HarvardX 125.9x Project: Higgs dataset classification

Yin-Chi Chan

2022-09-27

Contents

Preface

This template generates output in both HTML and PDF form. Code chunks are typeset in Fira Code, with code ligatures enabled.

0.1 R setup

```
# R 4.1 key features: new pipe operator, \(x) as shortcut for function(x)
# R 4.0 key features: stringsAsFactors = FALSE by default, raw character strings r"()"
if (packageVersion('base') < '4.1.0') {
  stop('This code requires R ⩾ 4.1.0!')
}

if(!require("pacman")) install.packages("pacman")
if(!require("keras")) {
```

```
  install.packages("keras")
  keras::install_keras()
}

library(pacman)
p_load(data.table, dtplyr, tidyverse, R.utils, Rfast, knitr,
       keras, caret, pROC, knitr, conflicted)
conflict_prefer('summarize', 'dplyr')
conflict_prefer('summarise', 'dplyr')
conflict_prefer('filter', 'dplyr')
conflict_prefer('between', 'dplyr')
conflict_prefer('auc', 'pROC')

# Somehow, this seems to prevent memory leaks
tensorflow::tf$compat$v1$disable_eager_execution()
```

1 Introduction

This report partially fufills the requirements for the HarvardX course PH125.9x: "Data Science: Capstone". The objective of this project is to apply machine learning techniques beyond standard linear regression to a publicly available dataset of choice.

1.1 The HIGGS dataset

The HIGGS dataset is a synthetic dataset simulating particle accelerator data (Baldi, Sadowski, and Whiteson 2014). Although the details of the simulated particle collisions are beyond the scope of this project, We summarize the contents of the dataset as follows.

The HIGGS dataset is available from the UCI Machine Learning Repository. The following code loads the dataset, downloading and unzipping the CSV file as necessary:

```
options(timeout=1800) # Give more time for the download to complete
if(!dir.exists('data_raw')) {
  dir.create('data_raw')
}
if(!file.exists('data_raw/HIGGS.csv')) {
  download.file(
    'https://archive.ics.uci.edu/ml/machine-learning-databases/00280/HIGGS.csv.gz',
    'data_raw/HIGGS.csv.gz')
  gunzip('data_raw/HIGGS.csv.gz')
}
```

```
# Load dataset into memory
higgs_all ← fread('data_raw/HIGGS.csv')

# Assign column names (csv contains no headers)
colnames(higgs_all) ← c('signal',
                     'lepton_pT', 'lepton_eta', 'lepton_phi',
                     'missing_E_mag', 'missing_E_phi',
                     'jet1_pT', 'jet1_eta', 'jet1_phi', 'jet1_btag',
                     'jet2_pT', 'jet2_eta', 'jet2_phi', 'jet2_btag',
                     'jet3_pT', 'jet3_eta', 'jet3_phi', 'jet3_btag',
                     'jet4_pT', 'jet4_eta', 'jet4_phi', 'jet4_btag',
                     'm_jj', 'm_jjj', 'm_lv', 'm_jlv',
                     'm_bb', 'm_wbb', 'm_wwbb')

# Separate input and output columns
xAll ← higgs_all ▷ select(-signal) ▷ as.data.table()
```

```
yAll ← higgs_all ▷ select(signal) ▷ as.data.table()
rm(higgs_all)
```

The HIGGS dataset contains 28 features and one binary target, `signal`. The `signal` value of an observation corresponds to whether a particle collision produced a Higgs boson as an intermediate product. Two possible processes are considered with the same input particles, one of which generates the Higgs boson (the "signal" process) and one which does not (the "background" process).

Of the 28 features in the dataset, the last seven features, prefixed 'm_', are called "high-level" features and are based on computing the mass of expected intermediate decay products in the signal and background processes, assuming that the observed final decay products were generated by each process. The 21 "low-level" features consist of momentum data for a lepton and four jets, *b*-tags for each jet marking the likelihood that the jet is associated with a bottom quark, and partial information of "missing" total momentum caused by undetected decay products such as neutrinos. Due to the nature of the simulated particle colliders and detectors, full directional data of this missing momentum is not available.

1.2 Creating the final test splits

Since `Keras`, one of the libraries we will use, uses "validation" internally when fitting a model, we will use the term "**final** validation set" to denote the final hold-out set for post-model evaluation.

```
# Create 10% test set split
set.seed(1)
idx ← createDataPartition(yAll$signal, p = 0.1, list = F)
gc()

x ← xAll[-idx,]
y ← yAll[-idx,]
xFinalTest ← x[idx,]
yFinalTest ← y[idx,]
rm(xAll, yAll)
gc()
```

1.3 Optimization metric

We will optimize the area under the receiver operating curve (AUC) of our models. The AUC is defined such that a perfect classifer has an AUC of 1 and a random classifier has an AUC of 0.5. An advantage of using the AUC is that it reflects the trade-off between the true and false positive rates of a model depending on the chosen classification threshold. We will use the `pROC` package to create and plot ROC curves, and to compute their area.

