PU learning, unbalanced data, and the missing boson

2023-01-16

Introduction

For this group project, we analysed the dataset from the HiggsML challenge ran by CERN in 2013 [3]. This contains data of simulated particle decays, the task being to distinguish between signal samples (Higgs boson decays) and background samples (non-Higgs decays). Rather than trying to solve the challenge's original task we decided to use the dataset in an investigation of positive and unlabelled (PU) classification with varying class priors.

In PU classification, some proportion (referred to in the code as `pu.pi`) of the positive training samples (which we chose to mean *background* training samples) have their labels removed (in reality this works by giving them the same label as the negative training samples). This leads to a more difficult classification task in general, however, we investigate the use of per-class costs as described by de Plessis [4] with the hopes of improving performance of our classifiers.

In real-world experiments, Higgs boson decays are much less common that the decays of the background samples, meaning that the signal and background classes have greatly imbalanced priors, which would cause the dataset to contain far fewer signal than background samples, potentially causing problems for our classifiers. Luckily, the dataset actually uses *simulated* particle decays, which allows the number of signal samples and background samples to be closer together. However, the problem of imbalanced priors is very interesting, particularly in the way it intereracts with PU data, hence we shall be creating new datasets from the original which have varying class priors (referred to in the code as `prop.signal`).

A more detailed description of our methodology and the origin of the dataset can be found in the accompanying report, this document exists more to provide the code that used in the final analysis that makes up the report.

Data generation

First we load the data and perform a train/test/validation split according to the labels provided by the Kaggle challenge.

```
set.seed(10)

data <- read.csv("./data/atlas-higgs-challenge-2014-v2.csv",
    header = TRUE)

# Split: use test sets from original kaggle competition,
# where 'b' from public leaderboard, 'v' from private
# leaderboard.  'u' means unused, so discard this data
# completely
train.data <- data[data$KaggleSet == "t", ]
test.data <- data[data$KaggleSet %in% c("b", "v"), ]

# Shuffle
train.data <- train.data[sample(1:nrow(train.data)), ]
test.data <- test.data[sample(1:nrow(test.data)), ]
```

```r
# Further split test into test + validation
prop.validation <- 0.05
n.test <- nrow(test.data) * (1 - prop.validation)
validation.data <- test.data[(n.test + 1):nrow(test.data), ]
test.data <- test.data[1:n.test, ]
```

We can now check the class priors as given in the dataset, and also use the provided weights to see what the real-world class priors are (the weights scale the data so that they correspond to the real-world data generated by CERN's ATLAS detector in 2012).

```r
# Dataset signal prior
sum(data$Label == "s")/nrow(data)
```

```
## [1] 0.341661
```

```r
# Real-world signal prior
sum((data$Label == "s") * data$Weight)/sum(data$Weight)
```

```
## [1] 0.001680841
```

Clearly a signal prior of around 34% gives an easier classification task than a signal prior of 0.1%, however, as mentioned before, we want to add the functionality to make datasets with different class priors. To this end, we now provide some utility functions that will be useful throughout the rest of the code, the first two of which are equivalent to the previous chunk, with the exception that `KaggleWeight` allows us to calculate the real-world prior based only on a subset of the data (e.g. the training set).

```r
# Util functions for working with the data

# Calculate class priors
prop_signal <- function(df) sum(df$Label == "s")/nrow(df)
prop_signal_weighted <- function(df) sum(df[df$Label == "s",
    ]$KaggleWeight)/sum(df$KaggleWeight)

# Calculate the proportion of positive data (the column
# `pu_label` will be introduced shortly and is fairly
# self-explanatory)
prop_pu <- function(df) sum(df$pu_label == 1)/nrow(df)

# Other useful functions
is_missing <- function(x) x <= -999
l1normalize <- function(x) x/sum(x)
fillna <- function(x, fillvalue) ifelse(is.na(x), fillvalue,
    x)

# Allows us to make correlation plots of just half the
# features at a time (since there are so many, a single
# plot would be very large).  Only three plots are required
# due to the symmetry of correlations.
show_corrplot_quadrants <- function(cordf) {
    n <- nrow(cordf)
    chunk1 <- 1:(ceiling(n/2))
    chunk2 <- (ceiling(n/2) + 1):n
    corrplot(cordf[chunk1, chunk1])
    corrplot(cordf[chunk1, chunk2])
    corrplot(cordf[chunk2, chunk2])
}
```

Data Exploration

There is around 20% of the data missing from our training data set.

```
(is_missing(train.data) %>%
    sum)/(nrow(train.data) * ncol(train.data))
```

```
## [1] 0.1805774
```

We can plot the proportion of missingness for each feature:

```
non_input_features <- c("EventId", "Label", "KaggleSet", "KaggleWeight", "Weight")
input_features <- colnames(train.data) %>%
                  Filter(function(x) !(x %in% non_input_features),.)
missingness <- train.data[,input_features] %>% is_missing %>% colSums
missingness <- missingness / nrow(train.data) %>% as.vector

pdf(file="results/img/prop_missingness.pdf")
par(mar=c(12,4,4,2))
barplot(missingness,
        las=2, # rotate xaxis label
        cex.names=0.8, # xaxis font size
        ylim=c(0,1), main="Proportion of missingness for each feature")
garbage <- dev.off()
```

We can plot the correlation matrix between different features, to see if there is any redundancy.

```
pdf(file = "./results/img/corr_plot.1.pdf")
train.data.na <- train.data[, input_features]
train.data.na[is_missing(train.data.na)] <- NA
na.features <- train.data.na %>%
    apply(., MARGIN = 2, function(x) {
        is.na(x) %>%
            any
    })
train.data.complete <- train.data.na[, !na.features]  # data frame with only complete features (no rows
cordf <- train.data.complete %>%
    cor
corrplot(cordf[1:10, 1:10])
garbage <- dev.off()
```

There is probably not any redudancy in the features, despite the large correlations. The plot below shows that outliers (when we look at pairwise interactions) seem to be helpful for classification:

```
pdf(file = "./results/img/outliers.1.pdf")
mtev <- train.data$DER_sum_pt - train.data$DER_pt_h
plot(train.data$DER_sum_pt, train.data$DER_pt_h, xlab = "trans momentum (lepton + hadronic tau + jet)",
    ylab = "trans momentum (lepton + hadronic tau + MTEV)", pch = 20,
    col = c("black", "red")[(train.data$Label == "s") + 1])
garbage <- dev.off()
```

Feature engineering

Define some functions to make the data amenable for the machine learning algorithms. This includes one hot encoding:

```
# We want to one hot encode some of the jet data since it
# is really categorical rather than numerical
```

```
one_hot_encode <- function(df) {
    df$jets_0 <- df$PRI_jet_num == 0
    df$jets_1 <- df$PRI_jet_num == 1
    df$jets_2 <- df$PRI_jet_num == 2
    df$jets_geq_3 <- df$PRI_jet_num >= 3
    df$unexpected_topology <- is_missing(df$DER_mass_MMC)
    df$PRI_jet_all = ifelse(is_missing(df$PRI_jet_all_pt), -999,
        df$PRI_jet_all_pt)
    df %<>%
        select(-c("PRI_jet_num"))
    df
}
```

We also scale the data, as this is numerically advantageous for ML algorithms:

```
# Scales all numerical columns in a dataframe, but using
# data in a different data frame. Intended use: >
# train.df.scaled <- scale_using_train_data(train.df,
# train.df) > test.df.scaled <-
# scale_using_train_data(test.df, train.df)
scale_using_train_data <- function(df, train_df) {
    if (!setequal(colnames(df), colnames(train_df))) {
        print(paste0("df \\ train_df : ", setdiff(colnames(df),
            colnames(train_df))))
        print(paste0("train_df \\ df : ", setdiff(colnames(train_df),
            colnames(df))))
        stop("scale_using_train_data: train df and df have different column sets...")
    }
    numeric_cnames <- select_if(df, is.numeric) %>%
        colnames
    for (cname in numeric_cnames) {
        x <- train_df[[cname]][!is_missing(train_df[[cname]])]
        mu <- mean(x)
        sigma <- sd(x)
        df[[cname]][!is_missing(df[[cname]])] <- (df[[cname]][!is_missing(df[[cname]])] -
            mu)/sigma
    }
    df
}
```

