Babraham Bioinformatics

Exercises: Introduction to Machine Learning

Version 2023-08

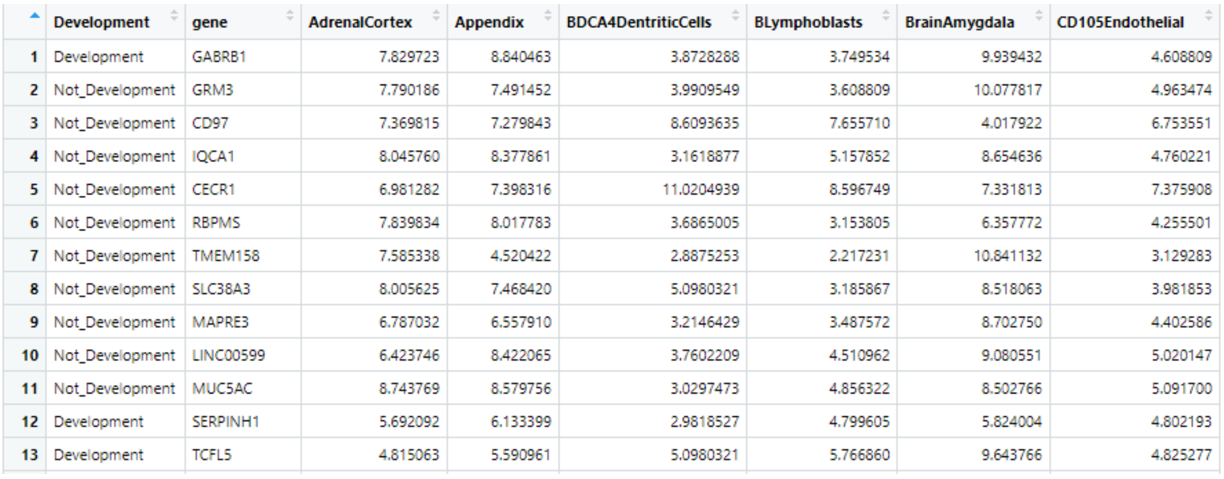
Licence

Exercise 1: Running machine learning models

In this exercise we have given you a filtered subset of the data in GSE1133, which is a microarray study measuring gene expression across a panel of around 90 different tissues.

The aim of the model is to try to predict which genes are involved in development. This is defined based on the “Developmental Process” Gene Ontology category (GO:0032502).

A snapshot of the first part of the data looks like this:



	Development	gene	AdrenalCortex	Appendix	BDCA4DentriticCells	BLymphoblasts	BrainAmygdala	CD105Endothelial
1	Development	GABRB1	7.829723	8.840463	3.8728288	3.749534	9.939432	4.608809
2	Not_Development	GRM3	7.790186	7.491452	3.9909549	3.608809	10.077817	4.963474
3	Not_Development	CD97	7.369815	7.279843	8.6093635	7.655710	4.017922	6.753551
4	Not_Development	IQCA1	8.045760	8.377861	3.1618877	5.157852	8.654636	4.760221
5	Not_Development	CECR1	6.981282	7.398316	11.0204939	8.596749	7.331813	7.375908
6	Not_Development	RBPMS	7.839834	8.017783	3.6865005	3.153805	6.357772	4.255501
7	Not_Development	TMEM158	7.585338	4.520422	2.8875253	2.217231	10.841132	3.129283
8	Not_Development	SLC38A3	8.005625	7.468420	5.0980321	3.185867	8.518063	3.981853
9	Not_Development	MAPRE3	6.787032	6.557910	3.2146429	3.487572	8.702750	4.402586
10	Not_Development	LINC00599	6.423746	8.422065	3.7602209	4.510962	9.080551	5.020147
11	Not_Development	MUC5AC	8.743769	8.579756	3.0297473	4.856322	8.502766	5.091700
12	Development	SERPINH1	5.692092	6.133399	2.9818527	4.799605	5.824004	4.802193
13	Development	TCFL5	4.815063	5.590961	5.0980321	5.766860	9.643766	4.825277

