Deep Learning Workflow and Results

Here we will be doing a quick foray into deep learning using the 10X genomics data published in July 2019. This data tracks around 5000 PBMC cells. For each cell, the data specify gene expression data, protein expression data, and antigen specificity data.

Broadly, our goal is to create a machine learning approach to predict protein expression from gene expression data. This model, if it is effective, could be useful for predicting cell type, imputing sparse protein counts, or transferring to different scRNA-seq data sets to generate new protein information.

We have a lot of data here to work with, but it is not obvious how best to approach building the model. I decided to use a basic neural net, as they have shown excellent performance recently in transcriptomic problems, and are ideal for predicting complex nonlinear functions such as protein values. Other options include decision forests and linear regression, both of which might work here and have relative benefits such as simplicity and readability.

Please note that several hyperparameters have been adjusted to help the vignette run quickly. I will mark these spots with more optimal values.

I will be using BioConductor/SingleCellExperiment to store and process the data, before feeding it into the Keras interface to TensorFlow to build the actual neural network. I will mostly show the best setup I have found, with notes explaining how I tested and made decisions. With that said, let's dive into the problem.

Cut up the data

We will begin with processing and subsetting of the data, to get it into a form that can be fed into Keras.

```
set.seed(31415)
library(DropletUtils)
library(BiocManager)
library(SingleCellExperiment)
trainingSCE <- read10xCounts("~/Documents/10Xdata/5k_pbmc_protein_v3_nextgem_filtered_feature_bc_matrix
testingSCE <- read10xCounts("~/Documents/10Xdata/5k_pbmc_protein_v3_filtered_feature_bc_matrix.h5")

trainingSize <- ncol(trainingSCE)
testingSize <- ncol(testingSCE)

# singler = CreateSinglerSeuratObject(counts =
# 'GSE74923_series_matrix.txt', annot = 'GSE74923_types.txt',
# project.name = 'GSE74923', min.genes = 500, min.cells = 2,
# npca = 10, regress.out ='nUMI', technology= 'C1', species =
# 'Mouse', citation = 'Kimmerling et al.
# 2016',reduce.file.size = T, variable.genes = 'de',
# normalize.gene.length = T)


# sum(rowData(trainingSCE) != rowData(testingSCE)) This is a
# good sanity check -> the rows in each SCE are equivalent.
# This will not always be the case

# this is a boolean list with TRUE in the indexes of proteins
proteinList <- (rowData(trainingSCE)$Type == "Antibody Capture") &
    !grepl("IgG*", rowData(trainingSCE)$Symbol)
getProteinMatrix <- function(x) {
```

```
    # from an SCE, gets the transposed matrix of proteins
    x[proteinList] %>% counts() %>% as.matrix() %>% t() %>% clr()  # we perform a centered log ratio tr
}

# tensorflow breaks if fed dashes, so we replace dashes in
# the protein names with underscores
fixProteinColNames <- function(x) {
    colnames(x) <- sub("-", "_", colnames(x))
    x
}

library(purrr)
library(compositions)
proteinTrainingData <- trainingSCE %>% getProteinMatrix() %>%
    fixProteinColNames()
proteinTestingData <- testingSCE %>% getProteinMatrix() %>% fixProteinColNames()

remove(fixProteinColNames)

geneTrainingSCE <- trainingSCE[rowData(trainingSCE)$Type == "Gene Expression"]
geneTestingSCE <- testingSCE[rowData(testingSCE)$Type == "Gene Expression"]
```