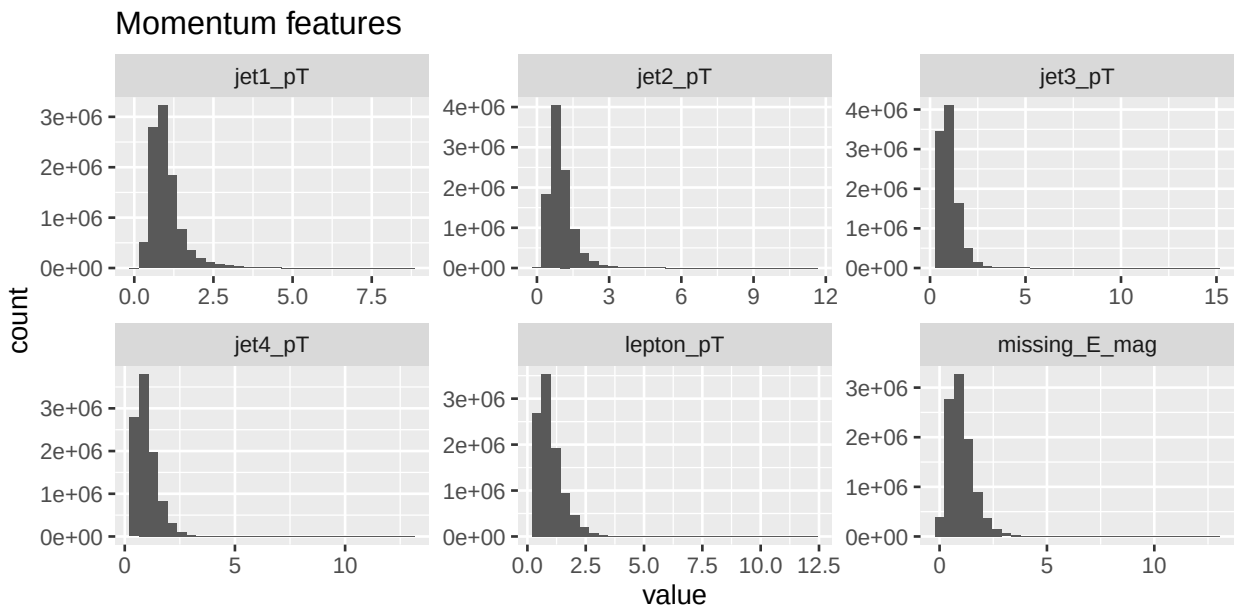
2 Data analysis and transformation

In this section, we analyze the various input features of the HIGGS dataset and apply deskewing and z-score normalization.

2.1 Low-level momentum features

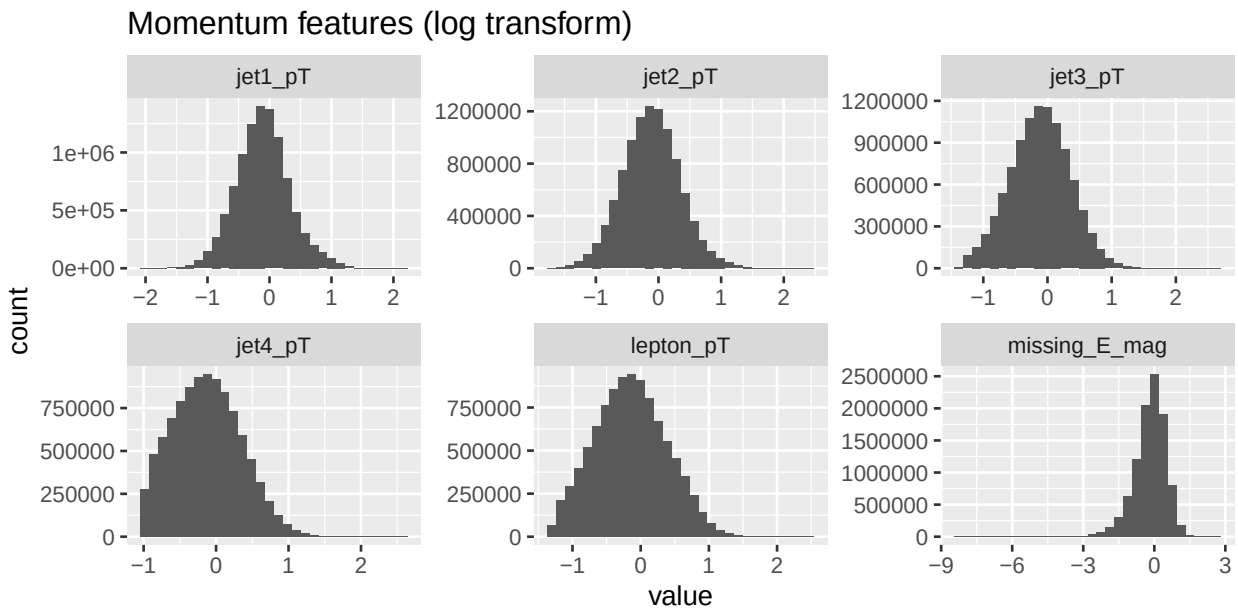
The input features containing `_pT` are related to transverse (perpendicular to the input beams) momentum, as is the high-level feature `missing_E_mag`. Histograms of these features are plotted as follows:

```
x ▷
  select(c(contains('_pT'), 'missing_E_mag')) ▷
  as.data.frame() ▷
  gather() ▷
  ggplot(aes(value)) +
  geom_histogram() +
  facet_wrap(~key, scales = "free", nrow = 2) +
  ggtitle('Momentum features')
```



The momentum features are all quite skewed. To deskew the data, we apply a log transformation:

```
x ▷
  select(c(contains('_pT'), 'missing_E_mag')) ▷
  as.data.frame() ▷
  log() ▷
  gather() ▷
  ggplot(aes(value)) +
  geom_histogram() +
  facet_wrap(~key, scales = "free", nrow = 2) +
  ggtitle('Momentum features (log transform)')
```



The skewness of the data before and after log transformation is as follows:

```
tibble(
  Feature = x ▷
```