The stats for the dataset are:

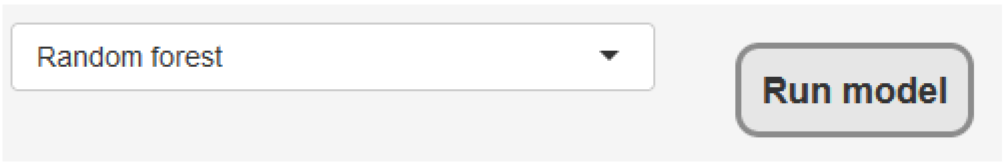
- There are 1241 measured genes
- 522 of the genes are development genes, 719 are not
- There are 92 variables (tissues) we can use for prediction. All of them are quantitative so are compatible with all model types
- The variable to predict is Categorical (Development or Not Development) so we can only use models with a Categorical output.
- Although there is a gene column we aren’t going to use that as there’s no way that this can be predictive since it’s a categorical value which is different for every gene.

Running Models

To let you try out some of these models you can go to:

https://www.bioinformatics.babraham.ac.uk/shiny/machinelearning/

Where we have built a simple interface which lets you run a variety of different model types on this data. Just select the model you want to run from the drop down box and press the “Run Model” button.



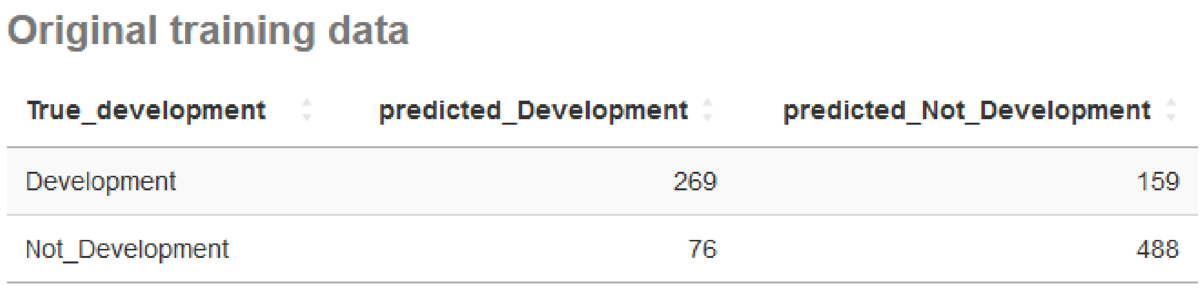
After the model has run you will see some information about the model on the left which summarises the parameters which were used to run it – you should be able to match these to the theory we talked about before.

On the right you will see a summary of some predictions made by the model. We have run two sets of data through the model.

1. We re-ran the data used to train the model back through it to see how well it is able to predict data it has seen before.

2. We set aside a portion of the original data before training the model and then ran this through the model after it was trained to see how well it works against data it hasn't seen before.

