#, log="y")
lines(d, tscale(d,b[2]), col="red")
lines(d, tscale(d,b[3]), col="green")
legend("topleft", legend=c("b=1", "b=2", "b=3"), fill=c("blue", "red", "green"))
```



Compute some initial parameters

```
s <- avg_median_pairwise_distance(X)

# K-Fold CV partitioning
K <- 10
kI <- partition_data(length(y), K, random = TRUE)

# number of rows and columns of data matrix
n <- nrow(X)
# d <- ncol(X) + n_rbf

# set colours for jet groups
colours <- generate_colours(G)
```

Use missing data pattern to partition data.

```
# get missing rows. separate by number of jets and presence of Higgs mass and
# fit separate models
# find columns with features with any missing values for each number of
# jets: 0, 1, 2+
jet_cats <- c(1:3, 1:3)
features_to_rm <- set_features_to_rm(X, G, kI, nj)
```

Run CV experiments

Loop over jet groups and folds and fit a model for each (so GK total) and then test each of these on their OOS subset, recording AUC and AMS.

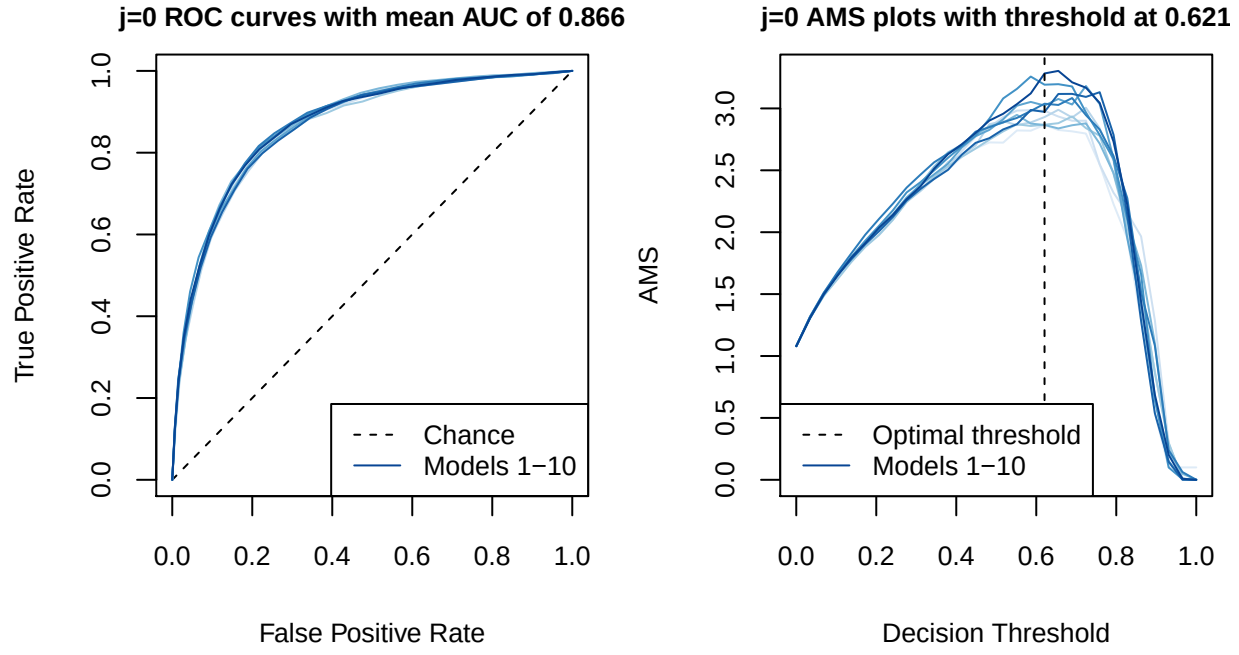
```
# loop over folds
#create lists to hold the k models and k roc curves
models <- vector("list", G*K)
rocs <- vector("list", G*K)
ams_obj <- vector("list", G*K)
b <- matrix(NA, nrow=d, ncol=G*K)

ams <- rep(NA, length=G*K)
auc <- rep(NA, length=G*K)

par(mfrow=c(2, 3))
for (mj in 1:G) {
    # loop over sets of jet number {0, 1, 2+} and mH presence/absence
    for (k in 1:K) {
        j <- jet_cats[mj]
        # get train and test split indices for this jet category and fold
        fit_row_idx <- kI != k & idx_jet_cat(nj, j)
        test_row_idx <- kI == k & idx_jet_cat(nj, j)