We found that adding new hand-crafted features helps the classifiers. We take inspiration from Tim Saliman's HiggsML project [9] for this, which earned second place in the original Kaggle challenge. In total, this function generates 70 new features and adds them to the supplied dataframe.

```
create_new_features <- function(df) {
    df[is_missing(df)] <- NA  # useful to use NA in calculations below
    # abs
    df$SEB_abs_eta_tau <- abs(df$PRI_tau_eta)
    df$SEB_abs_eta_lep <- abs(df$PRI_lep_eta)
    df$SEB_abs_eta_jet1 <- abs(df$PRI_jet_leading_eta)
    df$SEB_abs_eta_jet2 <- abs(df$PRI_jet_subleading_eta)
    df$SEB_deltaeta_tau_lep <- abs(df$PRI_tau_eta - df$PRI_lep_eta)
    df$SEB_deltaeta_tau_jet1 <- abs(df$PRI_tau_eta - df$PRI_jet_leading_eta)
    df$SEB_deltaeta_tau_jet2 <- abs(df$PRI_tau_eta - df$PRI_jet_subleading_eta)
    df$SEB_deltaeta_lep_jet1 <- abs(df$PRI_lep_eta - df$PRI_jet_leading_eta)
```

```
df$SEB_deltaeta_lep_jet2 <- abs(df$PRI_lep_eta - df$PRI_jet_subleading_eta)
df$SEB_deltaeta_jet_jet <- abs(df$PRI_jet_leading_eta - df$PRI_jet_subleading_eta)
# prodeta
df$SEB_prodeta_tau_lep <- df$PRI_tau_eta * df$PRI_lep_eta
df$SEB_prodeta_tau_jet1 <- df$PRI_tau_eta * df$PRI_jet_leading_eta
df$SEB_prodeta_tau_jet2 <- df$PRI_tau_eta * df$PRI_jet_subleading_eta
df$SEB_prodeta_lep_jet1 <- df$PRI_lep_eta * df$PRI_jet_leading_eta
df$SEB_prodeta_lep_jet2 <- df$PRI_lep_eta * df$PRI_jet_subleading_eta
df$SEB_prodeta_jet_jet <- df$PRI_jet_leading_eta * df$PRI_jet_subleading_eta
# aux
SEB_deltaphi_tau_lep <- abs(df$PRI_tau_phi - df$PRI_lep_phi)
SEB_deltaphi_tau_lep <- ifelse(SEB_deltaphi_tau_lep > pi,
    2 * pi - SEB_deltaphi_tau_lep, SEB_deltaphi_tau_lep)
SEB_deltaphi_tau_jet1 <- abs(df$PRI_tau_phi - df$PRI_jet_leading_phi)
SEB_deltaphi_tau_jet1 <- ifelse(SEB_deltaphi_tau_jet1 > pi,
    2 * pi - SEB_deltaphi_tau_jet1, SEB_deltaphi_tau_jet1)
SEB_deltaphi_tau_jet2 <- abs(df$PRI_tau_phi - df$PRI_jet_subleading_phi)
SEB_deltaphi_tau_jet2 <- ifelse(SEB_deltaphi_tau_jet2 > pi,
    2 * pi - SEB_deltaphi_tau_jet2, SEB_deltaphi_tau_jet2)
SEB_deltaphi_lep_jet1 <- abs(df$PRI_lep_phi - df$PRI_jet_leading_phi)
SEB_deltaphi_lep_jet1 <- ifelse(SEB_deltaphi_lep_jet1 > pi,
    2 * pi - SEB_deltaphi_lep_jet1, SEB_deltaphi_lep_jet1)
SEB_deltaphi_lep_jet2 <- abs(df$PRI_lep_phi - df$PRI_jet_subleading_phi)
SEB_deltaphi_lep_jet2 <- ifelse(SEB_deltaphi_lep_jet2 > pi,
    2 * pi - SEB_deltaphi_lep_jet2, SEB_deltaphi_lep_jet2)
SEB_deltaphi_jet_jet <- abs(df$PRI_jet_leading_phi - df$PRI_jet_subleading_phi)
SEB_deltaphi_jet_jet <- ifelse(SEB_deltaphi_jet_jet > pi,
    2 * pi - SEB_deltaphi_jet_jet, SEB_deltaphi_jet_jet)
# deltar
df$SEB_deltar_tau_lep <- sqrt(df$SEB_deltaeta_tau_lep^2 +
    SEB_deltaphi_tau_lep^2)
df$SEB_deltar_tau_jet1 <- sqrt(df$SEB_deltaeta_tau_jet1^2 +
    SEB_deltaphi_tau_jet1^2)
df$SEB_deltar_tau_jet2 <- sqrt(df$SEB_deltaeta_tau_jet2^2 +
    SEB_deltaphi_tau_jet2^2)
df$SEB_deltar_lep_jet1 <- sqrt(df$SEB_deltaeta_lep_jet1^2 +
    SEB_deltaphi_lep_jet1^2)
df$SEB_deltar_lep_jet2 <- sqrt(df$SEB_deltaeta_lep_jet2^2 +
    SEB_deltaphi_lep_jet2^2)
df$SEB_deltar_jet_jet <- sqrt(df$SEB_deltaeta_jet_jet^2 +
    SEB_deltaphi_jet_jet^2)
# aux
d <- df$PRI_tau_phi - df$PRI_lep_phi
d <- 1 - 2 * ((d > pi) | ((d < 0) & (d > -pi)))
a <- sin(df$PRI_met_phi - df$PRI_lep_phi)
b <- sin(df$PRI_tau_phi - df$PRI_met_phi)
df$SEB_met_phi_centrality <- d * (a + b)/sqrt(a^2 + b^2)
# centrality
df$SEB_lep_eta_centrality <- exp(-4 * (df$PRI_lep_eta - (df$PRI_jet_leading_eta +
    df$PRI_jet_subleading_eta)/2)^2/(df$PRI_jet_leading_eta -
    df$PRI_jet_subleading_eta)^2)
df$SEB_tau_eta_centrality <- exp(-4 * (df$PRI_tau_eta - (df$PRI_jet_leading_eta +
    df$PRI_jet_subleading_eta)/2)^2/(df$PRI_jet_leading_eta -
```

```
    df$PRI_jet_subleading_eta)^2)
# pt2
df$SEB_pt2_met_tau <- ((df$PRI_met * cos(df$PRI_met_phi) +
    df$PRI_tau_pt * cos(df$PRI_tau_phi))^2 + (df$PRI_met *
    sin(df$PRI_met_phi) + df$PRI_tau_pt * sin(df$PRI_tau_phi))^2)
df$SEB_pt2_met_lep <- ((df$PRI_met * cos(df$PRI_met_phi) +
    df$PRI_lep_pt * cos(df$PRI_lep_phi))^2 + (df$PRI_met *
    sin(df$PRI_met_phi) + df$PRI_lep_pt * sin(df$PRI_lep_phi))^2)
df$SEB_pt2_met_jet1 <- ((df$PRI_met * cos(df$PRI_met_phi) +
    df$PRI_jet_leading_pt * cos(df$PRI_jet_leading_phi))^2 +
    (df$PRI_met * sin(df$PRI_met_phi) + df$PRI_jet_leading_pt *
        sin(df$PRI_jet_leading_phi))^2)
df$SEB_pt2_met_jet2 <- ((df$PRI_met * cos(df$PRI_met_phi) +
    df$PRI_jet_subleading_pt * cos(df$PRI_jet_subleading_phi))^2 +
    (df$PRI_met * sin(df$PRI_met_phi) + df$PRI_jet_subleading_pt *
        sin(df$PRI_jet_subleading_phi))^2)
df$SEB_pt2_tau_lep <- ((df$PRI_tau_pt * cos(df$PRI_tau_phi) +
    df$PRI_lep_pt * cos(df$PRI_lep_phi))^2 + (df$PRI_tau_pt *
    sin(df$PRI_tau_phi) + df$PRI_lep_pt * sin(df$PRI_lep_phi))^2)
df$SEB_pt2_tau_jet1 <- ((df$PRI_tau_pt * cos(df$PRI_tau_phi) +
    df$PRI_jet_leading_pt * cos(df$PRI_jet_leading_phi))^2 +
    (df$PRI_tau_pt * sin(df$PRI_tau_phi) + df$PRI_jet_leading_pt *
        sin(df$PRI_jet_leading_phi))^2)
df$SEB_pt2_tau_jet2 <- ((df$PRI_tau_pt * cos(df$PRI_tau_phi) +
    df$PRI_jet_subleading_pt * cos(df$PRI_jet_subleading_phi))^2 +
    (df$PRI_tau_pt * sin(df$PRI_tau_phi) + df$PRI_jet_subleading_pt *
        sin(df$PRI_jet_subleading_phi))^2)
df$SEB_pt2_lep_jet1 <- ((df$PRI_lep_pt * cos(df$PRI_lep_phi) +
    df$PRI_jet_leading_pt * cos(df$PRI_jet_leading_phi))^2 +
    (df$PRI_lep_pt * sin(df$PRI_lep_phi) + df$PRI_jet_leading_pt *
        sin(df$PRI_jet_leading_phi))^2)
df$SEB_pt2_lep_jet2 <- ((df$PRI_lep_pt * cos(df$PRI_lep_phi) +
    df$PRI_jet_subleading_pt * cos(df$PRI_jet_subleading_phi))^2 +
    (df$PRI_lep_pt * sin(df$PRI_lep_phi) + df$PRI_jet_subleading_pt *
        sin(df$PRI_jet_subleading_phi))^2)
df$SEB_pt2_jet_jet <- ((df$PRI_jet_leading_pt * cos(df$PRI_jet_leading_phi) +
    df$PRI_jet_subleading_pt * cos(df$PRI_jet_subleading_phi))^2 +
    (df$PRI_jet_leading_pt * sin(df$PRI_jet_leading_phi) +
        df$PRI_jet_subleading_pt * sin(df$PRI_jet_subleading_phi))^2)
# trans_mass WARNING some square roots below cause NAN
# errors TODO maybe want to handle the sqrt errors more
# carefully / add indicator
df$SEB_trans_mass_met_tau <- sqrt((df$PRI_met + df$PRI_tau_pt)^2 -
    df$SEB_pt2_met_tau)
df$SEB_trans_mass_met_lep <- sqrt((df$PRI_met + df$PRI_lep_pt)^2 -
    df$SEB_pt2_met_lep)
df$SEB_trans_mass_met_jet1 <- sqrt((df$PRI_met + df$PRI_jet_leading_pt)^2 -
    df$SEB_pt2_met_jet1)
df$SEB_trans_mass_met_jet2 <- sqrt((df$PRI_met + df$PRI_jet_subleading_pt)^2 -
    df$SEB_pt2_met_jet2)
df$SEB_trans_mass_tau_lep <- sqrt((df$PRI_tau_pt + df$PRI_lep_pt)^2 -
    df$SEB_pt2_tau_lep)
df$SEB_trans_mass_tau_jet1 <- sqrt((df$PRI_tau_pt + df$PRI_jet_leading_pt)^2 -
```

```
    df$SEB_pt2_tau_jet1)
df$SEB_trans_mass_tau_jet2 <- sqrt((df$PRI_tau_pt + df$PRI_jet_subleading_pt)^2 -
    df$SEB_pt2_tau_jet2)
df$SEB_trans_mass_lep_jet1 <- sqrt((df$PRI_lep_pt + df$PRI_jet_leading_pt)^2 -
    df$SEB_pt2_lep_jet1)
df$SEB_trans_mass_lep_jet2 <- sqrt((df$PRI_lep_pt + df$PRI_jet_subleading_pt)^2 -
    df$SEB_pt2_lep_jet2)
df$SEB_trans_mass_jet_jet <- sqrt((df$PRI_jet_leading_pt +
    df$PRI_jet_subleading_pt)^2 - df$SEB_pt2_jet_jet)
# p2
df$SEB_p2_tau_lep <- df$SEB_pt2_tau_lep + (df$PRI_tau_pt *
    sinh(df$PRI_tau_eta) + df$PRI_lep_pt * sinh(df$PRI_lep_eta))^2
df$SEB_p2_tau_jet1 <- df$SEB_pt2_tau_jet1 + (df$PRI_tau_pt *
    sinh(df$PRI_tau_eta) + df$PRI_jet_leading_pt * sinh(df$PRI_jet_leading_eta))^2
df$SEB_p2_tau_jet2 <- df$SEB_pt2_tau_jet2 + (df$PRI_tau_pt *
    sinh(df$PRI_tau_eta) + df$PRI_jet_subleading_pt * sinh(df$PRI_jet_subleading_eta))^2
df$SEB_p2_lep_jet1 <- df$SEB_pt2_lep_jet1 + (df$PRI_lep_pt *
    sinh(df$PRI_lep_eta) + df$PRI_jet_leading_pt * sinh(df$PRI_jet_leading_eta))^2
df$SEB_p2_lep_jet2 <- df$SEB_pt2_lep_jet2 + (df$PRI_lep_pt *
    sinh(df$PRI_lep_eta) + df$PRI_jet_subleading_pt * sinh(df$PRI_jet_subleading_eta))^2
df$SEB_p2_jet_jet <- df$SEB_pt2_jet_jet + (df$PRI_jet_leading_pt *
    sinh(df$PRI_jet_leading_eta) + df$PRI_jet_subleading_pt *
    sinh(df$PRI_jet_subleading_eta))^2
# E
df$E_tau <- df$PRI_tau_pt * cosh(df$PRI_tau_eta)
df$E_lep <- df$PRI_lep_pt * cosh(df$PRI_lep_eta)
df$E_jet1 <- df$PRI_jet_leading_pt * cosh(df$PRI_jet_leading_eta)
df$E_jet2 <- df$PRI_jet_subleading_pt * cosh(df$PRI_jet_subleading_eta)
# mass
df$SEB_mass_tau_lep <- sqrt((df$E_tau + df$E_lep)^2 - df$SEB_p2_tau_lep)
df$SEB_mass_tau_jet1 <- sqrt((df$E_tau + df$E_jet1)^2 - df$SEB_p2_tau_jet1)
df$SEB_mass_tau_jet2 <- sqrt((df$E_tau + df$E_jet2)^2 - df$SEB_p2_tau_jet2)
df$SEB_mass_lep_jet1 <- sqrt((df$E_lep + df$E_jet1)^2 - df$SEB_p2_lep_jet1)
df$SEB_mass_lep_jet2 <- sqrt((df$E_lep + df$E_jet2)^2 - df$SEB_p2_lep_jet2)
df$SEB_mass_jet_jet <- sqrt((df$E_jet1 + df$E_jet2)^2 - df$SEB_p2_jet_jet)
# aux
sum_px <- df$PRI_met * cos(df$PRI_met_phi) + df$PRI_tau_pt *
    cos(df$PRI_tau_phi) + df$PRI_lep_pt * cos(df$PRI_lep_phi)
sum_py <- df$PRI_met * sin(df$PRI_met_phi) + df$PRI_tau_pt *
    sin(df$PRI_tau_phi) + df$PRI_lep_pt * sin(df$PRI_lep_phi)
df$SEB_pt_met_tau_lep <- sqrt((sum_px)^2 + (sum_py)^2)
sum_px_2 <- sum_px + fillna(df$PRI_jet_leading_pt * cos(df$PRI_jet_leading_phi),
    0)
sum_py_2 <- sum_py + fillna(df$PRI_jet_leading_pt * sin(df$PRI_jet_leading_phi),
    0)
df$SEB_pt_met_tau_lep_jet1 <- sqrt((sum_px_2)^2 + (sum_py_2)^2)
sum_px_3 <- sum_px_2 + fillna(df$PRI_jet_subleading_pt *
    cos(df$PRI_jet_subleading_phi), 0)
sum_py_3 <- sum_py_2 + fillna(df$PRI_jet_subleading_pt *
    sin(df$PRI_jet_subleading_phi), 0)
df$SEB_pt_met_tau_lep_jet1_jet2 <- sqrt((sum_px_3)^2 + (sum_py_3)^2)
df$SEB_sum_pt_met_tau_lep <- df$PRI_met + df$PRI_tau_pt +
    df$PRI_lep_pt
```

```
    df$SEB_sum_pt_met_tau_lep_jet1 <- df$SEB_sum_pt_met_tau_lep +
        fillna(df$PRI_jet_leading_pt, 0)
    df$SEB_sum_pt_met_tau_lep_jet1_jet2 <- df$SEB_sum_pt_met_tau_lep_jet1 +
        fillna(df$PRI_jet_subleading_pt, 0)
    df$SEB_sum_pt_met_tau_lep_jet_all <- df$SEB_sum_pt_met_tau_lep_jet1 +
        df$PRI_jet_all_pt
    df$SEB_sum_pt <- df$PRI_tau_pt + df$PRI_lep_pt + df$PRI_jet_all_pt
    df$SEB_pt_ratio_lep_tau <- df$PRI_lep_pt/df$PRI_tau_pt
    df[is.na(df)] <- -999  # move back to using NA
    # we shouldn't have to add any more missingness
    # indicators because hopefullly missingness in any of
    # the new features is indicated by existing missingness
    # indicators
    df
}
```

