Using a Neural Network to Predict Voter Preferences

Gustavo Caffaro Jun 2, 2020 · 7 min read

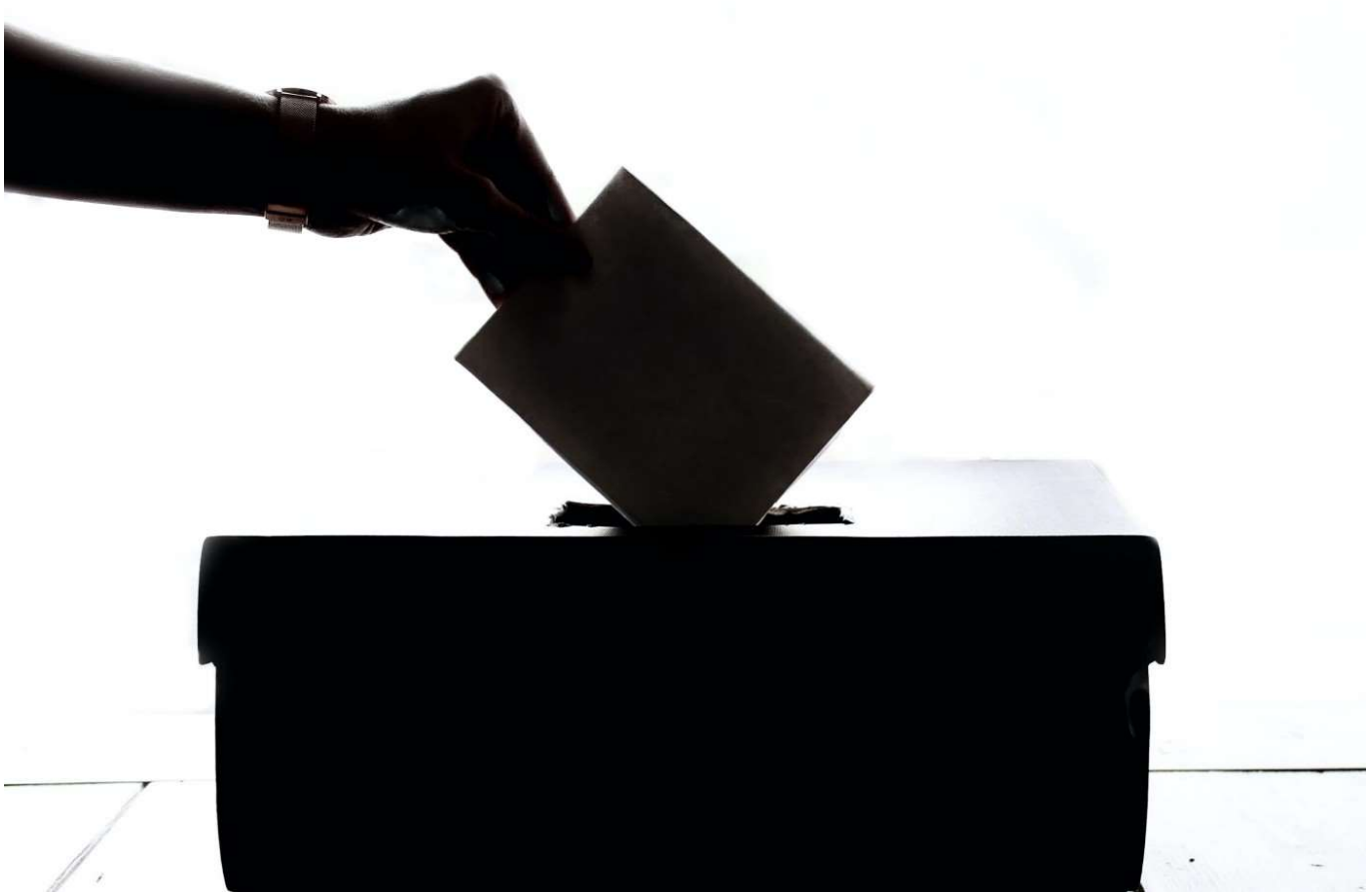

Photo by Element5 Digital on Unsplash

With presidential elections around the corner, political analysts, forecasters, and other interested parties are running to build their best estimate of election outcomes. Traditionally, polls have been used to gauge the level of popularity of political candidates, but increased computing power and the development of powerful statistical methods provide an interesting alternative to them. A good place to start forecasting elections is to first predict voters' political preferences. This is is what we will do here.