        Xtrain <- poly_transform(X[fit_row_idx, ], poly_order[mj])
        Xtest <- poly_transform(X[test_row_idx, ], poly_order[mj])

        # add r RBF centroid features, using the same reference centroids in training and testing sets
        if (n_rbf[mj] > 0) {
          rbf_centroids <- get_rbf_centroids(Xtrain, n_rbf[mj])
          Xi <- rbf_centroids$"xi"
          Xtrain <- add_rbf_features(Xtrain, s, n_rbf[mj], Xi=Xi)
          Xtest <- add_rbf_features(Xtest, s, n_rbf[mj], Xi=Xi)
        }

        # get indices for columns to include for this jet category
        col_idx <- get_valid_cols(colnames(X), features_to_rm, mj)

        # define train and test matrices
        Xtrain <- as.matrix(Xtrain[, col_idx])
        Xtest <- as.matrix(Xtest[, col_idx])

        # standardize the data (using training as a reference for the test set to avoid using any test
        Xtest <- scale_dat(Xtest, Xtrain)
        Xtrain <- scale_dat(Xtrain, Xtrain)

        # get index of this model, from jet category and fold (to retrieve in results arrays)
        model_idx <- get_model_idx(mj, k, K)

        # fit a logistic regression model to the CV training data
        models[[model_idx]] <- model_init[[mj]](X=Xtrain, y=y[fit_row_idx])

        # use it to predict the classifications of the test data
        p_hat <- models[[model_idx]]$predict(Xtest)
```

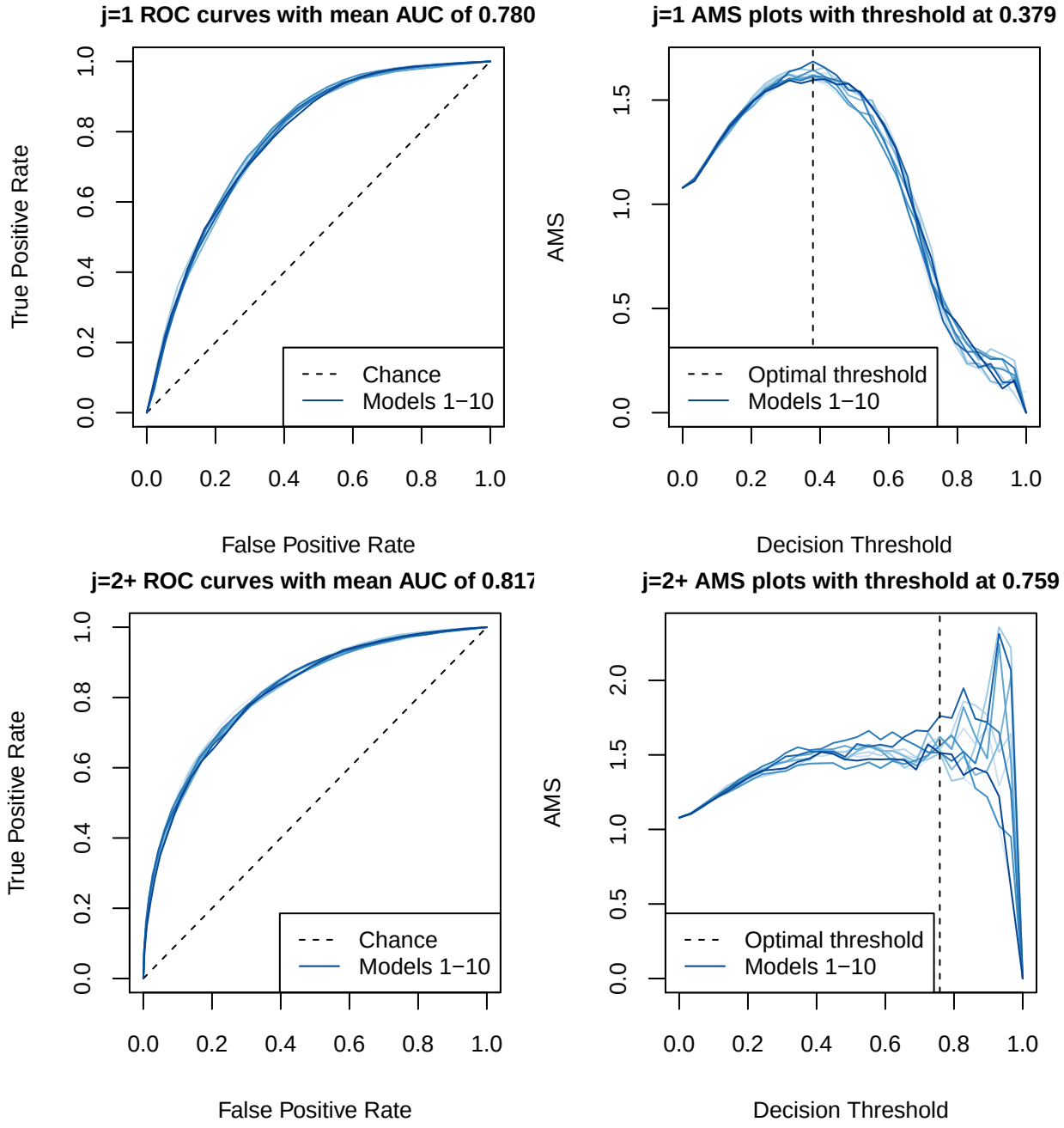
```
        # create an ROC curve object
        rocs[[model_idx]] <- ROC_curve$new(y[test_row_idx], p_hat)