Next we define the following function to generate a training, validation or test set with a given signal prior (`prop.signal`).

```
gen_dataset <- function(prop.signal, size = 1e+05, use_weights = F,
    seed = NULL, val = FALSE, test = FALSE) {
    if (!is.null(seed)) {
        set.seed(seed)
    }

    # Select the data
    data.to.use <- if (val) {
        if (test) {
            test.data
        } else {
            validation.data
        }
    } else {
        train.data
    }

    # Choose samples according to the desired prior
    df.signal <- data.to.use[data.to.use$Label == "s", ]
    df.bg <- data.to.use[data.to.use$Label == "b", ]
    if (use_weights) {
        # In this case we use the weights to specify the
        # sample probability of each data point
        df.signal <- df.signal[sample(nrow(df.signal), size = prop.signal *
            size, prob = df.signal$KaggleWeights, replace = F),
            ]
        df.bg <- df.bg[sample(nrow(df.bg), size = (1 - prop.signal) *
            size, prob = df.signal$KaggleWeights, replace = F),
            ]
    } else {
        # In this case samples have equal probability of
        # being added to the final dataframe
        df.signal <- df.signal[sample(nrow(df.signal), size = prop.signal *
            size, replace = F), ]
        df.bg <- df.bg[sample(nrow(df.bg), size = (1 - prop.signal) *
```

```
            size, replace = F), ]
    }

    # Must reshuffle the new dataset
    df <- rbind(df.signal, df.bg)
    df <- df[sample(nrow(df), size = nrow(df), replace = F),
        ]

    df <- df %>%
        select(-c(EventId, KaggleWeight, KaggleSet))
    return(df)
}
```

Similarly, we want to investigate changing the PU rate of misclassified positive samples, specified by a number `1-pu.pi`. To avoid the redundancy that would come with creating separate datasets for each desired `pu.pi`, we instead simply add columns to an existing dataset which correspond to the PU labels with a certain value of `pu.pi`. Note that here we also take the opportunity to convert labels from `"s"` and `"b"` for signal and background to `0` and `1`, noting that in the `pu.pi=1` case `"s"` and `0` correspond fully, as do `"b"` and `1`, whereas for other values of `pu.pi` some `"b"` samples will have PU labels of `0`.

```
add_pu_labels <- function(df) {
    # Add labels for pu.pi=0.1, 0.2, ..., 0.9
    for (pu.pi in seq(0.1, 0.9, 0.1)) {
        pu_label <- rep(1, nrow(df))
        pu_label[df$Label == "s"] <- 0  # make all signal samples unlabelled (i.e. have label 0)

        # Mislabel (1-pu.pi) proportion of the background
        # samples
        pu_label[sample(which(df$Label == "b"), (1 - pu.pi) *
            sum(df$Label == "b"), replace = F)] <- 0

        # Add the pu labels to the dataframe
        df = cbind(df, pu_label)
        colnames(df)[ncol(df)] <- paste("pu", pu.pi, sep = "")
    }

    # Do pu.pi=1 a a separate case
    df$pu1 = as.numeric(df$Label == "b")
    return(df)
}
```

We then wrap the generation of training, validation and test sets into a single function, all using the `gen_dataset(...)` function with different random seeds. From this we are able to specify each set's size and class priors.

```
gen_train_val <- function(prop.signal, size = 1e+05, size_val = 1000,
    size_test = 250000) {
    train <- gen_dataset(prop.signal, size, val = F, seed = 70)

    val <- gen_dataset(prop.signal, size = size_val, val = T,
        seed = 35)

    test <- gen_dataset(prop.signal, size = size_test, val = T,
        test = T, seed = 0)
```

```
    return(list(train = train, val = val, test = test))
}
```

Generate data set

Finally, before we move on to training our models, we use the following function to generate fully preprocessed datasets (i.e. containing the newly calculated features, the required one-hot-encoded features, and with scaled (mean 0 and standard deviation 1) numeric features). We named our datasets Steve, with training, validation and test sets written as csvs with filenames containing their specific `prop.signal` (which we allow to take values $0.1, 0.2, 0.3, 0.4, 0.5$).

```
write_steve <- function(prop.signal, size, size_val, size_test) {
    # First generate the required training, validation and
    # test sets
    dataset <- gen_train_val(prop.signal, size, size_val, size_test)
    df <- dataset$train
    val_df <- dataset$val
    test_df <- dataset$test

    # Perform the needed preprocessing on the input (X)
    # data for all three sets For each, this means removal
    # of true class labels, one-hot-encoding of certain
    # features and addition of the 70 new features.
    X.train.unscaled <- df %>%
        select(-c("Label")) %>%
        one_hot_encode %>%
        create_new_features
    X.train <- X.train.unscaled %>%
        scale_using_train_data(., .)

    X.val.unscaled <- val_df %>%
        select(-c("Label")) %>%
        one_hot_encode %>%
        create_new_features
    X.val <- X.val.unscaled %>%
        scale_using_train_data(., X.train.unscaled)

    X.test.unscaled <- test_df %>%
        select(-c("Label")) %>%
        one_hot_encode %>%
        create_new_features
    X.test <- X.test.unscaled %>%
        scale_using_train_data(., X.train.unscaled)

    df_pu = add_pu_labels(df)
    pu_cols = c()
    for (pu.pi in seq(0.1, 1, 0.1)) {
        pu_cols = c(pu_cols, paste("pu", pu.pi, sep = ""))
    }
    y.train = df_pu[, pu_cols]
    y.val = as.numeric(val_df$Label == "b")
    y.test = as.numeric(test_df$Label == "b")
    filename.end = paste(prop.signal)
```

```
    write.csv(as.matrix(X.train), paste("./aux/steve_final/X_train_",
        filename.end, ".csv", sep = ""), row.names = F)
    write.csv(as.matrix(y.train), paste("./aux/steve_final/y_train_",
        filename.end, ".csv", sep = ""), row.names = F)
    write.csv(as.matrix(X.val), paste("./aux/steve_final/X_val_",
        filename.end, ".csv", sep = ""), row.names = F)
    write.csv(as.matrix(y.val), paste("./aux/steve_final/y_val_",
        filename.end, ".csv", sep = ""), row.names = F)
    write.csv(as.matrix(X.test), paste("./aux/steve_final/X_test_",
        filename.end, ".csv", sep = ""), row.names = F)
    write.csv(as.matrix(y.test), paste("./aux/steve_final/y_test_",
        filename.end, ".csv", sep = ""), row.names = F)
}

for (prop.signal in seq(0.1, 0.5, 0.1)) {
    write_steve(prop.signal = prop.signal, size = 150000, size_val = 15000,
        size_test = 250000)
}
```

Models

Now we can present the code for our models. As mentioned in our report we use cost-sensitive support vector machines (SVMs) [1], stochastic variational Gaussian processes (SVGP) [6] and random forests [7]. In particular, we use Python libraries for all of these: LIBSVM [2] for the support vector machines; [5] for the GPs; and Scikit-learn [8] for the random forests.

Approximate Median Significance (AMS)

Here we also provide an implementation of the AMS-calculating function, the performance metric used in this dataset's original Kaggle challenge. This is an interesting measure of classifier performance, and a similar function is present in this document's accompanying R package, however, most of our analysis does not rely on us using it

The AMS is calculated (as per the Challenge document) via:

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

where s and b are the (potentially weighted) counts of signal and background samples respectively, and b_{reg} is a regularization term, with a suggested value of 10 given in the Challenge. In addition to this, the implementation below allows us to renormalise the weights of the samples in question in case we want to calculate the AMS on only a certain subset of the samples. The weights are used to scale each data point such that the whole dataset corresponds to LHC 2012 running. Hence, if a subset S' is defined, for example for testing, the weights need to be renormalized such that:

$$w'_j = w_j \frac{\sum_i w_i \mathbb{1}\{y_i = y_j\}}{\sum_{i \in s'} w_i \mathbb{1}\{y_i = y_j\}}$$

where y_i is the label of event i. It is worth pointing out that to avoid having to open the entire original data set in Python, we precompute the sum of background and signal weights from the training set, as these are needed for this renormalisation:

```
# Background sum
sum(train.data$Weight * as.numeric(train.data$Label == "b"))
```

```
## [1] 125317.9
```

```
# Signal sum
sum(train.data$Weight * as.numeric(train.data$Label == "s"))
```

```
## [1] 212.418
```

```
def calculateAMS(y_true, y_pred, w, b_reg=10):
    w_new = np.zeros(len(w))

    # 125317.9 is the sum of background (1) weights in original dataset
    w_new[y_true == 1] = w[y_true==1] * 125317.9 / sum(w[y_true==1])

     # 212.418 is the sum of signal (0) weights in original dataset
    w_new[y_true == 0] = w[y_true==0] * 212.418  / sum(w[y_true==0])

    s = sum(w_new * (1-y_true) * (1-y_pred))
    b = sum(w_new * y_true * y_pred)
    return(np.sqrt(2 * ((s + b + b_reg) * np.log(1 + s/(b + b_reg)) - s)))
```

Support Vector Machines

For our SVMs we perform cross-validation via a gridsearch to find the best values of our hyperparameters `C` (a regularisation parameter) and `gamma` (the RBF kernel coefficient—we don't use any other kernels here). This is done for each combination of `prop_signal` and `pu_pi` before the results are saved to a csv file used in analysis later.