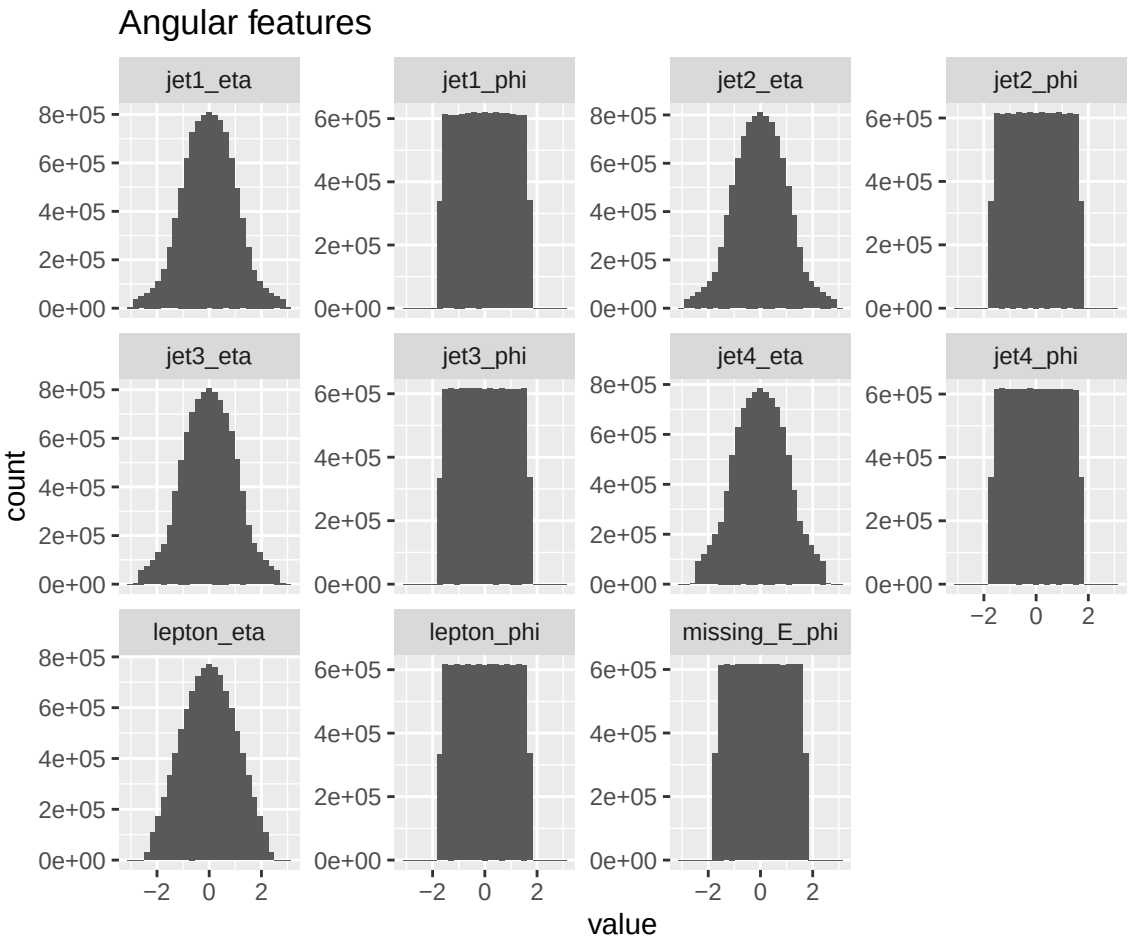
```
    select(c(contains('_pT'), 'missing_E_mag')) ▷
    as.data.table() ▷
    colnames(),
  Skewness = x ▷
    select(c(contains('_pT'), 'missing_E_mag')) ▷
    as.data.table() ▷
    as.matrix() ▷
    colskewness(),
  'Skewness (log)' = x ▷
    select(c(contains('_pT'), 'missing_E_mag')) ▷
    as.data.table() ▷
    log() ▷
    as.matrix() ▷
    colskewness()
) ▷
  kable(align = 'lrr', booktabs = T, linesep = '')
```

Feature	Skewness	Skewness (log)
lepton_pT	1.759399	0.1494493
jet1_pT	1.902766	0.1615350
jet2_pT	1.966270	0.0772273
jet3_pT	1.706359	0.0125490
jet4_pT	1.726289	0.2643133
missing_E_mag	1.487827	-0.8840618

2.2 Angular features

The features containing `_eta` or `_phi` describe angular data of the detected products of the simulated collision processes. Histograms for these features are as follows:

```
x ▷
  select(c(contains('_eta'), contains('_phi'))) ▷
  as.data.frame() ▷
  gather() ▷
  ggplot(aes(value)) +
  geom_histogram() +
  facet_wrap(~key, scales = "free_y", nrow = 3) +
  xlim(-pi,pi) +
  ggtitle('Angular features')
```

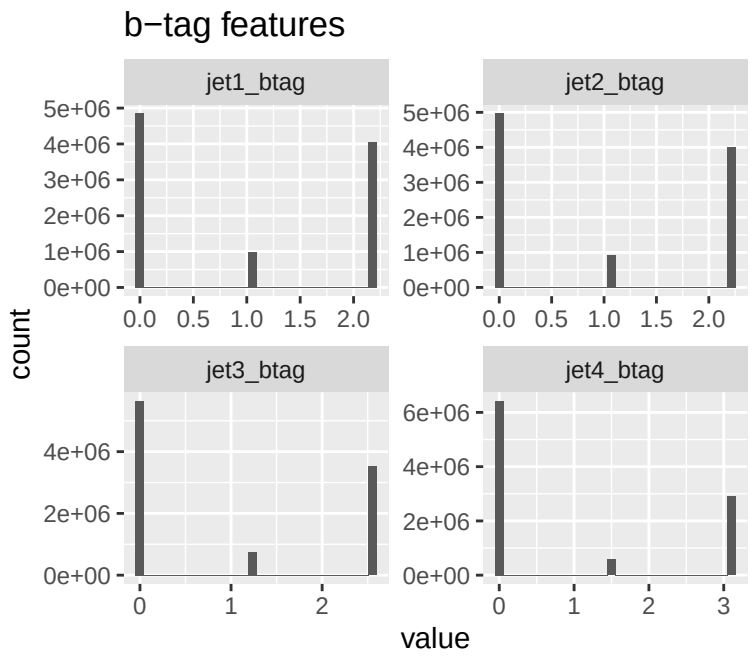


The histograms above do not show significant skew; therefore, deskewing will not be applied to these input features.