        # create an AMS curve object
        w_this_partition <- w[test_row_idx]
        ams_obj[[model_idx]] <- AMS_data$new(y[test_row_idx], p_hat, w_this_partition)
    }
}
auc <- sapply(rocs, function(x) x$calc_auc())
ams <- sapply(ams_obj, function(x) x$calc_ams())
```

Plot CV metrics

Plot ROC curves and AMS against decision threshold over the k folds for each group. By plotting AMS against threshold we can get a feeling for what threshold we should use for our final model and how consistent they are across folds. To account for the noise (especially important for the group j=2+) we find the minimum curve over the K fold curves (by taking the minimum over folds at each threshold point) and then find the threshold which maximises this curve.

```
par(mfrow=c(1,2))
par(mar=c(4,4,2,1))
for(j in 1:G){
  i1 <- get_model_idx(j, 1, K)
  i2 <- get_model_idx(j, K, K)

  plot_rocs(rocs[i1:i2], info=groups[j])
  plot_amss(ams_obj[i1:i2], info=groups[j])
}
```





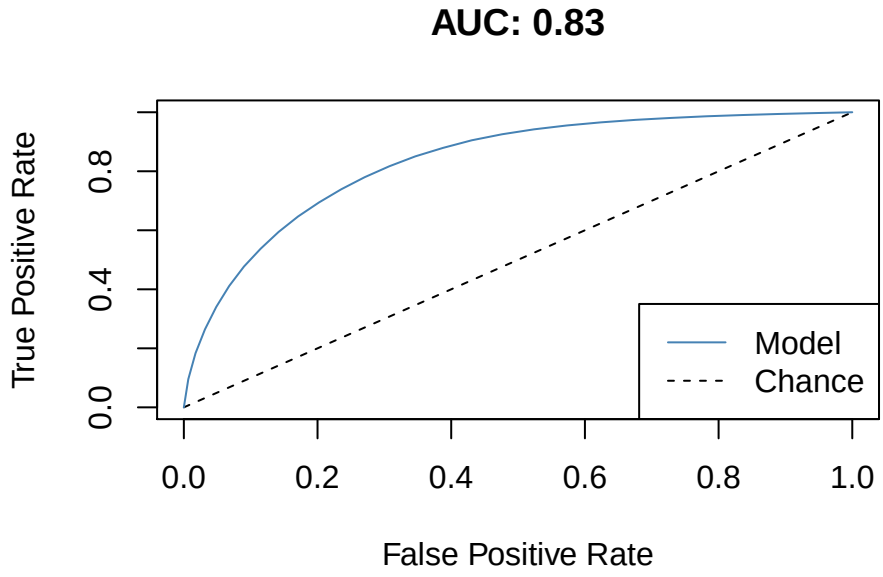