Importantly, note that this procedure is performed twice: first using the PU per-class costs (`c1` and `cx`) suggested by du Plessis et al. [4] and once without. These per-class costs can be written as:

$$c_1 = \frac{2\pi}{\eta}$$

$$c_x = \frac{1}{1-\eta}$$

where π is our class prior (`prop.signal`) and η is the proportion of positively labelled sample compared to unlabelled samples (i.e. $\eta = \frac{n}{n+n'}$ if there are n positively labelled samples and n' unlabelled ones). We similarly evaluate the GP models both without and without these costs, however, random forests are only trained without the costs as the is no natural way to incorporate them into the classification algorithm.

```
result_w = []  # Store the results of each trained model in results_w
i = 1  # Model counter

for prop_signal in [0.1,0.2,0.3,0.4,0.5]:
    # Load the data
    X_train = pd.read_csv("aux/steve_final/X_train_%s.csv" %prop_signal)
    y_train = pd.read_csv("aux/steve_final/y_train_%s.csv" %prop_signal)
    X_val = pd.read_csv("aux/steve_final/X_val_%s.csv" %prop_signal)
    y_val = pd.read_csv("aux/steve_final/y_val_%s.csv" %prop_signal)

    # Remove the weights as these are not suitable input features
    del X_train['Weight']
    del X_val['Weight']

    for pu_pi in [0,1,2,3,4,5,6,7,8, 9]:
        # CV for the best paramters
```

```
        n = 1000
        X_train_par = X_train.iloc[:n,:]
        y_train_par = y_train.iloc[:n, pu_pi]

        eta = sum(y_train_par == 1)/len(y_train_par)

        c1 = 2*(1-prop_signal)/eta
        cx = 1/(1-eta)

        if pu_pi == 9:
            c1 =1; cx =1

        model = svm.SVC(class_weight={1:c1,0:cx}, random_state=0)

        grid = {'C': scipy.stats.expon(scale=100), 'gamma': scipy.stats.expon(scale=.1),
          'kernel': ['rbf']}

        cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=0)
        grid_search = RandomizedSearchCV(estimator=model, param_distributions=grid, n_jobs=-1,
                                         cv=cv, scoring='accuracy',error_score='raise',
                                         random_state =0)
        grid_result = grid_search.fit(X_train_par, y_train_par)

        # Fit the model

        n = 10000
        n_val = y_val.shape[0]
        X_train_fit = X_train.iloc[:n,:]; y_train_fit = y_train.iloc[:n, pu_pi];

        eta = sum(y_train_fit == 1)/len(y_train_fit)

        c1 = 2*(1-prop_signal)/eta
        cx = 1/(1-eta)
        if pu_pi == 9:
            c1 =1; cx =1

        clf =  svm.SVC( probability=True,
        class_weight={1:c1,0:cx},
        random_state=0 ).set_params(**grid_result.best_params_)
        clf.fit(X_train_fit, y_train_fit)

        train_pred = clf.predict(X_train_fit)
        val_pred = clf.predict(X_val)
        a = [f"pu pi:{(pu_pi+1)/10}", f"prop signal:{prop_signal}",
              f"train accuracy: {sum(y_train_fit == train_pred) / n}",
              f"validation accuracy: {sum(y_val.values.reshape(n_val,) == val_pred) / n_val}",
              sum(train_pred==0)/n, sum(val_pred==0)/n_val, f"c1:{c1}", f"cx:{cx}"]
        result_w.append(a)
        print(i)
        i = i +1
        print(a)

with open('results_w.csv', 'w', encoding='UTF8', newline='') as f:
```

```
  writer = csv.writer(f)

  # write multiple rows
  writer.writerows(result_w)


result_un = [] # Store the results of each trained model in results_un
i = 1  # Model counter
c1 =1; cx =1  # The unweighted cases is equivalent to setting c1 and cx equal

for prop_signal in [0.1,0.2,0.3,0.4,0.5]:
    # Load the data
    X_train = pd.read_csv("aux/steve_final/X_train_%s.csv" %prop_signal)
    y_train = pd.read_csv("aux/steve_final/y_train_%s.csv" %prop_signal)
    X_val = pd.read_csv("aux/steve_final/X_val_%s.csv" %prop_signal)
    y_val = pd.read_csv("aux/steve_final/y_val_%s.csv" %prop_signal)

    # Remove the weights as these are not suitable input features
    del X_train['Weight']
    del X_val['Weight']

    for pu_pi in [0,1,2,3,4,5,6,7,8, 9]:
        # CV for the best parameters
        n = 1000
        X_train_par = X_train.iloc[:n,:]
        y_train_par = y_train.iloc[:n, pu_pi]

        model = svm.SVC(class_weight={1:c1,0:cx}, random_state=0)

        grid = {'C': scipy.stats.expon(scale=100), 'gamma': scipy.stats.expon(scale=.1),
          'kernel': ['rbf']}

        cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3, random_state=0)
        grid_search = RandomizedSearchCV(estimator=model, param_distributions=grid, n_jobs=-1,
                                         cv=cv, scoring='accuracy',error_score='raise',
                                         random_state =0)
        grid_result = grid_search.fit(X_train_par, y_train_par)

        # Fit the model

        n = 10000
        n_val = y_val.shape[0]
        X_train_fit = X_train.iloc[:n,:]; y_train_fit = y_train.iloc[:n, pu_pi];

        clf =  svm.SVC( probability=True,
        class_weight={1:c1,0:cx},
        random_state=0).set_params(**grid_result.best_params_)
        clf.fit(X_train_fit, y_train_fit)

        train_pred = clf.predict(X_train_fit)
        val_pred = clf.predict(X_val)
        a = [f"pu pi:{(pu_pi+1)/10}", f"prop signal:{prop_signal}",
              f"train accuracy: {sum(y_train_fit == train_pred) / n}",
```

```
            f"validation accuracy: {sum(y_val.values.reshape(n_val,) == val_pred) / n_val}",
            sum(train_pred==0)/n, sum(val_pred==0)/n_val]
        result_un.append(a)
        print(i)
        i = i +1
        print(a)

with open('results_un.csv', 'w', encoding='UTF8', newline='') as f:
  writer = csv.writer(f)

  # write multiple rows
  writer.writerows(result_un)
```

Gaussian Processes

```
class SEBVariationalELBO(VariationalELBO):
    def _log_likelihood_term(self, variational_dist_f, target, **kwargs):
        if 'seb_weights' in kwargs:
            weights = kwargs['seb_weights']
            return (weights * self.likelihood.expected_log_prob(target,
          variational_dist_f, **kwargs)).sum(-1)
        else:
            return super()._log_likelihood_term(variational_dist_f, target, **kwargs)

class GPModel(ApproximateGP):
    def __init__(self, inducing_points):
        variational_distribution = CholeskyVariationalDistribution(inducing_points.size(0))
        variational_strategy = VariationalStrategy(self, inducing_points, variational_distribution,
        learn_inducing_locations=True)
        super(GPModel, self).__init__(variational_strategy)
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

Model fitting

```
torch.manual_seed(0) # for reproducibility
batch_size = 256
inducing_size = 32
prop_signals = [0.1, 0.2, 0.3, 0.4, 0.5]
pu_pis = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
thresh = 10

results_csv = "results_final.csv"
if not os.path.isfile(f"results/gpc-weighted/{results_csv}"):
    with open(f"results/gpc-weighted/{results_csv}", "w") as f:
        f.write("prop_signal,pu_pi,n_estimator,bootstrap,criterion,max_features,min_samples_leaf,pu_cla
```

```
for prop_signal in prop_signals:
    # Load training, validation and (potentially) test data
    X_train = pd.read_csv(f"./aux/steve_final/X_train_{prop_signal}.csv")
    y_train_all = pd.read_csv(f"./aux/steve_final/y_train_{prop_signal}.csv")

    w_train = X_train["Weight"]
    X_train = X_train.drop("Weight", axis=1)

    X_val = pd.read_csv(f"./aux/steve_final/X_val_{prop_signal}.csv")
    y_val = pd.read_csv(f"./aux/steve_final/y_val_{prop_signal}.csv").iloc[:,0]

    w_val = X_val["Weight"]
    X_val = X_val.drop("Weight", axis=1)
    n_val = X_val.shape[0]

    X_test = pd.read_csv(f"./aux/steve_final/X_test_{prop_signal}.csv")
    y_test = pd.read_csv(f"./aux/steve_final/y_test_{prop_signal}.csv").iloc[:,0]

    w_test = X_test["Weight"]
    X_test = X_test.drop("Weight", axis=1)
    n_test = X_test.shape[0]

    X_train[X_train <= -thresh] = -thresh
    X_train[X_train >= thresh] = thresh

    X_val[X_val <= -thresh] = -thresh
    X_val[X_val >= thresh] = thresh

    X_test[X_test <= -thresh] = -thresh
    X_test[X_test >= thresh] = thresh

    # Remove feature names from Xs
    X_train = torch.Tensor(np.array(X_train))
    X_val = torch.Tensor(np.array(X_val))
    X_test = torch.Tensor(np.array(X_test))

    n = X_train.shape[0]

    for pu_pi in pu_pis:
        y_train = y_train_all[f"pu{pu_pi}"]

        # Set class weights (as in SM1 PU paper)
        c1 = 1; cx = 1
        if pu_pi != 1:
            eta = sum(y_train == 1)/len(y_train)
            c1 = 2*pu_pi/eta
            cx = 1/(1-eta)

        # Cycle through all possible random forest parameter combinations

        for pu_class_weights in [True, False]:

            start = time.time()
```

```
y_train = torch.Tensor(np.array(y_train));
y_val = torch.Tensor(np.array(y_val));
y_test = torch.Tensor(np.array(y_test))
train_dataset = TensorDataset(X_train[inducing_size:,:], y_train[inducing_size:])
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dataset = TensorDataset(X_test, y_test)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

inducing_points = X_train[:inducing_size, :]
model = GPModel(inducing_points=inducing_points)
likelihood = gpytorch.likelihoods.BernoulliLikelihood()

num_epochs = 10
optimizer = torch.optim.Adam([
    {'params': model.parameters()},
    {'params': likelihood.parameters()},
], lr=1e-1)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=0.75)

mll = SEBVariationalELBO(likelihood, model, num_data=y_train.size(0))
val_seb_weights = torch.Tensor([c1 if y == 1.0 else cx for y in y_test])
for j in range(num_epochs):
    for (i, (x_batch, y_batch)) in enumerate(train_loader):
        model.train()
        optimizer.zero_grad()
        if pu_class_weights:
            batch_seb_weights = torch.Tensor([c1 if y == 1.0 else cx for y in y_batch])
            loss = -mll(model(x_batch), y_batch, seb_weights=batch_seb_weights)
        else:
            loss = -mll(model(x_batch), y_batch)
        loss.backward()
        optimizer.step()
        if i % 100 == 0:
            print('Iter %d/%d (%d/%d)- Loss: %.3f' % (j + 1,
            num_epochs,
            i,
            len(train_loader),
            loss.item()))
    model.eval()
    if pu_class_weights:
        val_loss = -mll(model(X_test), y_test, seb_weights=val_seb_weights)
    else:
        val_loss = -mll(model(X_test), y_test)
    print('Epoch %d/%d - Val Loss: %.3f' % (j + 1, num_epochs, val_loss.item()))
    scheduler.step()
end = time.time()

test_pred = (likelihood(model(X_test)).mean > 0.5).detach().numpy().astype(int)
test_acc = np.mean((test_pred == y_test.detach().numpy()).astype(int))
train_pred = (likelihood(model(X_train)).mean > 0.5).detach().numpy().astype(int)
train_acc = np.mean((train_pred == y_train.detach().numpy()).astype(int))

# How much of the time was it predicting signal?
```

```
            train_pred_signal = sum(train_pred==0)/n
            test_pred_signal = (sum(y_test == 0) / n_test).item()

            # Calculate AMS
            train_ams = calculateAMS(y_train_all[f"pu1"], train_pred, w_train)
            test_ams = calculateAMS(y_test.detach().numpy(), test_pred, w_test)

            with open(f"results/gpc-weighted/{results_csv}", "a") as f:
                f.write(",".join([str(x) for x in
                    [prop_signal, pu_pi, "", "", "", "",
                    "", pu_class_weights, n, train_acc, "", test_acc,
                    train_pred_signal, "", test_pred_signal, train_ams, "", test_ams, end-start]]) + "\
```

Random Forests

The random forest evaluation code is largely similar to the GP code but with the actual model implementation requiring less work since we are just using the Scikit-learn module out of the box. First we evaluate the random forests with a variety of hyperparameters on our validation sets. To avoid excessive computation times we only perform these tests on `prop_signal`s of $0.1, 0.3$, and 0.5, with `pu_pi` only taking values in $\{0.2, 0.6, 1\}$. This is run three times and results are averaged to account for the randomness inherent in the models.