In this article we will build a simple neural network in R to predict voter preferences in the United States.

We will do this using Keras, an amazing open-source API that allows you to run neural network models in a simple yet powerful way. Although it runs natively in Python, RStudio has developed a package that allows seamless integration with R.

Before we start, please make sure you have the following R packages installed, as we will be using them to perform our predictions:

```
install.packages("keras")
install.packages("tidyr")
install.packages("ggplot2")
install.packages("dplyr")
install.packages("fastDummies")
```

Data

The data used to train the neural network comes from the 2018 Cooperative Congressional Election Study, administered by YouGov. It was compiled by Kuriwaki (2018), and was extracted from the Harvard Dataverse. You can download this data in .Rds file format for the years 2006–2018 here.

Assuming you downloaded the file and placed it in your working directory, we can proceed to import the data and see its structure:

```
d <- readRDS("cumulative_2006_2018.Rds")
dim(d)

[1] 452755     73
```

`d` is a dataframe with 452,755 rows (observations) and 73 columns (features). These features include geographic, demographic, and economic variables, in addition to other interesting variables such as political approval and news interest levels. Of course, they also include the presidential vote choice of each individual. This last variable will be our dependent variable, e.g. what we will predict with our model. Finally, a detailed explanation of each variable is provided in the aforementioned dataset source link.

Since we have data for the years 2006–2008, let's filter `d` to select only the responses made in 2018, the year of the latest survey:

```
dd <- d %>%
  filter(year == 2018)
```

Additionally, let's select only the variables that are of interest to our model and exclude missing values:

```
dd <- dd %>%
  select(st, dist, cong, # geography
         gender, birthyr, age, educ,
         race, faminc, marstat, # demographics
         newsint, # news interest
         approval_pres, # approval
         ideo5, # ideology
         voted_pres_16 # presidential vote
  )
dd <- dd[complete.cases(dd),]
```

This is how this data frame looks like:

The table below presents the number of respondents for each of the categories of voter preferences (variable `voted_pres_16`). It can be seen that around 88% of the respondents voted for either Donald Trump or Hilary Clinton, 9.91% voted for another candidate, and around 1.3% did not reveal their preferences or did not vote.

Coming back to our dataset `dd`, excluding the variable `age`, all our features are categorical. Thus, we need to one-hot encode these variables into dummies. There are many packages and functions to do this, but here we will use the function `dummy_cols` from the `fastDummies` package.

```
cat_vars <- colnames(dd)
cat_vars <- cat_vars[!cat_vars %in% c("age","voted_pres_16")]
all_data <- dummy_cols(dd,
                       select_columns = cat_vars,
                       remove_first_dummy = TRUE,
                       remove_selected_columns = TRUE)
```

We also convert our dependent variable, `voted_pres_16` into a numeric vector with integers (starting at zero) for each of the candidates, and we remove the variable `voted_pres_16` from our dataframe:

```
all_labels <- dd$voted_pres_16 %>%
  as.factor() %>%
  unclass() - 1

all_data <- all_data %>%
  select(-voted_pres_16) %>%
  as.matrix()
```

Finally, we separate our data into a training set (90%) and a test set (10%), so that after we train our model, we can test its performance on "new" data.

```
party_levels <- levels(all_labels)
elems <- sample(1:nrow(all_data),
                round(0.1*nrow(all_data)),
                replace = F)

# training data
train_data <- all_data[-elems,]
train_labels <- all_labels[-elems]

levels(train_labels) <- levels(all_labels)

# test data
test_data <- all_data[elems,]
test_labels <- all_labels[elems]

levels(test_labels) <- party_levels
```

Building the model

Our problem at hand is modeled as a classification problem, where each candidate on Table 1 represents a classification category (in total 5 categories). The input layer is formatted such that each of the 148 explanatory variables feeds a neuron of the input layer. These neurons are then connected to other neurons in the hidden layer. In this

example, we use 100 neurons for the hidden layer. Finally, the output layer has 5 units, one for each category. Figure 1 contains a graphical description of this neural network.