```
#select decision thresholds for each group based on CV results
thresholds <- c(0.6, 0.4, 0.7)
```

Run full model

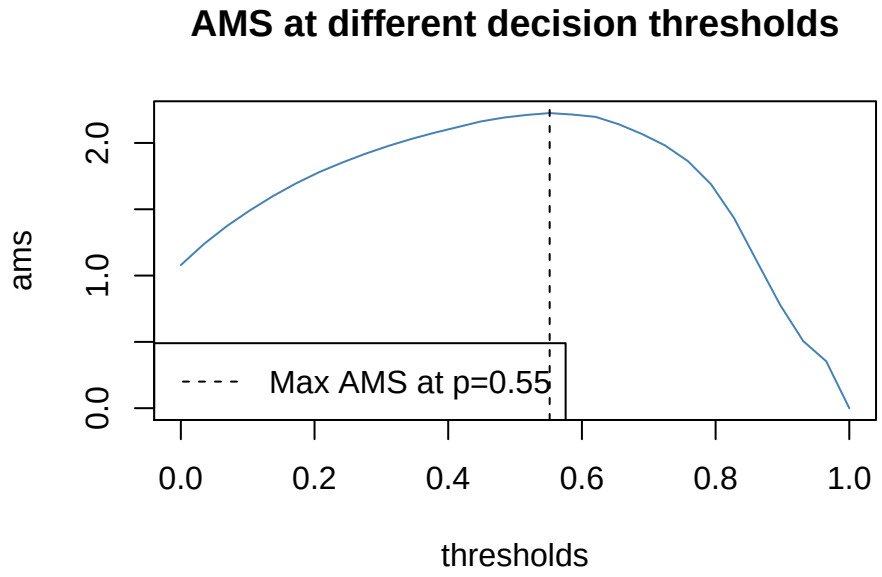
Fit model (for each category) on whole (unfolded) dataset

```
predictions <- rep(NA, length(Xv))
rbf_idx <- matrix(NA, nrow=G, ncol=n_rbf)

for (mj in 1:G) {
    # loop over sets of jet number {0, 1, 2+} and mH presence/absence
    j <- jet_cats[mj]
```

```
    fit_row_idx <- idx_jet_cat(nj, j) #& idx_higgs_mass(X, mj, G)
    test_row_idx <- idx_jet_cat(njv, j)# & idx_higgs_mass(Xv, mj, G)

    Xtrain <- poly_transform(X[fit_row_idx, ], poly_order[mj])
    Xtest <- poly_transform(Xv[test_row_idx, ], poly_order[mj])

    if (n_rbf[mj] > 0) {
      # add r RBF centroid features, using the same reference centroids in training and testing sets
      rbf_centroids <- get_rbf_centroids(Xtrain, n_rbf[mj])
      Xi <- rbf_centroids$"xi"
      # record which rows to use for OOS in public set (or any other OOS)
      Xtrain <- add_rbf_features(Xtrain, s, n_rbf[mj], Xi=Xi)
      Xtest <- add_rbf_features(Xtest, s, n_rbf[mj], Xi=Xi)
    }

    # get indices for columns to include for this jet category
    col_idx <- get_valid_cols(colnames(X), features_to_rm, mj)
    # standardize the data
    Xtrain <- as.matrix(Xtrain[, col_idx])
    Xtest <- as.matrix(Xtest[, col_idx])
    Xtest <- scale_dat(Xtest, Xtrain)
    Xtrain <- scale_dat(Xtrain, Xtrain)