2.3 *b*-tag features

Each *b*-tag feature contains three possible values:

```
x ▷
  select(contains('_btag')) ▷
  as.data.frame() ▷
  gather() ▷
  ggplot(aes(value)) +
  geom_histogram() +
  facet_wrap(~key, scales = "free", nrow = 2) +
  ggtitle('b-tag features')
```



Interestingly, the *b*-tag data is encoded to have unit mean, but not unit standard deviation:

```
# b-tag means
tibble(
Mean = x ▷
    select(contains('_btag')) ▷
    as.data.table() ▷
    as.matrix() ▷
    colMeans(),
'Standard Deviation' = x ▷
    select(contains('_btag')) ▷
    as.data.table() ▷
    as.matrix() ▷
    colVars(std=T)
) ▷
  rownames_to_column('Feature') ▷
  kable(align = 'lrr', booktabs = T, linesep = '')
```

Feature	Mean	Standard Deviation
1	1.0000549	1.027791
2	1.0000157	1.049421
3	1.0000600	1.193689
4	0.9998079	1.400151

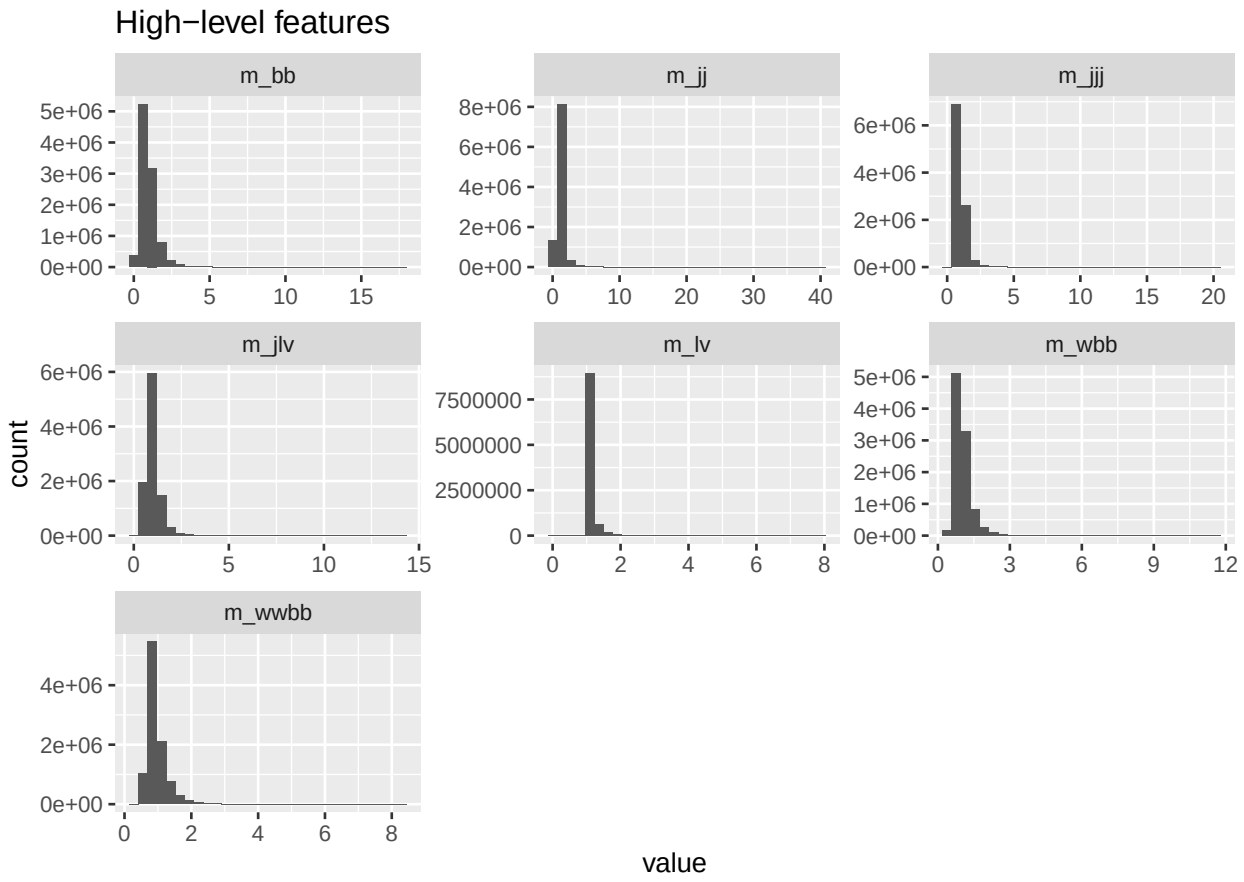
Nevertheless, we will leave these features alone.