Cleaning the data

Now we will clean and prepare the data to be used in the model. We use some basic techniques like filtering out poorly-expressed genes and normalizing the GEX matrix.

```
# just like for the proteins, we need to turn the delayed
# matrices to matrices
counts(geneTrainingSCE) <- as.matrix(counts(geneTrainingSCE))
counts(geneTestingSCE) <- as.matrix(counts(geneTestingSCE))

# keep genes with at least 1 read in at least 1% of cells
# (remove low frequency genes)
min_reads <- 1
min_cells <- 0.01 * ncol(geneTrainingSCE)
# calculate a boolean vector for which genes to keep
keep <- rowSums(counts(geneTrainingSCE) >= min_reads) >= min_cells
# subset both SCEs using this vector
geneTrainingSCE <- geneTrainingSCE[keep, ]
geneTestingSCE <- geneTestingSCE[keep, ]

# for readability, swap rownames from row ID to row symbol
library(scater)
featureNames <- uniquifyFeatureNames(ID = rowData(geneTrainingSCE)$ID,
    names = rowData(geneTrainingSCE)$Symbol)
rownames(geneTrainingSCE) <- featureNames
rownames(geneTestingSCE) <- featureNames

# normalize
geneTrainingSCE <- scater::normalize(geneTrainingSCE)
geneTestingSCE <- scater::normalize(geneTestingSCE)
```

Select Data

Here is where we decide which genes and proteins to train on. I decided to train the model on only the more highly variable genes. Hopefully this reduces noise in the model and makes it easier and faster for the model to train. I also decided to remove CD127 from the list of proteins to predict, as the actual values for it are mostly 0.

```
# find highly variable genes
library(scran)
fit <- trendVar(geneTrainingSCE, use.spikes = FALSE)
dec <- decomposeVar(geneTrainingSCE, fit)
hvg <- dec$bio > 0  # get biologically relevant genes

getGeneData <- function(x) {
    # subset both SCE's with that set of genes
    x[hvg, ] %>% logcounts() %>% t()
}

geneTrainingData <- getGeneData(geneTrainingSCE)
geneTestingData <- getGeneData(geneTestingSCE)

remove(getGeneData)
```

Building the model

I tried many iterations of the model's hyperparameters, and found some with decent performance on a few important proteins. Here is a compilation of what I know about these hyperparameters, and what they do. This list is sorted with most important first.

- Dropout - Dropout is the process of choosing randomly some proportion of nodes from a layer (often the input layer) to get dropped completely. This helps ensure that the model is training using all available data, as sometimes key genes are not included. Using dropout does seem to have a large positive effect on performance. Good values seem to range from ~.2 to ~.5.
- Learning rate - The rate at which the model changes its parameters. Values between 0.0002 and 0.001 seem to work fine.
  - If this is too high, the model will jump between local minima and potentially never converge. In the loss graph, this correlates to great variance in loss between adjacent epochs. If the model has not converged by the end of the run, it is probably losing a lot of prediction accuracy, so this is a problem
  - If this is too low, the model will simply take a long time to converge, which is not a big issue.
- Layer sizes - This is one I don't fully understand. One thing I have noticed is that too few layers in a bottleneck (less than 4?) is a bad thing. The number of nodes in a bottleneck layer represents the dimensionality through which the input space is forced. Too few, and the information carried by the bottleneck will not be enough to reconstruct important features.
- Number of layers - Again, I haven't seen much difference. I also haven't spent too much time looking. Usually ~3 is a solid number. Too many less and there is no space to approximate hard non-linear functions. Too many more and the model will have a hard time making use of the added complexity.
- Activation function - RELU is an activation function which zeros out any activation below 0, and sigmoid squishes any value asymptotically between 0 and 1. Basically the difference is that RELU destroys negative values, and sigmoid doesn't. I dabbled briefly with sigmoid activation and it ruined everything. I have been sticking to RELU. I think it does a good job of zeroing out noise in a lot of cases. I wouldn't be opposed to having a sigmoid layer here or there, but I do think the model requires 1 relu layer at a minimum.