```
# First make sure the steve_rf_gridsearch folder exists - if not, make it
try:
    os.mkdir("src/python code/steve_rf_gridsearch")
except OSError as e:
    pass

results_csv = "results_cv.csv"

# Check if a csv already exists for results - if not, make one w/ headers
if not os.path.isfile(f"src/python code/steve_rf_gridsearch/{results_csv}"):
    with open(f"src/python code/steve_rf_gridsearch/{results_csv}", "w") as f:
        f.write("prop_signal,pu_pi,n_estimator,bootstrap,criterion,max_features,min_samples_leaf,pu_cla

# Dataset and random forest parameters to try out
n = 10000

prop_signals = [0.1, 0.3, 0.5]
pu_pis = [0.2, 0.6, 1]

n_estimator_arr = [500, 1000]
bootstrap_arr = [True, False]
criterion_arr = ["entropy", "log_loss", "gini"]
max_features_arr = ["log2", "sqrt"]
min_samples_leaf_arr = [2, 5, 10]

# We don't use the PU class weights/costs (from du Plessis et al. PU paper), but SVM and GP models do.
pu_class_weights = False

# Calculate the number of tests per full-iteration
num_param_combs = np.prod([len(x) for x in [n_estimator_arr, bootstrap_arr, criterion_arr,
                                            max_features_arr,
                                            min_samples_leaf_arr]])
```

```
num_tests = num_param_combs * len(prop_signals) * len(pu_pis)

num_iter = 3  # number of times to train each forest - later we will take an
              # average of these results once they are all stored in a csv
for iter in range(num_iter):
    i = 0
    for prop_signal in prop_signals:
        # Load training, validation data
        X_train = pd.read_csv(f"./aux/steve_final/X_train_{prop_signal}.csv")
        y_train_all = pd.read_csv(f"./aux/steve_final/y_train_{prop_signal}.csv")

        w_train = X_train["Weight"]
        X_train = X_train.drop("Weight", axis=1)

        X_val = pd.read_csv(f"./aux/steve_final/X_val_{prop_signal}.csv")
        y_val = pd.read_csv(f"./aux/steve_final/y_val_{prop_signal}.csv").iloc[:,0]

        w_val = X_val["Weight"]
        X_val = X_val.drop("Weight", axis=1)
        n_val = X_val.shape[0]

        # Only train on n examples (chosen at random)
        if n is not None:
            trainRows = np.random.choice(range(X_train.shape[0]), size=n, replace=False)
            X_train = X_train.iloc[trainRows,:]
            y_train_all = y_train_all.iloc[trainRows,:]
            w_train = w_train[trainRows]

        # Remove feature names from Xs
        X_train = np.array(X_train)
        X_val = np.array(X_val)

        n = X_train.shape[0]

        for pu_pi in pu_pis:
            # Select the training labels for this particular pu_pi
            y_train = y_train_all[f"pu{pu_pi}"]

            # Cycle through all possible random forest parameter combinations
            for (n_estimator,
                 bootstrap,
                 criterion,
                 max_features,
                 min_samples_leaf) in itertools.product(n_estimator_arr, bootstrap_arr, criterion_arr, m

                # Specify the random forest classifier parameters
                params={"n_estimators": n_estimator,
                        "bootstrap": bootstrap,
                        "criterion": criterion,
                        "max_features": max_features,
                        "min_samples_leaf": min_samples_leaf,
                        "n_jobs":-1}
```

```
                # Create the classifier
                pu_estimator = RandomForestClassifier(**params)

                # Train it
                start = time.time()
                pu_estimator.fit(X_train, y_train)#, y_train*prop_signal+(1-y_train))
                end = time.time()

                # Get predictions
                train_pred = pu_estimator.predict(X_train)
                val_pred = pu_estimator.predict(X_val)

                # Get accuracies
                train_acc = sum(y_train == train_pred) / n
                val_acc = sum(y_val == val_pred) / n_val

                # How much of the time was it predicting signal?
                train_pred_signal = sum(train_pred==0)/n
                val_pred_signal = sum(y_val == 0) / n_val

                # Calculate AMS
                train_ams = calculateAMS(y_train_all[f"pu1"], train_pred, w_train)
                val_ams = calculateAMS(y_val, val_pred, w_val)

                i += 1
                with open(f"src/python code/steve_rf_gridsearch/{results_csv}", "a") as f:
                    f.write(",".join([str(x) for x in
                        [prop_signal, pu_pi, n_estimator, bootstrap, criterion, max_features,
                        min_samples_leaf, pu_class_weights, n, train_acc, val_acc,
                        train_pred_signal, val_pred_signal, train_ams, val_ams,
                        end-start]]) + "\n")
                    print(f"Iteration {iter+1}/{num_iter} Test {i}/{num_tests} done (w/ prop_signal={pr
```

Averaging out results

```
def averageOutRFResults(inputCSV, outputCSV, prop_signals, pu_pis, numTest, numIter):
  df = pd.read_csv(inputCSV)

  for i in range(numTest):
      # Get the rows corresponding to tests with identical parameters
      row = df.iloc[[i + j*numTest for j in range(numIter)],:]

      for col in ["train_acc", "val_acc", "train_pred_signal", "val_pred_signal",
                  "train_ams", "val_ams", "time"]:
          # Replace the results of the first instance of this parameter combination
          # by the mean over all instances (of which there should be numIter)
          df.loc[i, col] = sum(row[col])/numIter

  # Truncate the dataframe to only include the mean results
  df = df[:numTest]

  # Write out the new, averaged dataframe
  df.to_csv(outputCSV, index=False)
```

```
averageOutRFResults("src/python code/steve_rf_gridsearch/results_cv.csv",
                    "src/python code/results_final_fulltrain.csv",
                    [0.1, 0.2, 0.3, 0.4, 0.5],
                    [0.2, 0.6, 1],
                    1080, 3)
```

Since we found that random forest performed better than SVMs and GPs on the validation set, we also ran it on the test set. This was done with the overall-best performing parameters from the gridsearch: 1000 trees per forest, each of which trained on 10,000 examples without bootstrap aggregation and using Gini impurity to decide on splits based on a random subset of $\log_2 d$ features (out of the full d features), plus the requirement that leaf nodes must contain at least 10 training examples. (For completeness, these models were also reran on the validation set.)

```
results_csv = "results_final_fulltrain.csv"

# Check if a csv already exists for results - if not, make one w/ headers
if not os.path.isfile(f"src/python code/steve_rf_gridsearch/{results_csv}"):
    with open(f"src/python code/steve_rf_gridsearch/{results_csv}", "w") as f:
        f.write("prop_signal,pu_pi,n_estimator,bootstrap,criterion,max_features,min_samples_leaf,pu_cla

# Dataset and random forest parameters
n = 10000  # train on only 10000 examples for speed
prop_signals = [0.1, 0.2, 0.3, 0.4, 0.5]
pu_pis = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]

    # Best parameter values found through CV/gridsearch (see results.csv)
    n_estimator_arr = [1000]
    bootstrap_arr = [False]
    criterion_arr = ["gini"]
    max_features_arr = ["log2"]
    min_samples_leaf_arr = [10]

# Calculate the number of tests per full-iteration
num_tests = len(prop_signals) * len(pu_pis)

num_iter = 5  # number of times to train each forest - later we will take an
              # average of these results once they are all stored in a csv
for iter in range(num_iter):
    i = 0
    for prop_signal in prop_signals:
        # Load training, validation and test data
        X_train = pd.read_csv(f"./aux/steve_final/X_train_{prop_signal}.csv")
        y_train_all = pd.read_csv(f"./aux/steve_final/y_train_{prop_signal}.csv")

        w_train = X_train["Weight"]
        X_train = X_train.drop("Weight", axis=1)

        X_val = pd.read_csv(f"./aux/steve_final/X_val_{prop_signal}.csv")
        y_val = pd.read_csv(f"./aux/steve_final/y_val_{prop_signal}.csv").iloc[:,0]

        w_val = X_val["Weight"]
        X_val = X_val.drop("Weight", axis=1)
        n_val = X_val.shape[0]
```

```
X_test = pd.read_csv(f"./aux/steve_final/X_test_{prop_signal}.csv")
y_test = pd.read_csv(f"./aux/steve_final/y_test_{prop_signal}.csv").iloc[:,0]

w_test = X_test["Weight"]
X_test = X_test.drop("Weight", axis=1)
n_test = X_test.shape[0]

# Only train on n examples (chosen at random)
if n is not None:
    trainRows = np.random.choice(range(X_train.shape[0]), size=n, replace=False)
    X_train = X_train.iloc[trainRows,:]
    y_train_all = y_train_all.iloc[trainRows,:]
    w_train = w_train[trainRows]

# Remove feature names from Xs
X_train = np.array(X_train)
X_val = np.array(X_val)
X_test = np.array(X_test)

n = X_train.shape[0]

for pu_pi in pu_pis:
    # Select the training labels for this particular pu_pi
    y_train = y_train_all[f"pu{pu_pi}"]

    # Specify the random forest classifier parameters
    params={"n_estimators": 1000,
            "bootstrap": False,
            "criterion": "gini",
            "max_features": "log2",
            "min_samples_leaf": 10,
            "n_jobs":-1}

    # Create the classifier
    pu_estimator = RandomForestClassifier(**params)

    # Train it
    start = time.time()
    pu_estimator.fit(X_train, y_train)#, y_train*prop_signal+(1-y_train))
    end = time.time()

    # Get predictions
    train_pred = pu_estimator.predict(X_train)
    val_pred = pu_estimator.predict(X_val)

    # Get accuracies
    train_acc = sum(y_train == train_pred) / n
    val_acc = sum(y_val == val_pred) / n_val

    # How much of the time was it predicting signal?
    train_pred_signal = sum(train_pred==0)/n
    val_pred_signal = sum(y_val == 0) / n_val
```

```
            # Calculate AMS
            train_ams = calculateAMS(y_train_all[f"pu1"], train_pred, w_train)
            val_ams = calculateAMS(y_val, val_pred, w_val)

            test_pred = pu_estimator.predict(X_test)
            test_acc = sum(y_test == test_pred) / n_test
            test_pred_signal = sum(y_test == 0) / n_test
            test_ams = calculateAMS(y_test, test_pred, w_test)

            i += 1
            with open(f"src/python code/steve_rf_gridsearch/{results_csv}", "a") as f:
                elif mode == "test":
                    f.write(",".join([str(x) for x in
                        [prop_signal, pu_pi, n_estimator, bootstrap, criterion, max_features,
                        min_samples_leaf, pu_class_weights, n, train_acc, val_acc, test_acc,
                        train_pred_signal, val_pred_signal, test_pred_signal, train_ams, val_ams, test_a

                    print(f"Iteration {iter+1}/{num_iter} Test {i}/{num_tests} done (w/ prop_signal={pro
```

And finally we must also average out the 5 runs we've just performed on the test set (note we put the output csv in the `results` folder with the results from the other two models).