2.4 High-level features

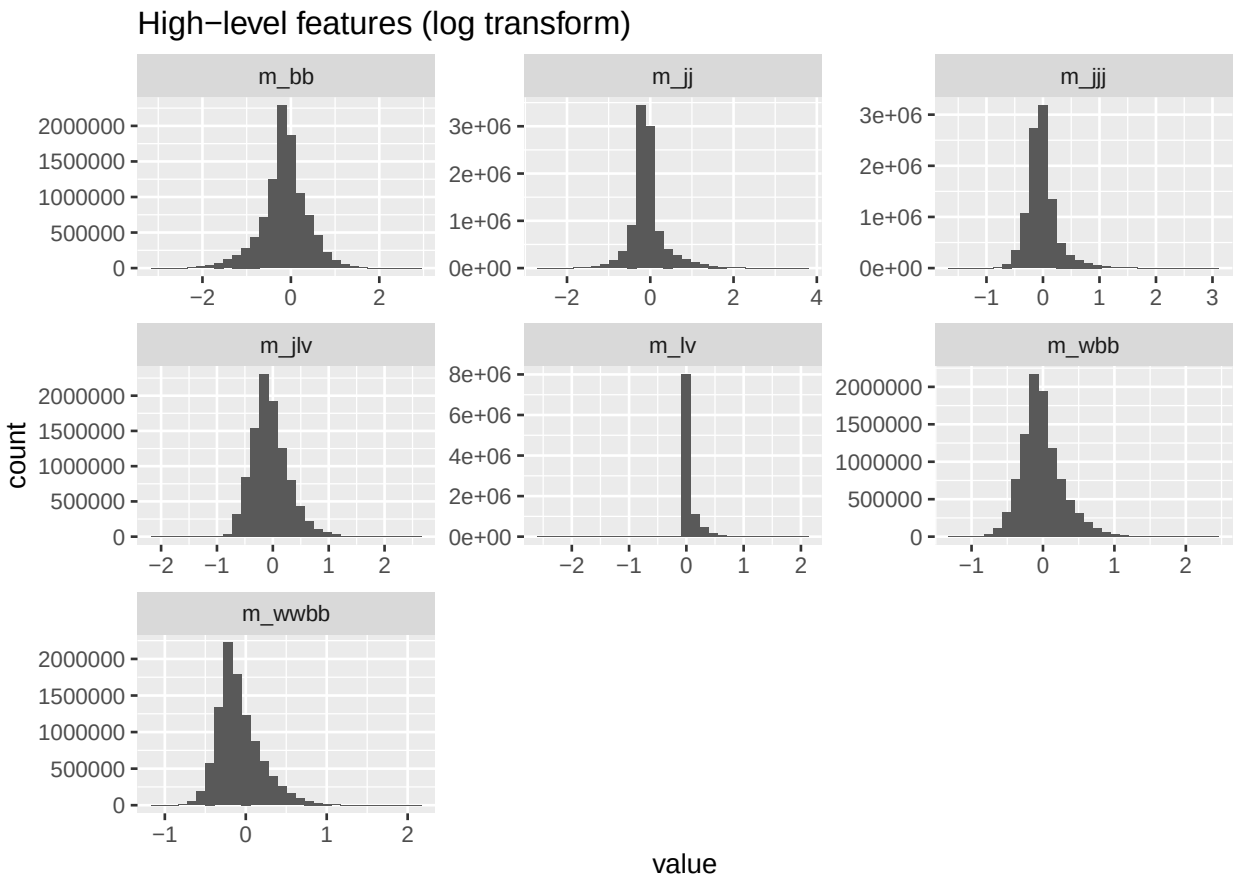
The high-level features for the HIGGS dataset are related to tranverse momentum and, as with the low-level momentum features, are quite skewed:

```
x ▷
  select(contains('m_')) ▷
  as.data.frame() ▷
  gather() ▷
  ggplot(aes(value)) +
```

```
  geom_histogram() +
  facet_wrap(~key, scales = "free", nrow = 3) +
  ggtitle('High-level features')
```




```
x ▷
  select(contains('m_')) ▷
  as.data.frame() ▷
  log() ▷
  gather() ▷
  ggplot(aes(value)) +
  geom_histogram() +
  facet_wrap(~key, scales = "free", nrow = 3) +
  ggtitle('High-level features (log transform)')
```



The skewness of the high-level features, before and after log transformation, are as follows:

```
tibble(
  Feature = x ▷
    select(contains('m_')) ▷
    as.data.table() ▷
    colnames(),
  Skewness = x ▷
    select(contains('m_')) ▷
    as.data.table() ▷
    as.matrix() ▷
    colskewness(),
  'Skewness (log)' = x ▷
    select(contains('m_')) ▷
    as.data.table() ▷
    log() ▷
    as.matrix() ▷
    colskewness()
) ▷
  kable(align = 'lrr', booktabs = T, linesep = '')
```

Feature	Skewness	Skewness (log)
m_jj	6.529976	1.1279300
m_jjj	5.007659	1.4937149
m_lv	4.614399	2.9117405
m_jlv	2.850652	0.7598249
m_bb	2.424460	-0.3608753
m_wbb	2.687129	0.8387949
m_wwbb	2.548881	1.0247968

2.5 Final data transformation

The following code applies log transformation to selected columns of the HIGGS training and test datasets. We will also convert our data types into matrices here for input into subsequent stages of our analysis:

```
# Apply log transform
x ← x ▷
  mutate(across(
    c(contains('m_'), contains('_pT'), contains('_mag')),
    log)) ▷
  as.data.table()

# Convert to matrices
x ← x %>% as.matrix()

# Apply log transform
xFinalTest ← xFinalTest ▷
  mutate(across(
    c(contains('m_'), contains('_pT'), contains('_mag')),
    log)) ▷
  as.data.table()

# Convert to matrices
x ← x %>% as.matrix()
xFinalTest ← xFinalTest %>% as.matrix()

gc()
```