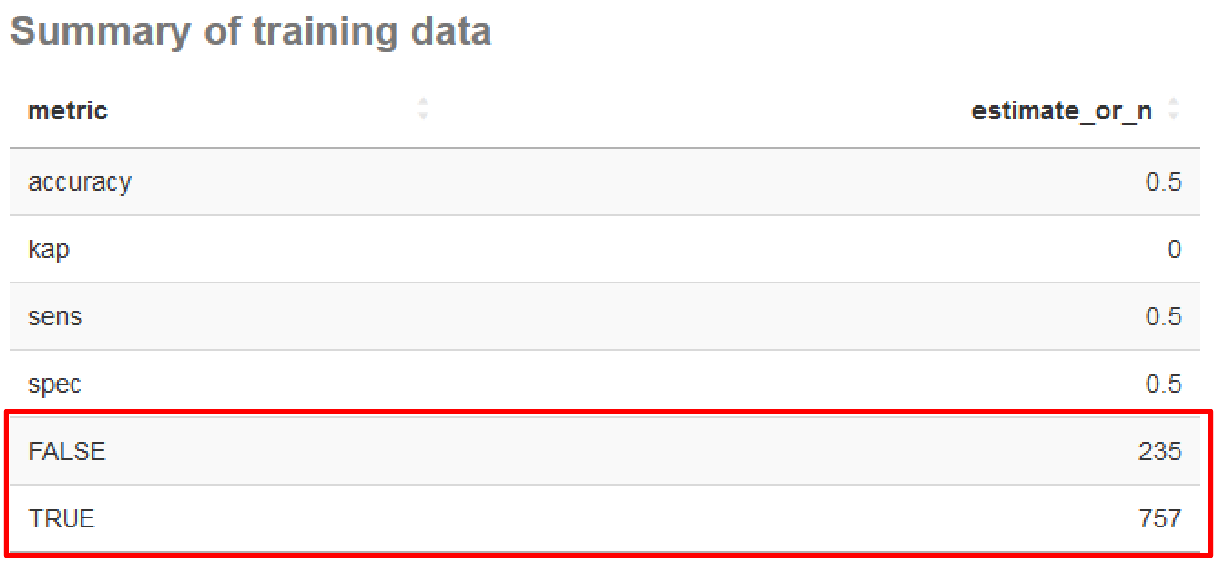
Results



Original training data

True_development	predicted_Development	predicted_Not_Development
Development	269	159
Not_Development	76	488

This table shows a summary of the predictions the model made and how they matched against the known correct values in the data. It's important to validate a model against data where you know the answer, before using it to make predictions on data where the answer isn't known.



Summary of training data

metric	estimate_or_n
accuracy	0.5
kap	0
sens	0.5
spec	0.5
FALSE	235
TRUE	757

In this table you can see the total number of correct (TRUE) and incorrect (FALSE) predictions the model made.

Questions:

Run the different models and look at their output and the summary of the predictions they make then answer the questions below.

1. Do all of the models perform similarly well, or are some better than others?

2. Do the models perform similarly well on the data they have seen before and the data they haven't seen before?

3. Do the more complex models perform better than the simpler ones?

4. If you run each model a couple of times, do the results change? If they only change for some of the models why is this?

5. If you hadn't run a model, but had simply assigned the most frequent category (Not Development) to every prediction, how many correct answers would you expect to have seen in the test data of 249 samples? Do any of the models do substantially better than this?

Changing Model Parameters

We have a second interface which lets you rerun the random forest model whilst changing the parameters used to construct it:

https://www.bioinformatics.babraham.ac.uk/shiny/optimising_model/

In this version you can change the total number of trees constructed, the number of random predictors selected at each branch point, and the minimum number of measures which must appear in a node at the bottom of the tree so it doesn't get too complex at the bottom of the tree.

Try running the model a few times and seeing what effect changing these parameters has on the results.

What settings would you have to use to mimic a conventional decision tree?

What do you think the effect of changing the different parameters would be? Do you see this in the results?

Exercise 2: Evaluating Models

Now that you have learned about the way that models can be tested you can initially look back at the results you found in exercise 1 to see the additional metrics which were supplied alongside the raw results.

1. Are the models actually identifying developmental genes at a rate which is significantly higher than you'd get by guessing?

2. What is the balance in the models between sensitivity (the ability to say that a developmental gene is a developmental gene) and specificity (the ability to identify non-developmental genes)?

3. Are there differences in the sensitivity / specificity trade-off between the models?

4. Which models appear to be most strongly overfitted to the training data (do well on training, and poorly on testing)?

Exercise 3: Building your first model

In this exercise you will build your first tidymodels model. You can going to use a dataset comprising all of the canonical proteins in the mouse genome. For each of these you have some basic information about the gene, transcript and protein, plus you have the compositional break down of the protein into its component amino acids.

The aim of your model is to predict which of these proteins contains one or more transmembrane segments, such that the protein is normally found embedded within a membrane.

To do this you are going to build and train a random forest model. The steps in the modelling procedure will be:

1. Load the R packages we're going to need for this analysis
2. Load in the original data
3. Prepare the data for modelling
    a. Convert the variable to predict to be a factor
    b. Remove proteins with missing data
    c. Shuffle the rows
4. Split the data into a training and testing subset
5. Build the model
6. Train the model using the training data
7. Predict the transmembrane proteins from the testing data
8. Check how good the predictions are

Below we will talk you through how to construct a script in RStudio to perform all of these steps. In an actual modelling experiment we would include more evaluation of the data before starting on the modelling, so this is a somewhat truncated version of the full procedure you'd use.