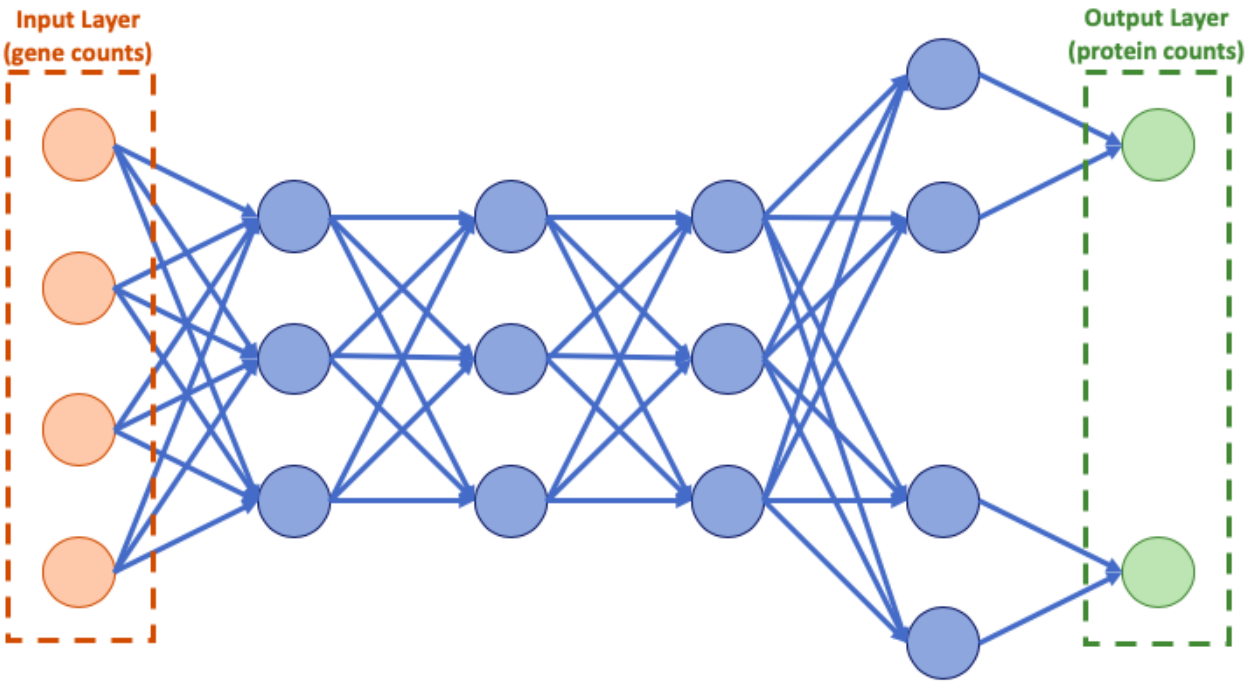


Figure 1: A simplified view of the neural net architecture. Layer sizes from left to right were actually 64, 32, 4, 8, 4, 1, as opposed to the 4, 3, 2, 3, 2, 1 shown here.

- Batch size - This is the number of samples that the model goes through before it decides which direction to train. Generally a power of 2, often 32, 64, or 128. Small sizes will mean the model is often pulled different directions towards different local minima, whereas large sizes will cause the model to move in the same direction. I find that 128 works nicely, but haven't played with it too much.
- Optimizer function / Loss function - I didn't test either of these, but I will include a note about them here. The optimizer function used is RMSprop

Evolution of model:

1. I started with only a couple of layers, with high dimensionality (around 64), predicting only a single protein expression. This was taking a long time to train the network with this large parameter space to train.
2. The training time went down a lot as I brought down the dimension of the layers, closer to 8 or 4. This didn't seem to affect the performance, so I kept it to keep training times down.
3. I added a dropout layer, and this vastly improved performance.
4. I added the extra proteins, which actually decreased the performance of the couple proteins I was originally looking at.

```
library(keras)

# set the input layer to be the right dimension
inputs <- layer_input(shape = c(dim(geneTrainingData)[2]))

# create the first FOUR layers
output0 <- inputs %>%
  # each of these lines creates a new layer, and specifies the number of nodes and the activation funct
  layer_dropout(rate = 0) %>%
  layer_dense(units = 256, activation = 'relu') %>%
  layer_dense(units = 128, activation = 'relu') %>%
```