    #fit a logistic regression model to the all of the training data
    model <- model_init[[mj]](X=Xtrain, y=y[fit_row_idx])

    #use it to predict the classifications of the test data
    p_hat <- model$predict(Xtest)

    predictions[test_row_idx] <- p_hat
}

#create objects for overall results
full_roc <- ROC_curve$new(yv, predictions)
amsv_obj <- AMS_data$new(yv, predictions, wv)

full_roc$plot_curve()
```




```
amsv_obj$plot_ams()
```




```
#if we use the threshold of 0.6 for each group we get an overall AMS of
y_pred <- NA * predictions
for (mj in 1:G) {
  j <- jet_cats[mj]
  idx <- idx_jet_cat(njv, j)
  y_pred[idx] <- predictions[idx] >= thresholds[mj]
}

result_amsv <- ams_metric(yv, y_pred, wv)
result_aucv <- full_roc$auc
```

Summary results

Summary results:

```
get_threshold_ams <- function(ams_mat, G, K, thresholds) {
  ams <- rep(NA, G * K)
  for (g in 1:G) {
    thresh_idx <- which.min(abs(ams_obj[[1]]$thresholds - thresholds[g]))
    model_set <- ((g-1)*K+1):(g*K)
    ams[model_set] <- ams_mat[thresh_idx, model_set]
  }
  return(ams)
}

get_mean_mad_ams <- function(ams, G, K) {
  mads <- rep(NA, G)
  for (g in 1:G) {
    model_set <- ((g-1)*K+1):(g*K)
    mads[g] <- mad(ams[model_set])
  }
  return(mean(mads))
}

result_ams <- get_threshold_ams(ams, G, K, thresholds)
mad_ams <- get_mean_mad_ams(result_ams, G, K)

sprintf("Results for lambda=%.2g, G=%i, n_rbf=%i, K=%i on training set ('%s') and validation set ('%s')"
```

```
## [1] "Results for lambda=0.0046, G=3, n_rbf=1, K=10 on training set ('t') and validation set ('v')"
## [2] "Results for lambda=0.0001, G=3, n_rbf=0, K=10 on training set ('t') and validation set ('v')"
## [3] "Results for lambda=10, G=3, n_rbf=1, K=10 on training set ('t') and validation set ('v')"
```

```
sprintf("CV OOS AUC = %.2f ± %.1f", mean(auc), mad(auc))
```

```
## [1] "CV OOS AUC = 0.82 ± 0.1"
```

```
sprintf("CV OOS AMS = %.2f ± %.1f", mean(result_ams), mad_ams)
```

```
## [1] "CV OOS AMS = 2.03 ± 0.1"
```

```
sprintf("Validation set AUC = %.2f", result_aucv)
```

```
## [1] "Validation set AUC = 0.83"
```

```
sprintf("Validation set AMS = %.2f", result_amsv)
```

```
## [1] "Validation set AMS = 2.00"
```

Save final results to a LaTeX table.

```
library(Hmisc)

# keep top 5 rows
output <- t(matrix(c(mean(auc), mad(auc), mean(result_ams), mad_ams, result_aucv, NA, result_amsv, NA),
colnames(output) <- c("AUC", "mad.AUC", "AMS", "mad.AMS")
output <- data.frame(output)
output <- mutate_if(output, is.numeric, round, digits=3)
row.names(output) <- c("CV OOS", "Test set")

# Generate LaTeX table
if (do_save_outputs) {
```

Table 2: Results for $\lambda = (0.0046, 0.0001, 10)$, $G = 3$, $n_{rbf} = (1, 0, 1)$, $b = (3, 3, 3)$, K=10 on training set ('t') and validation set ('v')

output	AUC	mad.AUC	AMS	mad.AMS
CV OOS	0.821	0.052	2.029	0.072
Test set	0.833		1.998	

```
  latex(output, file=path_join(c(dirname(getwd()), "doc/final_results_table.tex")), caption=sprintf("Res
}
```