To get started you need to open a new R script, save it, then set the location of the data you're going to use.

Setting up your environment

Inside RStudio select

File > New File > R Script

Once the script has opened go to **File > Save As** and save it into the **MachineLearningData** folder in a file called **model.R**

In the RStudio menu select **Session > Set Working Directory > To Source File Location**

Loading the R packages we need

We will be using two packages in this script, the tidyverse package, which will do the general data manipulation for us, and the tidymodels package which will do the modelling.

We can load these with

```
library(tidyverse)
library(tidymodels)
tidymodels_prefer()
```

The last line here simply says that we should always use functions from tidymodels, even if another function with the same name, but from a different package exists.

Loading the input data

To load the data from the TSV file it's saved in we need to do

```
read_delim("transmembrane_data.txt") -> data
```

You can then click on the data in the Environment tab (top right) and have a look at what the data looks like.

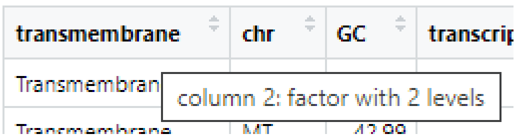
Preparing the data for modelling

Turing transmembrane into a factor

If a column is going to be used as the value to predict then it must have a data type of "factor" which is a data type specifically used to represent data which can hold one of a defined set of values. Our transmembrane predictions are currently just in a text column so we need to change that.

```
data %>%
  mutate(
    transmembrane = factor(transmembrane)
  ) -> data
```

After you've run this, hold your mouse over the transmembrane column header when looking at the data. It should now says that it is a factor



transmembrane	chr	GC	transcrip
Transmembran			
Transmembrane	MT	42.99	

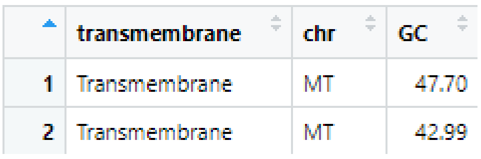
column 2: factor with 2 levels

Removing the gene_id column

In our data the gene_id column just holds the name of the gene. This isn't useful in the model and will just slow things down or cause them to overfit, so we need to remove it.

```
data %>%
  select(-gene_id) -> data
```

You should now see that the gene_id column has gone, and that the transmembrane column is now the first one.



	transmembrane	chr	GC
1	Transmembrane	MT	47.70
2	Transmembrane	MT	42.99

Shuffling the rows

For some types of model there may be information contained in the order the rows appear (for example if all of the transmembrane proteins were next to each other). To prevent this information from having any effect we can just shuffle all of the rows.

```
data %>%
  sample_frac() -> data
```

This won't change the structure of your data but where the data originally put all proteins from the same chromosome together you should now see that they are all mixed up.

Removing missing values

We will remove any rows in which any of the columns have missing values.

```
data %>%
  na.omit() -> data
```

After running this you should see that the number of rows in the data goes down from 19,701 to 18,352.

Because we are going to run a random forest model this is all of the preparation we need to do. Later we may try other model types where we would need to make the data behave in a more quantitatively nice way, but tree based models really don't care.

Splitting the data

Before we construct the model we must split off some training data so that we aren't using the same data to test the model as we are to train it.

```
data %>%
  initial_split(prop=0.8, strata=transmembrane) -> split_data
```

This will split off 80% of our data to be used for training and 20% for testing.

We can see the data in the two subsets by running:

```
training(split_data)
```

..or..

```
testing(split_data)
```

You should see about 14,600 rows in the training data and about 3,600 in the testing.

Building the model

Now all the data is prepared we can go on and build a model. We're going to build a random forest model using the ranger engine. We also need to tell it that it's going to make a classification prediction.

```
rand_forest(trees=100) %>%
  set_engine("ranger") %>%
  set_mode("classification") -> forest_model
```

To see the model you can run

```
forest_model %>% translate()
```

Note that a lot of the options in the model fit template are set to "`missing_arg()`" which means that they are values we will need to supply later in the process.

