BioTrain.TV

#Babraham Bioinformatics

Exercises: Introduction to Machine Learning

Version 2026-05
(reduced)

Licence

Exercise 1: 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

Open RStudio and create a new project. Save in the same space as you have the data saved.

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**

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.

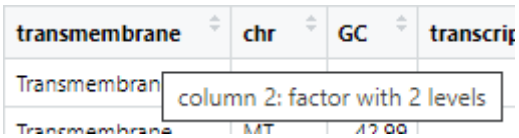
Preparing the data for modelling

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



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
```

You may have to run the following if there is an error message requiring package installs.

```
install.packages("ranger")
```

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
```

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)

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.

Exercise 2: Evaluating the predictions

From the set of predictions created in the previous exercise, we can now see how well the model actually did by comparing the predictions to the known true values and calculating evaluation metrics.

Firstly, we can get 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 Cohen'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.

Additional task: Feature importance

If you have finished the above tasks and had a play with the model setup to see the impact of changing different parameters, below is an extra task that you can have a look at. This gives less hints so you may need to utilise the TidyModels and/or specific function help documents (e.g. by using `?vip`).

A benefit of using a random forest model is that we can pull out feature importance from the model itself, to see which predictors are most important in the classification. There are a few ways that this can be pulled out and visualised.

Use the `vip()` function from the vip package to produce a Variable Importance Plot (vip) showing the top 20 most important predictors, based on impurity. *Hint: you will have to set the importance measure within your forest model.*

Use the `importance()` function in the ranger package to extract the importance values for each predictor.

Additional Exercise: 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 then integrate this into a workflow to run it. Recipes and workflows allow for increased reproducibility when there are multiple pre-processing steps that need to be repeated across several data subsets and new datasets. The Recipes package in TidyModels has a large number of pre-defined recipe 'steps' that can be added, which cover most standard pre-processing requirements. A list of available steps can be found here, but we will include what we need for the example below.

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
```

You may have to run the below for this to work, install any additional packages if prompted:

```
install.packages("brulee")
```

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)
```