```
averageOutRFResults("src/python code/steve_rf_gridsearch/results_final_fulltrain.csv",
                    "results/random_forest/random_forest_results_final.csv",
                    [0.1, 0.2, 0.3, 0.4, 0.5],
                    [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1],
                    95, 5)
```

Results

Define some helper functions for plotting

```
fix_results_df <- function(df) {
    df$pu_class_weights <- ifelse(df$pu_class_weights == "True",
        TRUE, FALSE)
    df %>%
        arrange(prop_signal, pu_pi)
}

contour_excess_test_acc <- function(df, title = "") {
    contour_breaks <- seq(0, 0.4, by = 0.05)
    ggplot(df, aes(prop_signal, pu_pi, z = test_acc - (1 - prop_signal))) +
        geom_contour_filled(breaks = contour_breaks) + scale_fill_viridis_d(drop = F) +
        xlim(c(0.1, 0.5)) + ylim(c(0.1, 1)) + ggtitle(title)
}

contour_excess_val_acc <- function(df, title = "") {
    contour_breaks <- seq(0, 0.4, by = 0.05)
    ggplot(df, aes(prop_signal, pu_pi, z = val_acc - (1 - prop_signal))) +
        geom_contour_filled(breaks = contour_breaks) + scale_fill_viridis_d(drop = F) +
        xlim(c(0.1, 0.5)) + ylim(c(0.1, 1)) + ggtitle(title)
}

contour_test_ams <- function(df) {
```

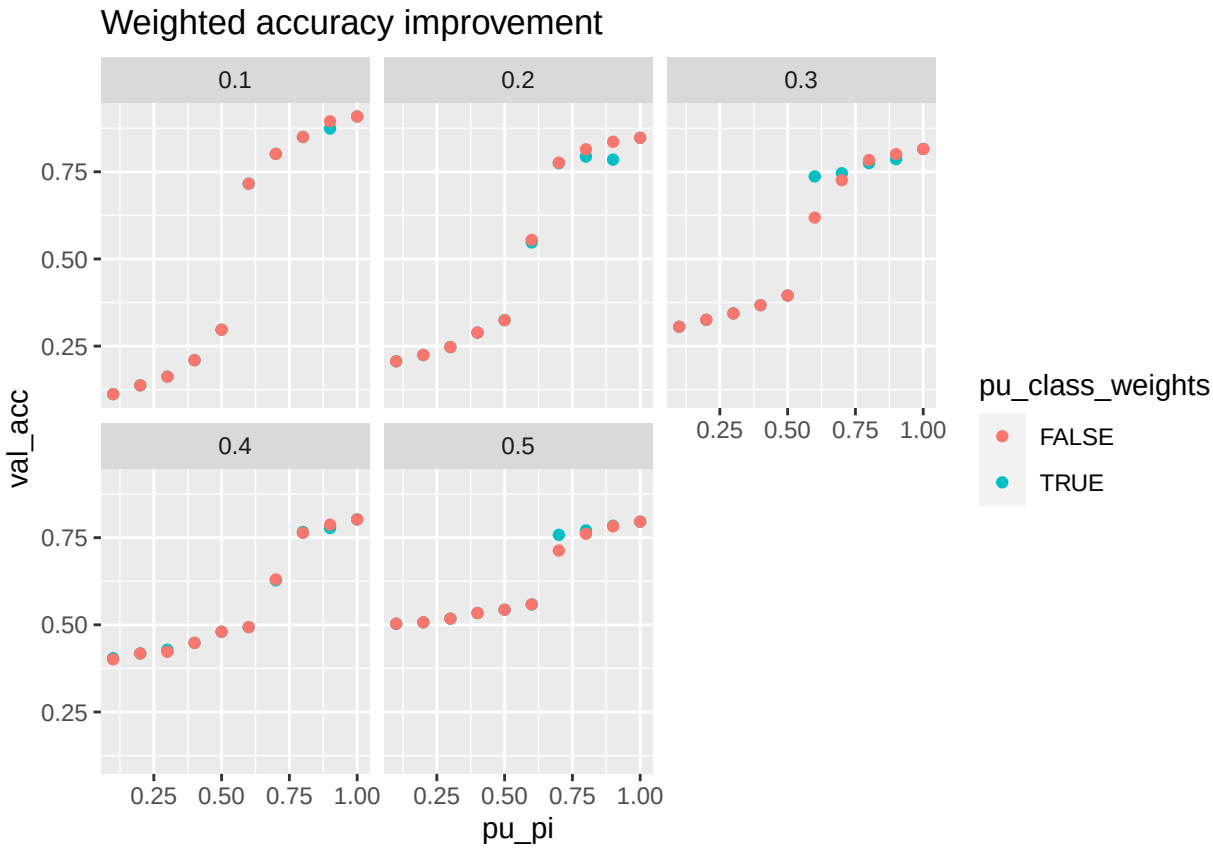
```
    ggplot(df, aes(prop_signal, pu_pi, z = test_ams)) + geom_contour_filled() +
        ggtitle("Test AMS")
}

contour_train_ams <- function(df) {
    ggplot(df, aes(prop_signal, pu_pi, z = train_ams)) + geom_contour_filled() +
        ggtitle("Train AMS")
}

contour_weighted_vs_unweighted <- function(df) {
    contour_breaks <- seq(0, 0.5, by = 0.05)
    weighted_acc <- df[df$pu_class_weights, ]$test_acc
    unweighted_acc <- df[!df$pu_class_weight, ]$test_acc
    stopifnot(length(weighted_acc) == length(unweighted_acc))
    # assume we sorted these figures by pu_pi and
    # prop_signal... otherwise this subtraction makes no
    # sense
    df$weighted_minus_unweighted <- weighted_acc - unweighted_acc
    ggplot(df, aes(prop_signal, pu_pi, z = weighted_minus_unweighted)) +
        geom_contour_filled(breaks = contour_breaks) + scale_fill_viridis_d(drop = F) +
        xlim(c(0.1, 0.5)) + ylim(c(0.1, 1)) + ggtitle("Weighted accuracy improvement")
}

contour_weighted_vs_unweighted_validation <- function(df) {
    contour_breaks <- seq(0, 0.5, by = 0.05)
    weighted_acc <- df[df$pu_class_weights, ]$val_acc
    unweighted_acc <- df[!df$pu_class_weight, ]$val_acc
    stopifnot(length(weighted_acc) == length(unweighted_acc))
    # assume we sorted these figures by pu_pi and
    # prop_signal... otherwise this subtraction makes no
    # sense
    df$weighted_minus_unweighted <- weighted_acc - unweighted_acc
    ggplot(df, aes(prop_signal, pu_pi, z = weighted_minus_unweighted)) +
        geom_contour_filled(breaks = contour_breaks) + scale_fill_viridis_d(drop = F) +
        xlim(c(0.1, 0.5)) + ylim(c(0.1, 1)) + ggtitle("Weighted accuracy improvement")
}
```

```
gpc.results <- read.csv("./results/gpc-weighted/results_final.csv") %>%
    fix_results_df
rf.results <- read.csv("./results/random_forest/random_forest_results_final.csv") %>%
    fix_results_df
svm.results <- read.csv("./results/SVM_results.csv")
svm.results$pu_class_weights <- c(rep(T, 50), c(rep(F, 50)))
# impute pu_pi 1 results for pu_class_weights=T make sure
# to sort again!
rf.tmp <- rf.results[rf.results$pu_pi == 1 & !rf.results$pu_class_weights,
    ]
rf.tmp$pu_class_weights <- T
rf.results <- rbind(rf.results, rf.tmp) %>%
    arrange(prop_signal, pu_pi)
```

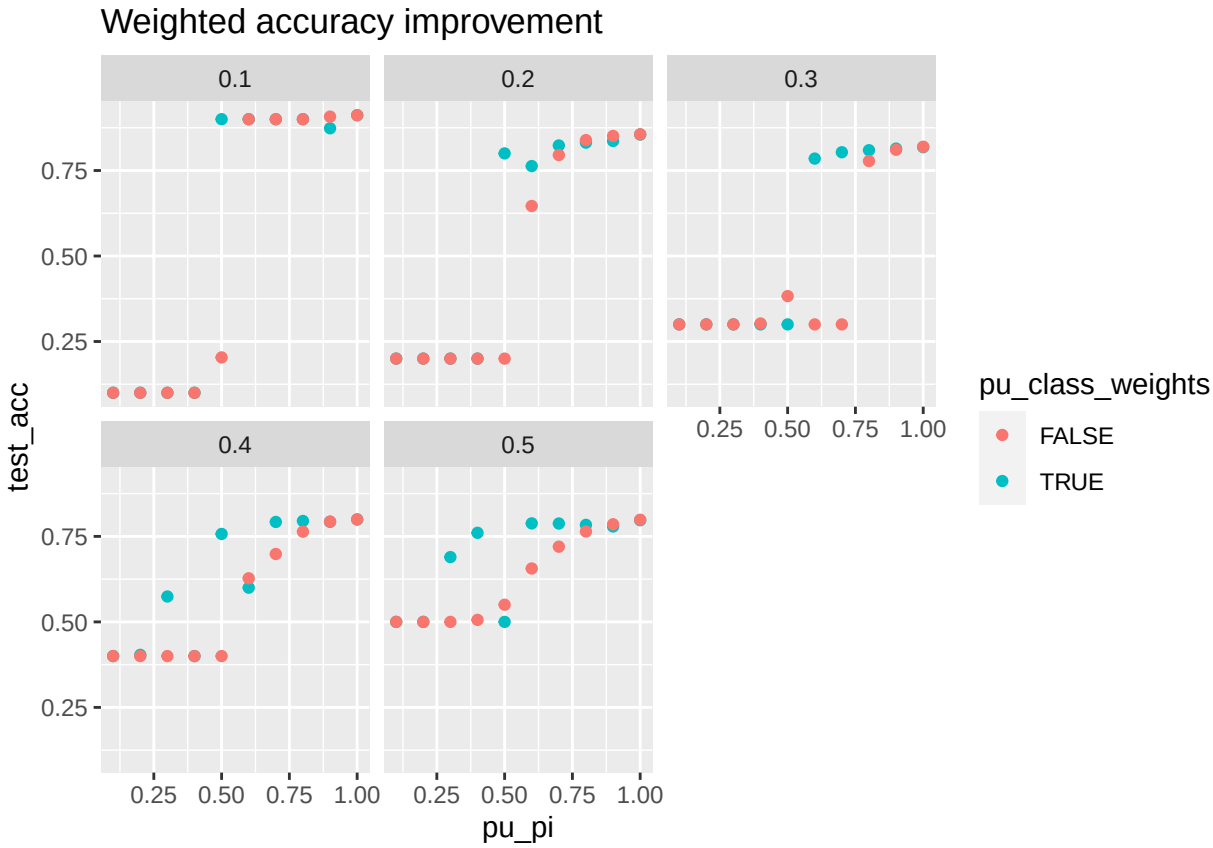
Weighted vs. Unweighted classifier

We check whether each of our classifiers actually perform better when adding weights

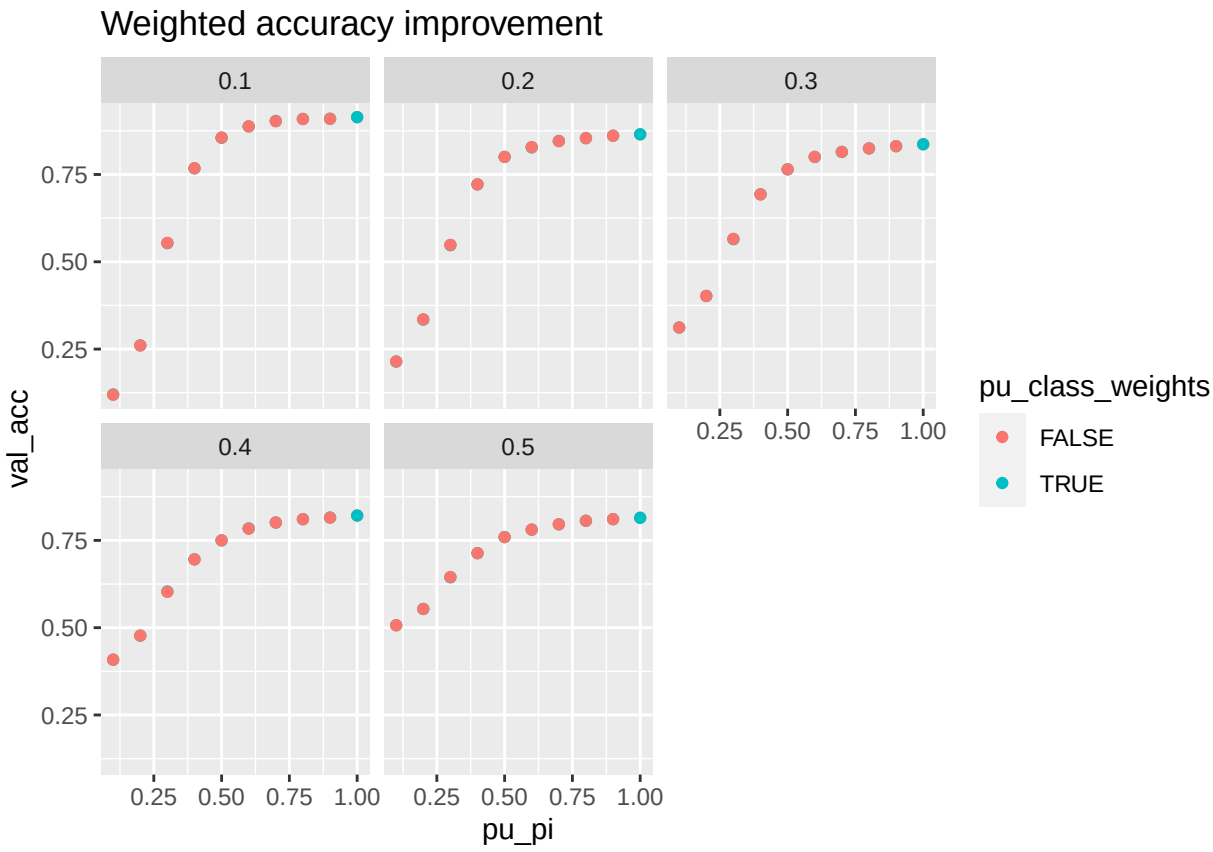
```
w_vs_unw_pu_pi_val <- function(df, prop.signal) {
    ggplot(df, aes(x = pu_pi, y = val_acc, colour = pu_class_weights)) +
        geom_point() + facet_wrap(vars(prop_signal)) + ggtitle("Weighted accuracy improvement")
}