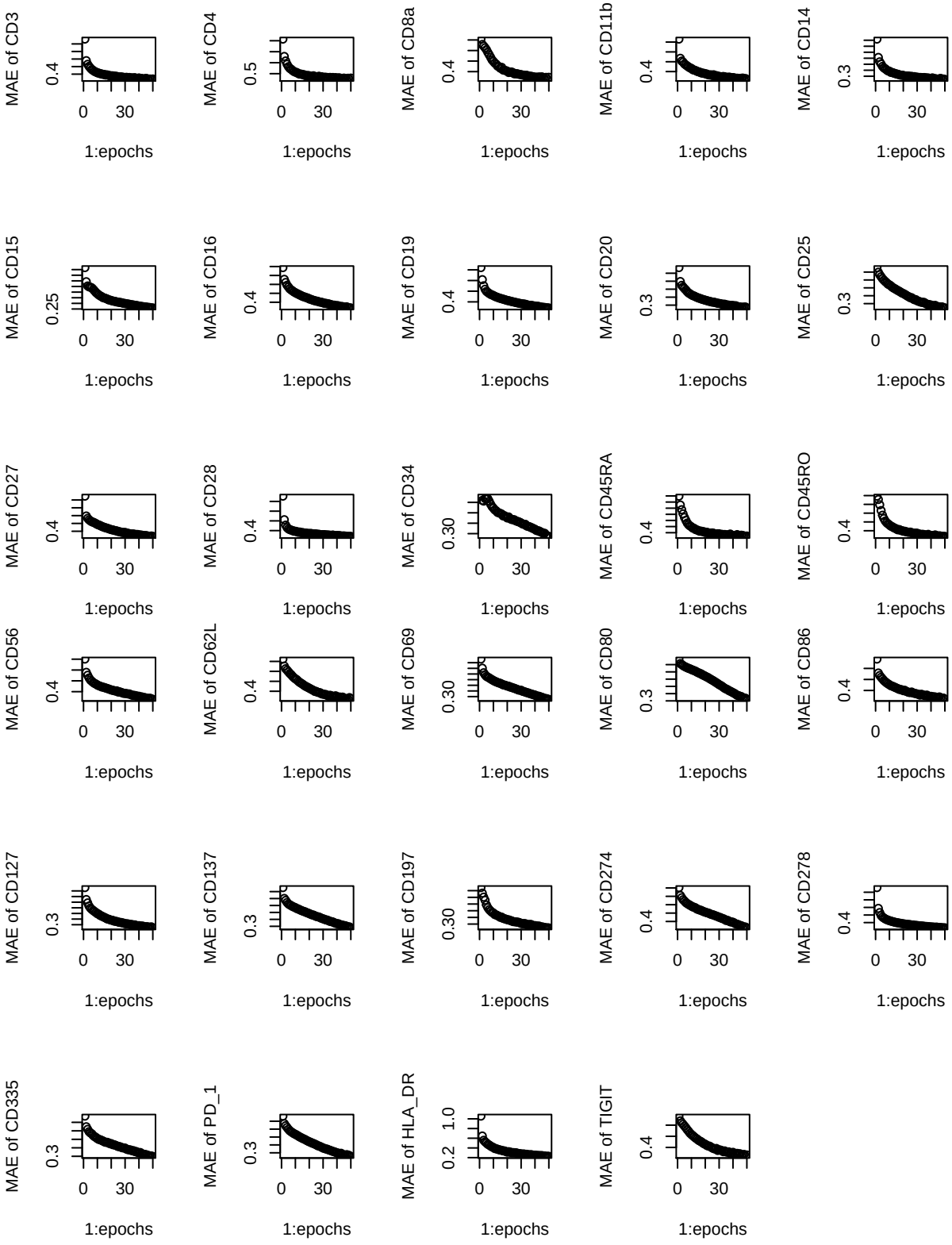
```
  layer_dense(units = 64, activation = 'relu')

# all of the layers are shared up until the last two, during which each protein gets its own set of 8 n
makeOutputLayer <- function(x) {
  layer_dense(output0, units=32) %>%
    layer_dense(units=1, name= colnames(proteinTrainingData)[x])
}

# for each protein, create the last 2 layers and add them to the original 4. See the architecture diagr
outputs <- lapply(1:dim(proteinTrainingData)[2], makeOutputLayer)

model <- keras_model(inputs=inputs, outputs=outputs)

model %>% compile(
  loss = "mse",
  optimizer = optimizer_rmsprop(lr = .0001), # learning rate here
  metrics = list("mean_absolute_error"),
)

model %>% summary()

# Display training progress by printing a single dot for each completed epoch.
print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
    if (epoch %% 50 == 0) cat("\n")
    cat(".")
  }
)
# epochs refers to how many passes through the data we make, and affects run time linearly. we can use
epochs <- 50
batch_size <- 128
```

```
set.seed(42)
# Train the model
history <- model %>% fit(x = geneTrainingData, y = lapply(1:dim(proteinTrainingData)[2],
    function(x) {
        proteinTrainingData[, x]
    }), batch_size = batch_size, epochs = epochs, validation_split = 0.2,
    verbose = 0, callbacks = list(print_dot_callback))
```