Figure 1. Representation of our neural network

Thus, we define our model:

```
# Define model
model <- keras_model_sequential()
model %>%
  layer_dense(units = 100, activation = "relu",
              input_shape = dim(train_data)[2]) %>%
  layer_dense(units = length(party_levels), activation = 'softmax')
```

The activation function for the first stage (input to hidden layer) is Rectified Linear Unit, or ReLu, while the activation function for the second stage (hidden to output layer) is softmax.

We now proceed to compile and train the model. The optimizer algorithm that we will use here is the *adam*, an adaptative optimization algorithm usually used to train deep neural networks. The loss function used is the *sparse categorical crossentropy*. Finally, we will take around 20% of our training data for the model to iteratively calculate the validation error.

```
###### Compile the model ######
model %>% compile(
  optimizer = 'adam',
  loss = 'sparse_categorical_crossentropy',
  metrics = c('accuracy')
)

###### Train the Model #####
early_stop <- callback_early_stopping(monitor = "val_loss",
                                      patience = 20)

model %>% fit(train_data,
              train_labels,
              validation_split = 1/5,
              callbacks = list(early_stop),
              epochs = 500)
```

The above algorithm will fit our neural network for 500 epochs, and it will stop before that if test model performance does not increase for 20 continuous epochs.

Model Performance

After training our model, we want to evaluate it using our test data by making predictions and looking at model performance:

```
###### Evaluate Model ######
score <- model %>% evaluate(test_data, test_labels, verbose = 0)

cat('Test loss:', score$loss, "\n")
cat('Test accuracy:', score$acc, "\n")
```

Model Performance

Test loss and accuracy of 0.4882 and 0.8461, respectively! Not bad!

Even so, we would now like to take a look at where our model failed. A detailed presentation of the performance of the model is found in Figure 2.

Figure 2. Confusion Matrix

The image above contains a confusion matrix of the performance of our model. Correct classification rates (high accuracy on the diagonal and low values on the off-diagonal) are colored green, while incorrect classification rates (low values on the diagonal and high values on the off-diagonal) are colored red.

A close look at the confusion matrix shows that the model made no correct predictions to the categories of “Did not vote” and “Not sure/Don’t recall”. This is due to the small number of observations that belonged to these categories relative to other categories: the number of respondents that answered “did not vote” or “not sure/don’t recall”, represent only 0.86% and 0.43% of the total sample, respectively (Table 1). Therefore,

more information is required in order to accurately predict these categories: it is not enough to have information of the voter's political and ideological preferences in order to know if the respondent *did not vote* or if they *do not recall* for whom they voted.

Additionally, it may seem surprising that the model had a very poor performance assigning voters to the "others" category (a very poor 14.1% accuracy). This is especially true since about 9.91% of all observations belong to this category (see Table 1). Nonetheless, it is important to notice that this category includes a very diverse pool of presidential candidates, such as Gary Johnson (Libertarian Party) and Jill Stein (Green Party). These candidates have diverse political ideologies, and represent a heterogenous mix of voter preferences and demographics. Therefore, we may argue that it is actually expected that the model is unable to accurately predict any votes to presidential candidates that fall into this category.

So, we built a model that predicts voter preferences. How do we forecast the outcome of an election?

This is a significantly harder task and is out of the scope of this article, but a good start is training the model we developed here with poll data, and use data from an electoral roll to predict the political preferences of a given State or the whole US population.

I hope you enjoyed this post and if you did, please let me know!

Kuriwaki, Shiro, 2018, "Cumulative CCES Common Content (2006–2018)", https://doi.org/10.7910/DVN/II2DB6, Harvard Dataverse, V4

Neural Networks US Elections Keras R Predictions