w_vs_unw_pu_pi_test <- function(df, prop.signal) {
    ggplot(df, aes(x = pu_pi, y = test_acc, colour = pu_class_weights)) +
        geom_point() + facet_wrap(vars(prop_signal)) + ggtitle("Weighted accuracy improvement")
}

w_vs_unw_pu_pi_val(svm.results)
```




```
w_vs_unw_pu_pi_test(gpc.results)
```



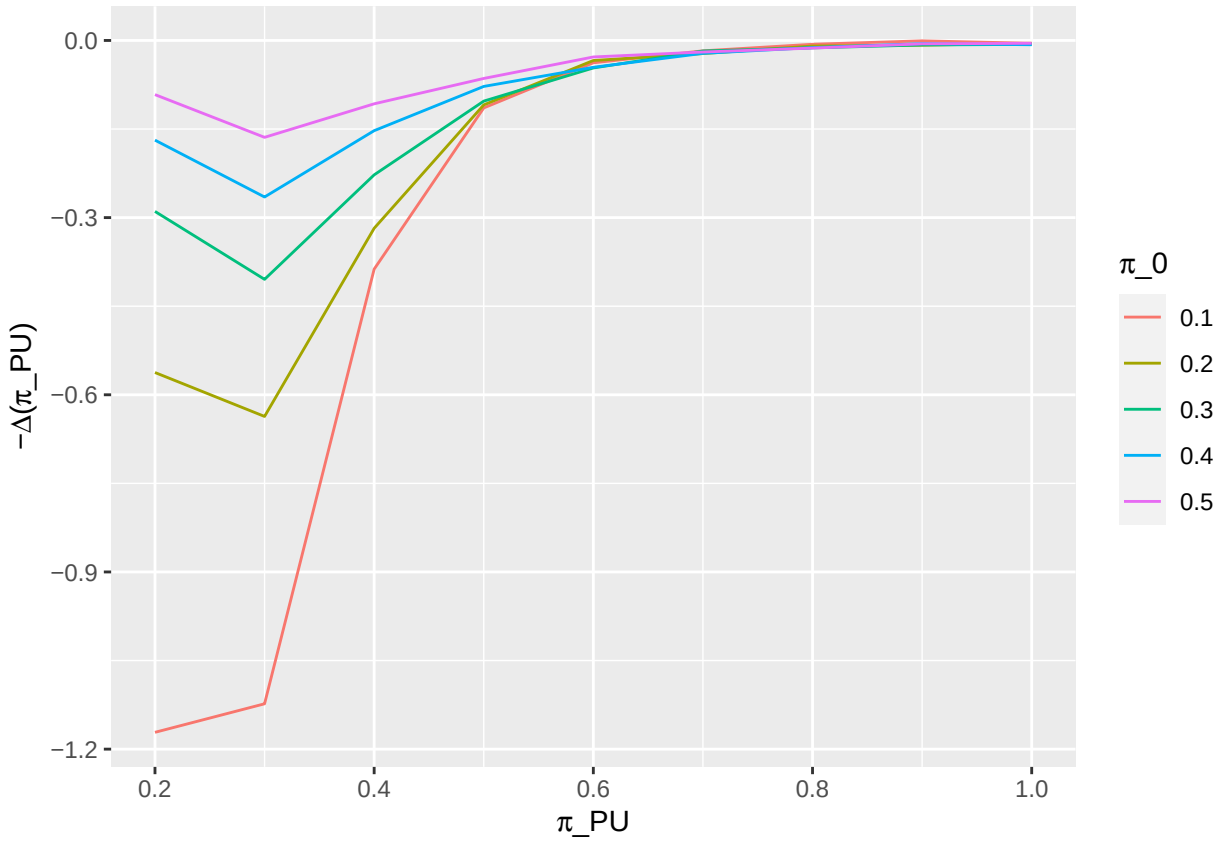

```
w_vs_unw_pu_pi_val(rf.results)
```



Effect of $\pi_{\mathbf{PU}}$

Calculate relative change of the validation accuracy:

```
rf.results.t <- rf.results %>%
    filter(pu_class_weights) %>%
    group_by(prop_signal) %>%
    mutate(pu_pi_val_acc_rel_change = -(val_acc - lag(val_acc))/lag(val_acc)) %>%
    ungroup

tmp.df <- rf.results.t %>%
    filter(pu_class_weights) %>%
    filter(pu_pi > 0.1) %>%
    select("prop_signal", "pu_pi", "pu_pi_val_acc_rel_change") %>%
    mutate(prop_signal = as.character(prop_signal))
ggplot(tmp.df, aes(x = pu_pi, y = pu_pi_val_acc_rel_change, group = prop_signal,
    color = prop_signal)) + geom_line() + ylab(TeX("-\\Delta(\\pi_PU)")) +
    xlab(TeX("\\pi_PU")) + labs(color = TeX("\\pi_0"), fill = TeX("\\pi_0"))
```

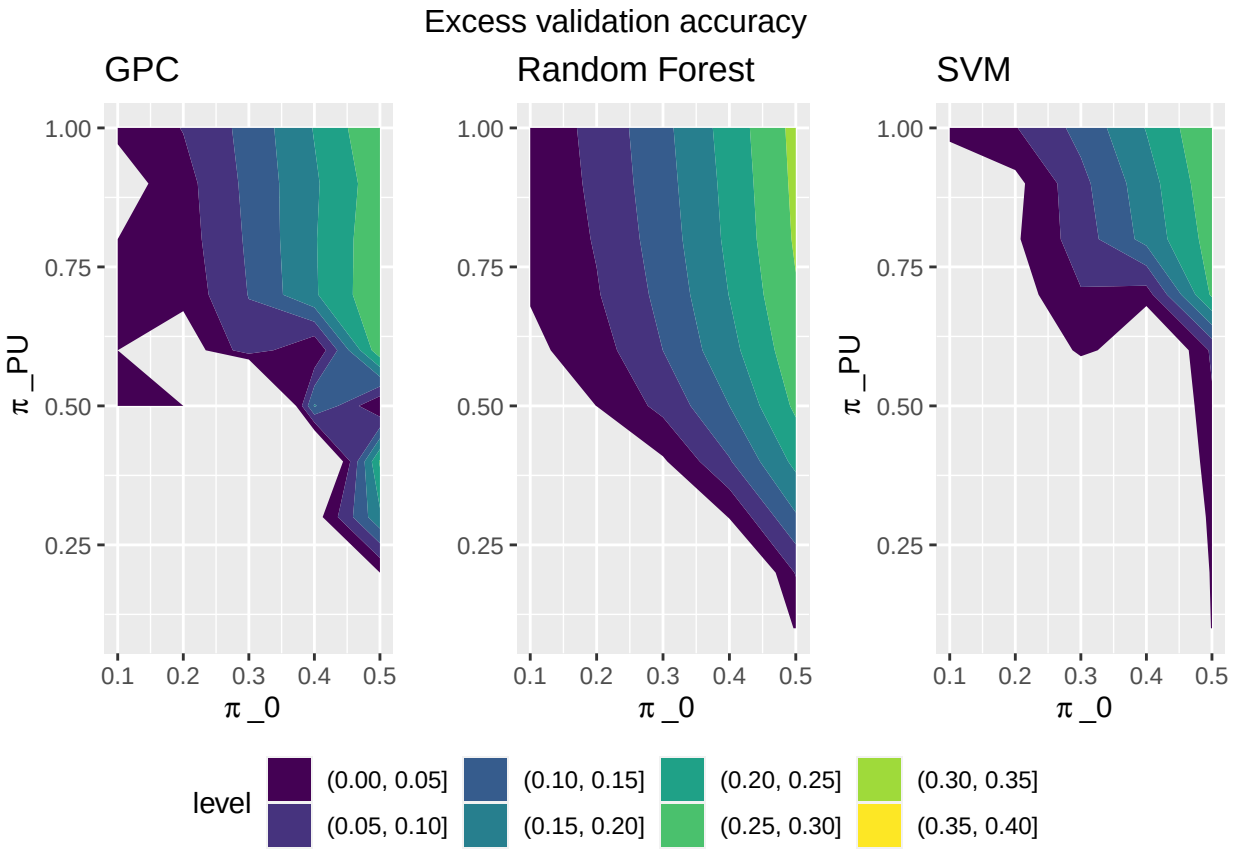



```
ggsave("./results/img/pu_pi_val_acc_rel_change.pdf")
```

```
## Saving 6.5 x 4.5 in image
```

We can comapre how the different classes of models perform (in terms of excess accuracy) over different regions of the parameter space:

```
gpc.excess <- gpc.results %>%
    filter(pu_class_weights == T) %>%
    contour_excess_val_acc(title = "GPC") + xlab(expression(~pi ~
    "_0")) + ylab(expression(~pi ~ "_PU"))
rf.excess <- rf.results %>%
    filter(pu_class_weights == T) %>%
    contour_excess_val_acc(title = "Random Forest") + ylab("") +
    xlab(expression(~pi ~ "_0"))
svm.excess <- svm.results %>%
    filter(pu_class_weights == T) %>%
    contour_excess_val_acc(title = "SVM") + xlab(expression(~pi ~
    "_0")) + ylab(expression(~pi ~ "_PU"))

excess.plt <- ggarrange(gpc.excess, rf.excess, svm.excess, common.legend = T,
    legend = "bottom", nrow = 1) %>%
    annotate_figure(top = text_grob("Excess validation accuracy"))
excess.plt
```

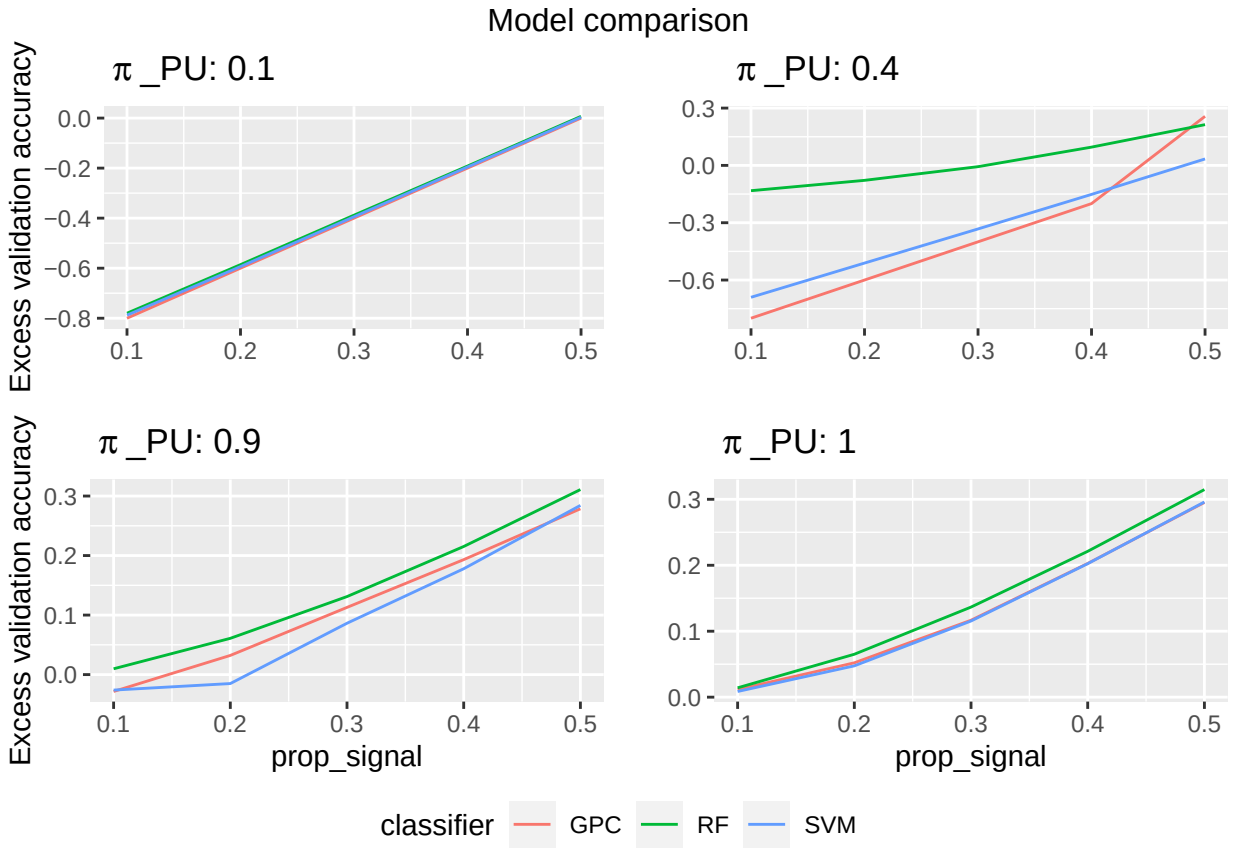


Compare models on excess accuracy

We can compare the models with each other. RF is easily the winner!

```
all_results <- rbind.fill(svm.results, gpc.results, rf.results)
all_results$classifier <- factor(c(rep("SVM", 100), rep("GPC",
    100), rep("RF", 100)))
compare_models_signal <- function(weight = TRUE, pu.pi = 0.5) {
    ggplot(data = all_results %>%
        filter(pu_class_weights == weight) %>%
        filter(pu_pi == pu.pi), aes(x = prop_signal, y = val_acc -
        (1 - prop_signal))) + geom_line(aes(colour = classifier)) +
        ylab("Excess validation accuracy")
}

pu_0.1.plt <- compare_models_signal(pu.pi = 0.1) + xlab("") +
    ggtitle(expression(paste(~pi ~ "_PU", ": ", 0.1)))
pu_0.4.plt <- compare_models_signal(pu.pi = 0.4) + ylab("") +
    xlab("") + ggtitle(expression(paste(~pi ~ "_PU", ": ", 0.4)))
pu_0.9.plt <- compare_models_signal(pu.pi = 0.9) + ggtitle(expression(paste(~pi ~
    "_PU", ": ", 0.9)))
pu_1.plt <- compare_models_signal(pu.pi = 1) + ylab("") + ggtitle(expression(paste(~pi ~
    "_PU", ": ", 1)))

compare.plt <- ggarrange(pu_0.1.plt, pu_0.4.plt, pu_0.9.plt,
    pu_1.plt, common.legend = T, legend = "bottom") %>%
    annotate_figure(top = text_grob("Model comparison"))
```

```
compare.plt
```

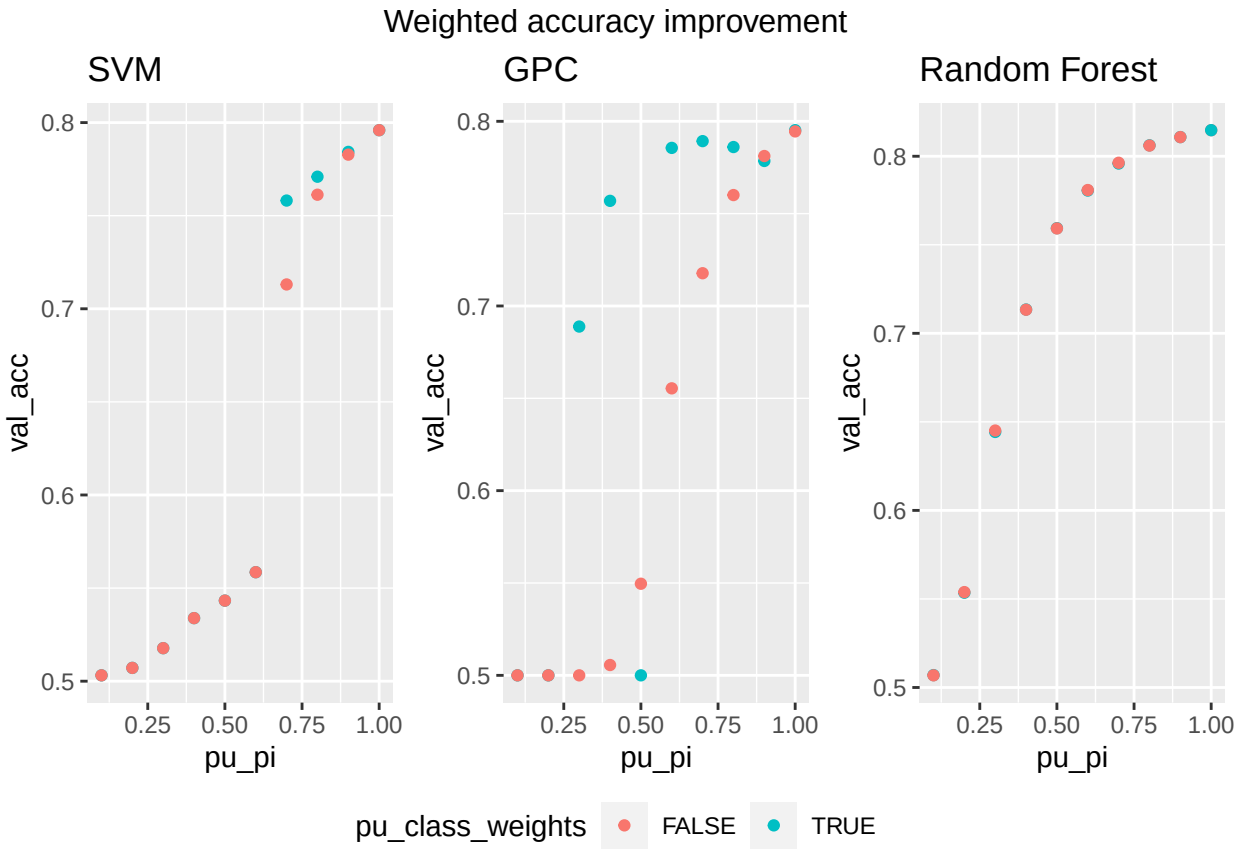



```
ggsave("./results/img/model_comparison.pdf", plot = compare.plt)
```

```
## Saving 6.5 x 4.5 in image
```

Finally we compare how the models peform with and without the PU weights. It is clear that Random Forests seem to be the same with/without the weights, whereas GPC benefits the most.

```
w_vs_unw_pu_pi_val_signal <- function(df, ps, title = "") {
    ggplot(df %>%
        filter(prop_signal == ps), aes(x = pu_pi, y = val_acc,
        colour = pu_class_weights)) + geom_point() + ggtitle(title)
}

svm.weigth <- w_vs_unw_pu_pi_val_signal(svm.results, 0.5, "SVM")
gpc.weigth <- w_vs_unw_pu_pi_val_signal(gpc.results, 0.5, "GPC")
rf.weigth <- w_vs_unw_pu_pi_val_signal(rf.results, 0.5, "Random Forest")

weigth.plt <- ggarrange(svm.weigth, gpc.weigth, rf.weigth, common.legend = T,
    legend = "bottom", nrow = 1) %>%
    annotate_figure(top = text_grob("Weighted accuracy improvement"))
weigth.plt
```




```
ggsave("./results/img/weigth_comparison.pdf", plot = weigth.plt)
```

```
## Saving 6.5 x 4.5 in image
```

References

[1] Francis R. Bach, David Heckerman, and Eric Horvitz. Considering cost asymmetry in learning classifiers. *Journal of Machine Learning Research*, 7(63):1713–1741, 2006.

[2] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3), may 2011.

[3] ATLAS collaboration. Dataset from the atlas higgs boson machine learning challenge 2014, Jan 1970.

[4] Marthinus C du Plessis, Gang Niu, and Masashi Sugiyama. Analysis of learning from positive and unlabeled data. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[5] Jacob R. Gardner, Geoff Pleiss, David Bindel, Kilian Q. Weinberger, and Andrew Gordon Wilson. Gpytorch: Blackbox matrix-matrix gaussian process inference with GPU acceleration. *CoRR*, abs/1809.11165, 2018.

[6] James Hensman, Alexander Matthews, and Zoubin Ghahramani. Scalable variational gaussian process classification. In *Artificial Intelligence and Statistics*, pages 351–360. PMLR, 2015.

[7] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1, 1995.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] Tim Salimans. HiggsML. https://github.com/TimSalimans/HiggsML/, 2014.