Here we can take a look at some interesting statistics from the model process.

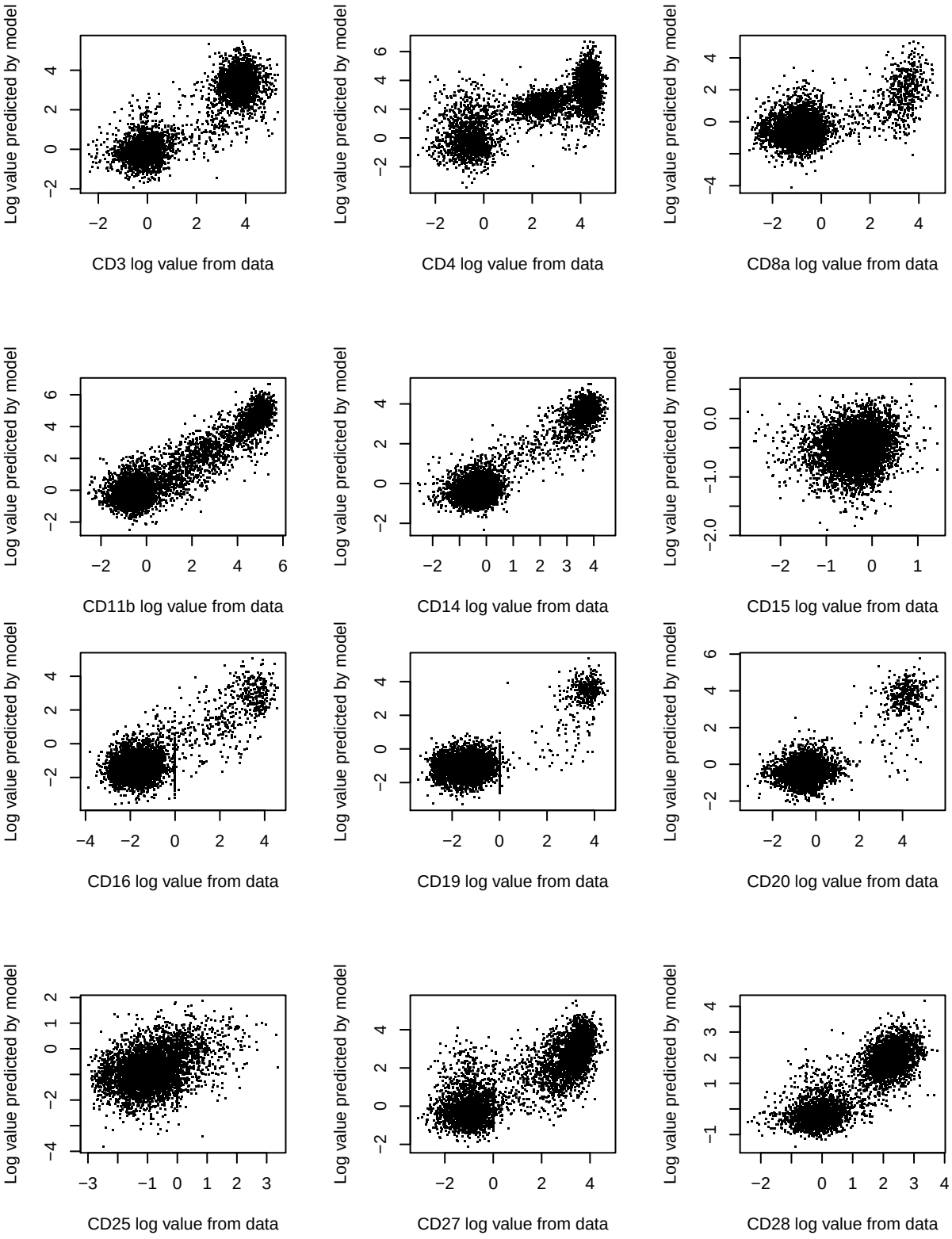
First we see a set of plots that show the model's loss over time for four proteins, CD45RA, CD8a, CD45RO, and CD279_PD_1. We can clearly see that as the model trains, it becomes more effective at predicting these values.