The following code then scales the training data to have zero mean and unit standard deviation:

```
m ← colMeans(x)
sd ← colVars(x, std = T) # std=T → compute st. dev. instead of variance

x ← scale(x, center = m, scale = sd)
xFinalTest ← scale(xFinalTest, center = m, scale = sd)
```

Finally, we extract the target values to a vector:

```
y ← y$signal
yFinalTest ← yFinalTest$signal
```

3 GPU-accelerated machine learning using Keras

We will use the `keras` package for machine learning. The `keras` package allows us to model the data using neural networks (NN); in this case, we will use a straightforward Multilayer Perceptron (MLP) with an Rectified Linear Unit (ReLU) activation function on all hidden nodes and a logistic activation function on the output node, thus giving a final estimate between 0 and 1 of the probability that a given observation corresponds to the "signal" process. The ReLu activation function, $\text{relu}(x) = \max(0, x)$, is chosen for its simplicity and computational efficiency.

Note that in theory, an NN with sufficient breadth can approximate any function, even with just a single hidden layer (Asadi and Jiang 2020). However, we want a model that can learn the *general* characteristics of the signal and background processes from which our data is obtained. Therefore, to avoid overfitting, we withhold 20% of the training data in each epoch for validation. Additionally, 50% dropout is applied to each hidden layer, meaning that only half of the hidden nodes are used in each training batch. This has also been shown to reduce overfitting effectively (Srivastava et al. 2014).

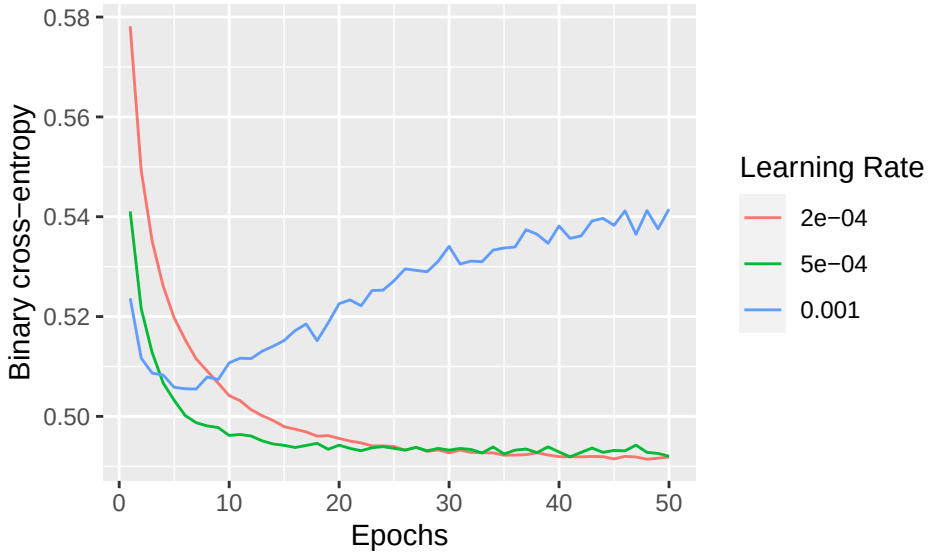
3.1 Tuning the learning rate

The following code plots the binary cross-entropy (a loss metric for binary classification networks) after each training epoch for three different learning rates, using the Adam(Kingma and Ba 2014) optimizer. We will train networks with 3 hidden layers of 256 nodes each.

```
train_keras_history ← function(x, y, depth, breadth,
                        dropout = 0.5, learning_rate=0.0002, epochs = 50) {
  model ← keras_model_sequential()
  model ▷
    layer_dense(breadth, 'relu', input_shape = ncol(x)) ▷
    layer_dropout(rate = dropout)

  # subsequent hidden layers
  if (depth > 1) {
    for (layer in seq(2,depth)) {
      model ▷ layer_dense(breadth, 'relu') ▷ layer_dropout(rate = dropout)
    }
  }

  # output layer (logistic activation function for binary classification)
  model ▷ layer_dense(1, 'sigmoid')

  # compile model
  model ▷
    keras::compile(
      loss = 'binary_crossentropy',
      optimizer = optimizer_adam(learning_rate = learning_rate),
      metrics = metric_auc()
    )

  # a larger batch size trains faster but uses more GPU memory
  history ← model ▷ fit(x, y, epochs = epochs, batch_size = 8192, validation_split = 0.2)

  rm(model)
  gc()
  k_clear_session()
  tensorflow::tf$compat$v1$reset_default_graph()

  history
}
```

```
tensorflow::set_random_seed(42, disable_gpu = F)
if (!file.exists('cache/nn_results.RDdata')) {
  history1 ← train_keras_history(x, y, 3, 256, learning_rate = 1e-3)
  history2 ← train_keras_history(x, y, 3, 256, learning_rate = 5e-4)
  history3 ← train_keras_history(x, y, 3, 256, learning_rate = 2e-4)
  save(history1, history2, history3, file = 'cache/nn_results.RDdata')
}
load('cache/nn_results.RDdata')
tensorflow::set_random_seed(42, disable_gpu = F)
```

```
tibble(
  epoch = seq(50),
  `1e-3` = history1$metrics$val_loss,
  `5e-4` = history2$metrics$val_loss,
  `2e-4` = history3$metrics$val_loss) ▷
  pivot_longer(-epoch,'lr',values_to = 'loss') ▷
  mutate(lr = as.numeric(lr)) ▷
  ggplot(aes(epoch,loss, color=as.factor(lr))) +
  geom_line() +
  xlab('Epochs') +
  ylab('Binary cross-entropy') +
  labs(color='Learning Rate')
```



The results show that a smaller learning rate provides smoother loss as a function of the number of training epochs, but takes longer to train. Nevertheless, even the lowest learning rate shown above results in reasonably fast convergence. On the other hand, a learning rate that is too large may lead to non-convergence of the loss function. Based on the above results, we will use a training rate of 2×10^{-4} for all subsequent modelling.