Training the model

We now need to train the model. We are going to give it the training data from our split data, and we're going to tell it that it should try to predict the transmembrane values using all of the rest of the columns.

```
forest_model %>%
  fit(transmembrane ~ ., data=training(split_data)) -> forest_fit
```

Once the model is fit we can see it by running

```
forest_fit
```

We should see all of the variables for the model in place, and see some of the details of the data and the fit (number of variables and cases etc).

Testing the model

To test the model we need to use it to make predictions about data where we know the answer, which is what our testing data is for. We are going to use the predict function to make predictions on this data. To make a prediction we need to pass in a new dataset with the same variables as the training data and it will make predictions.

```
forest_fit %>%
  predict(testing(split_data))
```

Which will give us something like:

```
# A tibble: 3,671 × 1
   .pred_class
   <fct>
 1 Soluble
 2 Soluble
 3 Transmembrane
 4 Soluble
 5 Soluble
 6 Soluble
 7 Soluble
 8 Soluble
 9 Soluble
10 Soluble
```

The problem with this is that it only outputs the predictions, we don't see the rest of the data, including the column which says what the answer should have been, so we need to join those predictions to the training data

```
forest_fit %>%
  predict(testing(split_data)) %>%
  bind_cols(testing(split_data)) -> prediction_results
```

You can now click on the `prediction_results` in the environment window to see the predictions (in the `.pred_class` column) alongside the known correct answers (in the `transmembrane` column)

Evaluating the predictions

From the set of predictions we can now see how well the model actually did by comparing the predictions to the known true values.

We can start by simply counting the number of times we see different combinations of predictions and true values in the data.

```
prediction_results %>%
  group_by(transmembrane, .pred_class) %>%
  count()
```

From this you can see how many times a correct and incorrect prediction was made and the break down of the mistakes which were made.

We can also get more specific values for sensitivity and specificity

```
prediction_results %>%
  sens(transmembrane, .pred_class)
```

..and..

```
prediction_results %>%
  spec(transmembrane, .pred_class)
```

Finally we can get an overall accuracy value, and we can also get the Coehn’s kappa value to say whether we’re actually performing better than chance on the data.

```
prediction_results %>%
  metrics(transmembrane, .pred_class)
```

What is your evaluation of how well the model has performed? Feel free to try playing with the setup parameters for the model to see if you can improve on the initial performance. Remember though that there is a random component, so just because a model works better once doesn’t mean that those settings will always be better.

Exercise 4: Using Recipes and Workflows

We're going to build another model from the same transmembrane data as before, but this time we're constructing a neural net.

Because neural networks have more constraints on the data which goes into the model we're going to have to do more pre-processing, and we're going to have to apply this to both the training and testing data (and we'd have to do it to any unknown proteins in future), so we're going to automate this with a recipe and we're going to integrate this into a workflow to run it.

For the first part of the model where we:

1. Loaded the required packages
2. Loaded the data
3. Prepared the data
4. Split the data into training and testing

We can follow the same steps as before, or we can use the same `split_data` variable as for the random forest model.

Building a Recipe

Firstly we're going to build a recipe which will combine the formula for prediction and the training data. Once we have it we can then add steps to it to complete the pre-processing.

```
recipe(
  transmembrane ~ . ,
  data=training(split_data)
) -> neural_recipe
```

We can then view the recipe with

```
neural_recipe
```

Now we have a recipe we can add processing steps to it. The steps will be:

1. Log transform the `gene_length` and `transcript_length` columns
2. Z-Score normalise all of the numeric columns
3. Turn all of the text columns into dummy number columns

```
neural_recipe %>%
  step_log(gene_length, transcript_length) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_dummy(all_nominal_predictors()) -> neural_recipe
```

Look at the recipe again to see the new steps have been added.