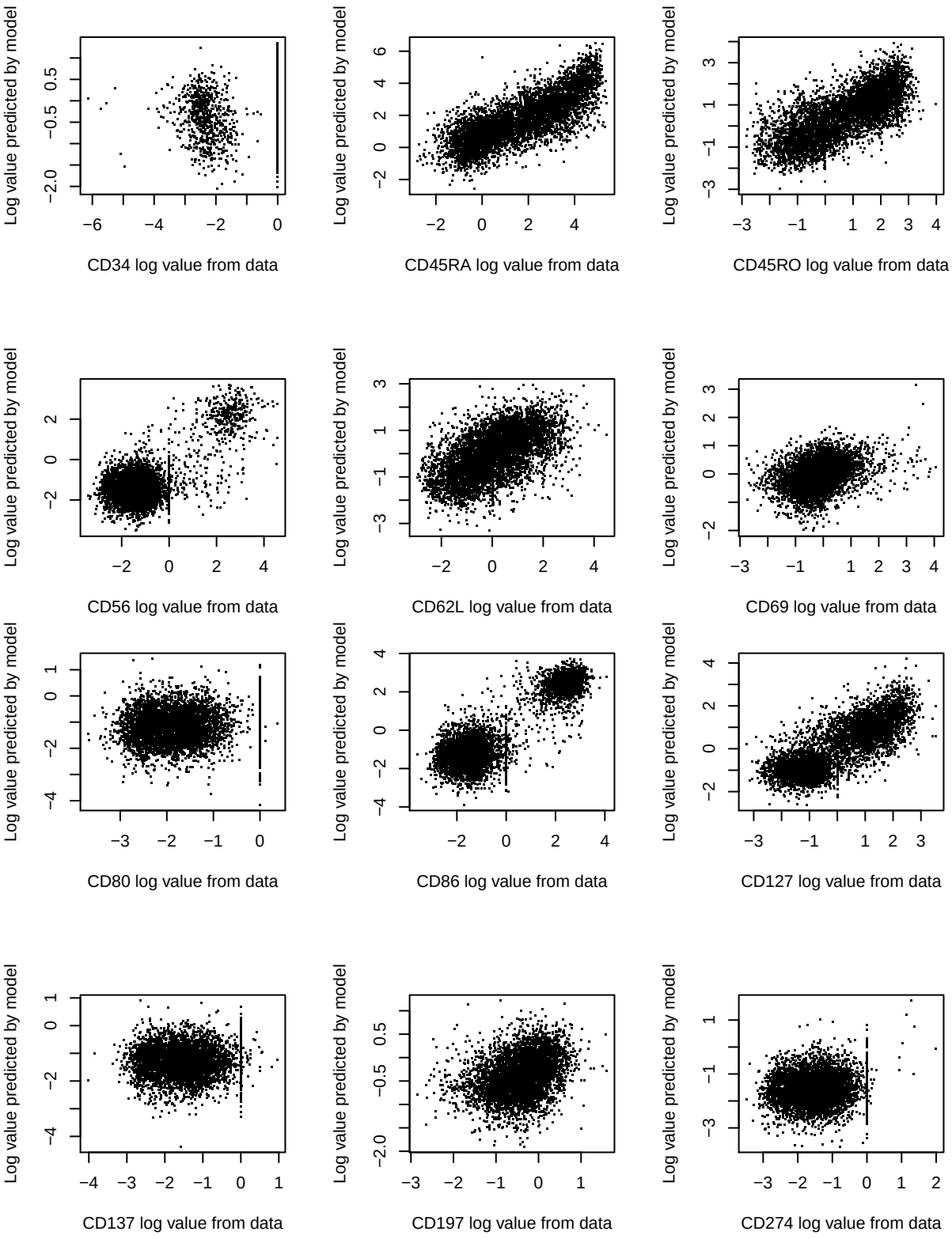
```
lossPlots()
```

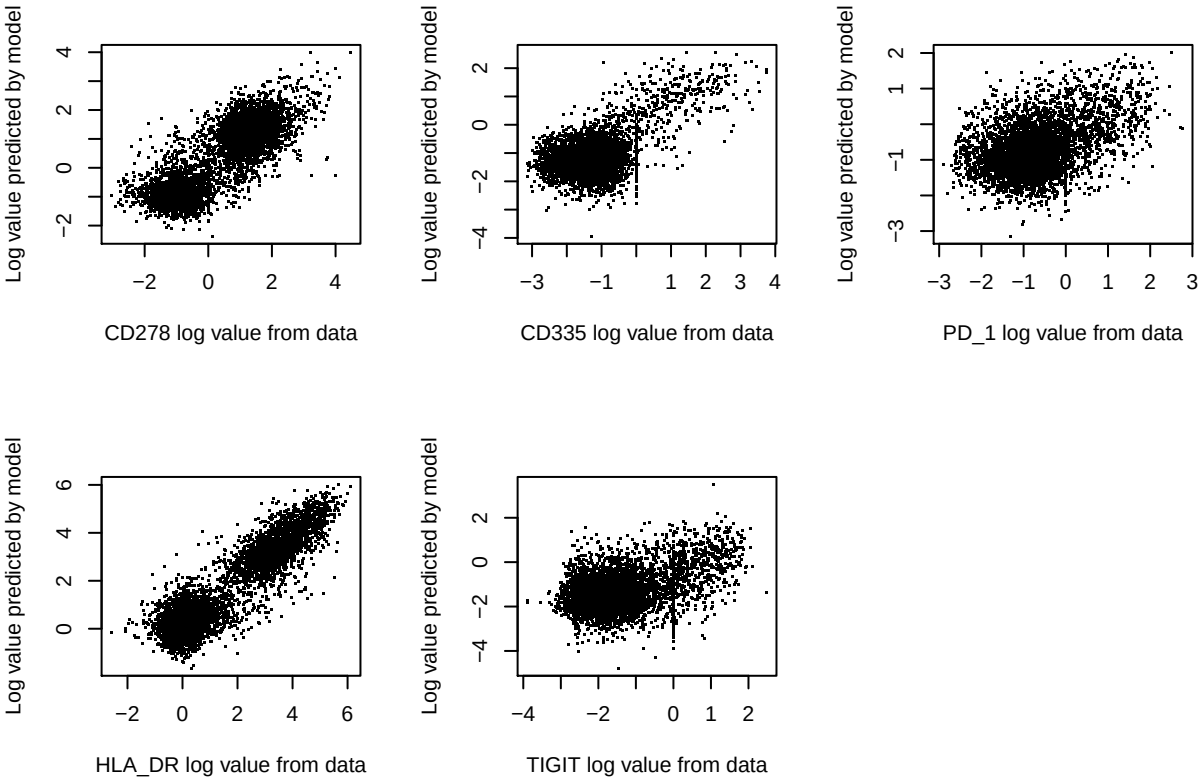


Here we can see a scatter plot showing on one axis the actual protein values for each cell, and on the other axis, the model's predicted values for those same proteins.

```
scatters()
```