3.2 Effect of MLP depth and breadth

The following function trains an MLP with the specified number of hidden layers (`depth`) and nodes per hidden layer (`breadth`), and returns the model and training history. We will use the Adam (Kingma and Ba 2014) optimizer with a learning rate of 2×10^{-4} and 50 epochs.

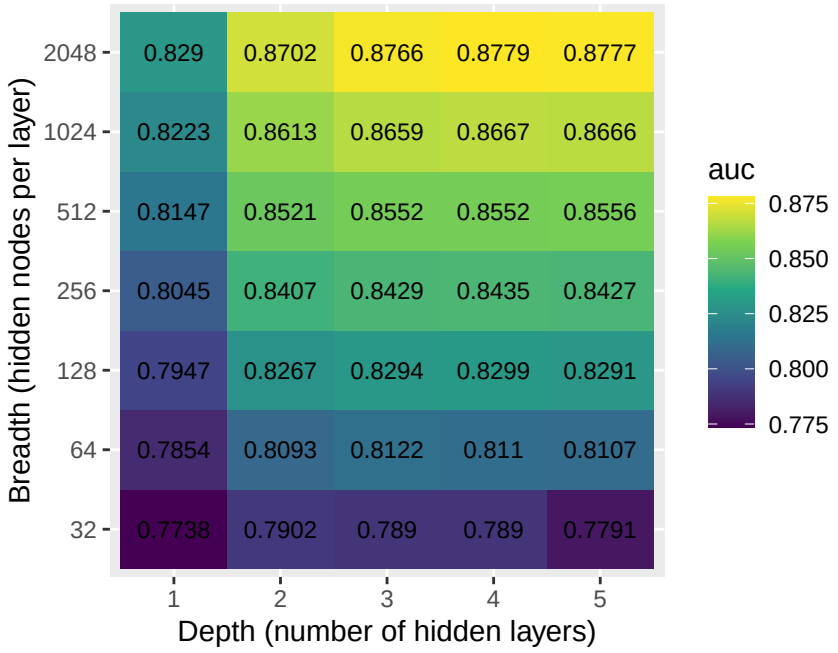
```
# Tuning values given defaults are not included in our parameter search for this
# project.
train_keras_auc ← function(x, y,
                           depth, breadth,
                           dropout = 0.5, learning_rate = 0.0002,
                           epochs = 50) {
  cat('DNN: ', depth, 'x', breadth, '\n')

  model ← keras_model_sequential()

  # By default, Keras applies Glorot uniform initialization for weights
  # and zero initialization for biases. Glorot uniform initialization
  # samples weights from Uniform(-sqrt(6/n),sqrt(6/n)) where n is the
  # sum of in and out nodes between two input/hidden/output layers.
```

```
  # first hidden layer
  model ▷
    layer_dense(units = breadth, activation = 'relu', input_shape = ncol(x)) ▷
    layer_dropout(rate = dropout)

  # subsequent hidden layers
  if (depth > 1) {
    for (layer in seq(2,depth)) {
      model ▷
        layer_dense(units = breadth, activation = 'relu') ▷
        layer_dropout(rate = dropout)
    }
  }

  # output layer (logistic activation function for binary classification)
  model ▷
    layer_dense(units = 1, activation = 'sigmoid')

  # compile model
  model ▷
    keras::compile(
      loss = 'binary_crossentropy',
      optimizer = optimizer_adam(learning_rate = learning_rate),
      metrics = metric_auc()
    )

  # a larger batch size trains faster but uses more GPU memory
  history ← model ▷ fit(x, y, epochs = epochs, batch_size = 8192, validation_split = 0.2)
  ypred ← model ▷ predict(x, batch_size = 8192) ▷ as.vector()
  auc ← roc(y,ypred) ▷ auc() ▷ as.numeric()

  rm(model)
  gc()
  k_clear_session()
  tensorflow::tf$compat$v1$reset_default_graph()

  auc
}
```

The following code computes and plots the AUC for NNs with 1 to 5 hidden layers and from 16 to 2048 hidden nodes in each hidden layer:

```
# Try NN training for different NN depths and breadths.  Cache using .RData file.
tensorflow::set_random_seed(42, disable_gpu = F)
if (!file.exists('cache/nn_results2.RDdata')) {
  nn_results ← tibble(depth = integer(), breadth = integer(), auc = numeric())

  for(l in 1:5) { # depth: number of hidden layers
    for (n in 2^c(5:11)) { # breadth: hidden nodes per layer
      nn_results ← nn_results ▷
        add_row(depth = l, breadth = n, auc = train_keras_auc(x, y, l, n))
    }
  }
  save(nn_results, file = 'cache/nn_results2.RDdata')
}
load('cache/nn_results2.RDdata')
tensorflow::set_random_seed(42, disable_gpu = F)
```

```
# heatmap of AUC vs depth and breadth
nn_results ▷ ggplot(aes(as.factor(depth), as.factor(breadth), fill = auc)) +
  geom_tile() +
  geom_text(aes(label = round(auc,4)), color = "black", size = 3) +
  scale_fill_viridis_c() +
  xlab('Depth (number of hidden layers)') +
  ylab('Breadth (hidden nodes per layer)')
```



The results show that for NNs with at least 64 nodes per hidden layer, there is little difference between having three layers or more; however, there is a slight advantage to having three hidden layers compared to two. In all cases, the widest NN at each depth provides the best AUC. Therefore, the 3x2048 NN appears to provide the best trade-off between network complexity and AUC. If memory or compute resources are limited, a 2x512 NN also appears to give reasonable estimation power.