Building the model

We can now create the neural network model. We're going to use a single hidden layer with 10 nodes in it. You could play around with the settings made here once you had the basic model in place.

```
mlp(
  epochs = 1000,
  hidden_units = 10,
  penalty = 0.01,
  learn_rate = 0.01
) %>%
  set_engine("brulee", validation = 0) %>%
  set_mode("classification") -> nnet_model
```

The arguments here are as follows:

- `epochs` = how many rounds of refinement (back propagation) the model goes through
- `hidden_units` = how many nodes we want in the hidden layer.
- `penalty` = a value which penalises complexity in the model to try to prevent overfitting
- `learn_rate` = how much the estimates are moved to try to optimise the model

Again, these values could be modified after generating an initial model, but these will give us something to work from.

We can see the model with

```
nnet_model %>% translate()
```

Building a workflow

A workflow will combine the recipe and the model together and will allow us to run everything at once.

```
workflow() %>%
  add_recipe(neural_recipe) %>%
  add_model(nnet_model) -> neural_workflow
```

We can view the workflow with

```
neural_workflow
```

Training the model via the workflow

To train the model we run the `fit` function and pass in our training data. This will preprocess the data then feed it to the model.

```
fit(neural_workflow,data=training(split_data)) -> neural_fit
```

This will take a couple of minutes to complete. Once complete we can see the fitted model with

```
neural_fit
```

You should see that a load more parameters have now been set because the model and the pre-processing have been finalised.

Evaluating the Model

We can now use the model to make predictions on our testing data to see how well it is performing. As before, the predict function only returns the predictions, so we need to bind the results to the training data itself so we can see the predictions alongside the known correct values.

```
predict(neural_fit, new_data=testing(split_data)) %>%
  bind_cols(testing(split_data)) %>%
  select(.pred_class, transmembrane) -> neural_predictions
```

You can look at the contents of the `neural_predictions` variable to get an idea of how well it did.

Now we can calculate some of the standard metrics from this. We can make up a simple confusion table.

```
neural_predictions %>%
  group_by(.pred_class,transmembrane) %>%
  count()
```

..or if we want to be fancier…

```
neural_predictions %>%
  group_by(.pred_class,transmembrane) %>%
  count() %>%
  pivot_wider(
    names_from=.pred_class,
    values_from=n,
    names_prefix = "predicted_"
  ) %>%
  rename(true_transmembrane=transmembrane)
```

We can also calculate the specific metrics

```
neural_predictions %>%
  metrics(transmembrane, .pred_class)


neural_predictions %>%
  sens(transmembrane, .pred_class)


neural_predictions %>%
  spec(transmembrane, .pred_class)
```

Additional Exercise: Tuning models

For a final more challenging exercise we are going to try to rerun the transmembrane data but this time using a k-nearest neighbour model. As well as changing the model type we will also try to optimise the number of nearest neighbours to use.

The preparation of the data will be the same as before initially, but then we will hit some changes.

For the model you are going to use a knn model, and let the number of neighbours be a tuneable parameter

```
nearest_neighbor(neighbors = tune(), weight_func = "triangular") %>%
  set_mode("classification") %>%
  set_engine("kknn") -> model
```

For the data you need to build a 10 fold cross validation split of the full dataset, rather than a single 80% split.

```
vfold_cv(
  data,
  v=10
) -> vdata
```

You can then build a workflow from the model and data using the same formula as before.

Once you have the workflow you can look at the tuneable paramters.

```
workflow %>%
  extract_parameter_set_dials()
```

…and from these we want to change the neighbors parameter to run from 1 to 50

```
workflow %>%
  extract_parameter_set_dials() %>%
  update(
    neighbors = neighbors(c(1,50))
  ) -> tune_parameters
```

We're then going to run the workflow generating a regular grid of 20 samples over the 1-50 range. We are going to measure both the sensitivity and specificity of the model.

```
workflow %>%
  tune_grid(
    vdata,
    grid = grid_regular(tune_parameters, levels=20),
    metrics = metric_set(sens,spec)
  ) -> tune_results
```

Finally we can plot out the tuned results to see which value for k we think is best.

```
autoplot(tune_results)
```