Log value predicted by model
CD3 log value from data
CD4 log value from data
CD8a log value from data
CD11b log value from data
CD14 log value from data
CD15 log value from data
CD16 log value from data
CD19 log value from data
CD20 log value from data
CD25 log value from data
CD27 log value from data
CD28 log value from data


Log value predicted by model
CD34 log value from data
CD45RA log value from data
CD45RO log value from data
CD56 log value from data
CD62L log value from data
CD69 log value from data
CD80 log value from data
CD86 log value from data
CD127 log value from data
CD137 log value from data
CD197 log value from data
CD274 log value from data



And finally, I printed out a simple list of the correlation between the model's predictions and the actual values. As you can see, many of the proteins are not able to be predicted at all by the model.

```
correlations()
```

```
## [1] "The correlation between predicted and actual values for CD3 is 0.909676231866245"
## [1] "The correlation between predicted and actual values for CD4 is 0.800834107007638"
## [1] "The correlation between predicted and actual values for CD8a is 0.629210862362377"
## [1] "The correlation between predicted and actual values for CD11b is 0.931843737533176"
## [1] "The correlation between predicted and actual values for CD14 is 0.936885248282747"
## [1] "The correlation between predicted and actual values for CD15 is 0.153514643371158"
## [1] "The correlation between predicted and actual values for CD16 is 0.716966122621897"
## [1] "The correlation between predicted and actual values for CD19 is 0.787549352831161"
## [1] "The correlation between predicted and actual values for CD20 is 0.810730373629228"
## [1] "The correlation between predicted and actual values for CD25 is 0.334431388476539"
## [1] "The correlation between predicted and actual values for CD27 is 0.854318684188813"
## [1] "The correlation between predicted and actual values for CD28 is 0.877458752195086"
## [1] "The correlation between predicted and actual values for CD34 is 0.232352697952568"
## [1] "The correlation between predicted and actual values for CD45RA is 0.789499918672195"
## [1] "The correlation between predicted and actual values for CD45RO is 0.741998263754516"
## [1] "The correlation between predicted and actual values for CD56 is 0.692066742768106"
## [1] "The correlation between predicted and actual values for CD62L is 0.557031415526623"
## [1] "The correlation between predicted and actual values for CD69 is 0.345073264067854"
## [1] "The correlation between predicted and actual values for CD80 is 0.0565858338391011"
## [1] "The correlation between predicted and actual values for CD86 is 0.876559036179337"
## [1] "The correlation between predicted and actual values for CD127 is 0.794954486784606"
## [1] "The correlation between predicted and actual values for CD137 is 0.00728490340459858"
## [1] "The correlation between predicted and actual values for CD197 is 0.317958561820638"
## [1] "The correlation between predicted and actual values for CD274 is 0.101296365055485"
## [1] "The correlation between predicted and actual values for CD278 is 0.821256905500665"
```

```
## [1] "The correlation between predicted and actual values for CD335 is 0.545541624197315"
## [1] "The correlation between predicted and actual values for PD_1 is 0.418979847272021"
## [1] "The correlation between predicted and actual values for HLA_DR is 0.915287009975571"
## [1] "The correlation between predicted and actual values for TIGIT is 0.342229247821346"
## [1] "The sum of squared correlations is 12.7911937706668"
## [1] "The average correlation is 0.441075647264374"
```

Discussion

Overall the results are relatively good. Considering that we are predicting within a single cell type, a good fraction of the variance in the protein expression can be explained by the gene quantities. If the model contained multiple types of cells, a lot more of the variance probably could be explained, as it is easier to predict expression using cell type markers than using other genes.

This method can easily be applied to other data sets, as it is very extensible and makes almost no assumptions. In fact, it will likely have more impressive performance on other data sets, as it is able to lean on cell type markers.