3.3 Dropping high-level features

Since the high-level features in the HIGGS dataset are derived from the other, low-level features, it is clear that a good model should be able to learn the data without use of these features. The following code trains a 3x2048 NN on the HIGGS dataset once using all features and once just the low-level features:

```
tensorflow::set_random_seed(42, disable_gpu = F)
if (!file.exists('cache/nn_results3.RDdata')) {
  auc_low_only ← train_keras_auc(x[,1:21], y, 3, 2048, learning_rate = 2e-4)
  save(auc_low_only, file='cache/nn_results3.RDdata')
}
load('cache/nn_results3.RDdata')
tensorflow::set_random_seed(42, disable_gpu = F)
```

The following code displays the AUC for the both models:

```
tibble(
  Features = c('All', 'Low-level only'),
  AUC = c(
    nn_results ▷ filter(breadth == 2048 & depth == 3) ▷ pull(auc),
    auc_low_only
```

```
  )
) ▷ kable(align = 'lr', booktabs = T, linesep = '')
```

Features	AUC
All	0.8766273
Low-level only	0.8663024

The AUC for the low-level-only NN is only slightly lower than that for the full network, demonstrating that the NN is able to learn the complex non-linear relationships between the independent and dependent variables on it its own without the need for additional derived variables.

4 Final model selection and validation

Based on the results in Section 3, we will use a 3x2048 NN for our final model, using all available input features.

```
train_keras_model ← function(x, y, depth, breadth,
                         dropout = 0.5, learning_rate=0.0002, epochs = 50,
                         filename) {
  model ← keras_model_sequential(input_shape = ncol(x))

  # hidden layers
  for (layer in seq(depth)) {
    model ▷ layer_dense(breadth, 'relu') ▷ layer_dropout(rate = dropout)
  }

  # output layer (logistic activation function for binary classification)
  model ▷ layer_dense(1, 'sigmoid')

  # compile model
  model ▷
    keras::compile(
      loss = 'binary_crossentropy',
      optimizer = optimizer_adam(learning_rate = learning_rate),
      metrics = metric_auc()
    )

  # a larger batch size trains faster but uses more GPU memory
  history ← model ▷
    fit(x, y, epochs = epochs, batch_size = 8192, validation_split = 0.2)

  # need to save model BEFORE clean-up below
  save_model_hdf5(model, filename, include_optimizer = F)

  rm(model)
  gc()
  k_clear_session()
  tensorflow::tf$compat$v1$reset_default_graph()
}

tensorflow::set_random_seed(42, disable_gpu = F)
if (!file.exists('cache/final_nn.hdf5')) {
  train_keras_model(
    x, y, 3, 2048, learning_rate = 2e-4, filename = 'cache/final_nn.hdf5'
  )
}
finalModel ← load_model_hdf5('cache/final_nn.hdf5')
tensorflow::set_random_seed(42, disable_gpu = F)
```

The AUC of the ROC curve for the final test set is:

```
yPred ← finalModel$predict(xFinalTest) ▷ drop() # drop length-1 dimensions
auc(roc(yFinalTest,yPred))
```

```
## Area under the curve: 0.8772
```

5 Conclusion

We examined the HIGGS dataset, containing 11 million simulated particle collision events and 28 explanatory variables (21 low-level variable and 7 dervived or high-level variables). The goal is to predict for each event whether a Higgs boson was generated. A neural network was created using Keras, with hyperparameter tuning for the learning rate and NN size. It was found that additional hidden layers beyond three did not significantly improve prediction ability. The final NN was generated with three hidden layers of 2048 nodes each, generating a final area under the ROC curve of **0.877**.

The choice of neural networks for prediction, using Keras, was motivated by the large size of the HIGGS dataset. Nevertheless, working with the prediction model required careful memory management and occasionally restarting the entire R session and/or RStudio.

Another possible approach for the HIGGS dataset is boosted trees, where each added tree attempts to predict the remaining error after all previous trees are applied. R libraries include XGBoost and lightGBM, however, lightGBM requires manual installation while the pre-built R package for XGBoost with GPU support is marked as experimental (https://xgboost.readthedocs.io/en/stable/install.html#r). In contrast, Keras is easy to install, thanks to the command `keras::install_keras()` from within R itself, which also prepares the necessary Python environment containing Tensorflow and all other Keras dependencies. Thus, only CUDA and CUDNN need to be installed separately.

References

Asadi, Behnam, and Hui Jiang. 2020. "On Approximation Capabilities of ReLU Activation and Softmax Output Layer in Neural Networks." https://doi.org/10.48550/arXiv.2002.04060.

Baldi, P., P. Sadowski, and D. Whiteson. 2014. "Searching for Exotic Particles in High-Energy Physics with Deep Learning." *Nature Communications* 5 (1). https://doi.org/10.1038/ncomms5308.

Kingma, Diederik P., and Jimmy Ba. 2014. "Adam: A Method for Stochastic Optimization." https://doi.org/10.48550/arXiv.1412.6980.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15 (56): 1929–58. http://jmlr.org/papers/v15/srivastava14a.html.