Machine Learning for Automated Coding

This notebook gives a simple example of using supervised machine learning to automatically code the topic of a patent abstract. There's a lot of code here – feel free to copy and paste much of it as you work on the exercises and you do your assignment. This is also only barely scratching the surface, and intended to be as transparent of an introduction as possible in terms of showing how everything is done. Of note, the `caret` package in R can be extremely helpful in doing some of what's done here with much less code involved (see https://www.machinelearningplus.com/machine-learning/caret-package/ for more information). However, there's somewhat of a learning curve associated with it, as well as the inability to see how everything works, so I only use the `caret` package to do a few things, like calculating precision and recall.

```
# If you have not installed these, make sure to install first!

# Tidyverse and text analysis tools
library(tidyverse)
#library(tidyr)
library(tidytext)

# For decision trees
library(rpart)
library(rpart.plot)
library(rattle)

# For nearest neighbors
library(class)

# For ML evaluation
library(caret)
```

Data

We scraped the data from PatentsView, then created labels of whether or not the patent was about cell biology or not.

> The labels here were generated using Latent Dirichlet Allocation, a form of unsupervised learning, to assign topics based on abstracts, so that I could easily assign a label to each one. In your assignments, you'll be using hand-coded sentiments as the labels in the data that you work with.

```
umd_abstracts <- read.csv('C:/Users/32636/Desktop/2023Winter_UM/Surv622/ass2_Using the Reddit API and S
rename(patent_abstract=text,
       label=Sent) %>%
  mutate(label=ifelse(label=='Positive','True','False'))

umd_abstracts$ID <- as.character(umd_abstracts$ID) # Need to make this character for later joins
glimpse(umd_abstracts)
```

```
## Rows: 203
## Columns: 6
## $ X              <int> 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61~
```

```
## $ patent_abstract <chr> "Build guide for Dwarf Fortress", "How would people fe~
## $ timestamp       <int> 1678284981, 1678302801, 1678310454, 1678104476, 167813~
## $ date_utc        <chr> "3/8/2023", "3/8/2023", "3/8/2023", "3/6/2023", "3/6/2~
## $ ID              <chr> "1931", "2054", "2058", "2262", "2309", "2327", "2339"~
## $ label           <chr> "True", "True", "True", "True", "True", "True", "True"~
```

Here, we have a few key variables. The ID variable is simply numbers from 1 to 1,243, denoting the unique patent. The patent title, patent year, and patent abstract are all included. Finally, the label denotes whether the topic of the patent was related to cell biology (True) or not (False). Our goal is to build a machine learning model that takes this data and is able to predict the topic of the abstract based on the content of the abstract.

These abstracts are exactly as we got them when we pulled from the PatentsView API. We need to do some cleaning first. After we do that, let's take a look what the most common words are.

We want to do the following steps to turn this text data into features:

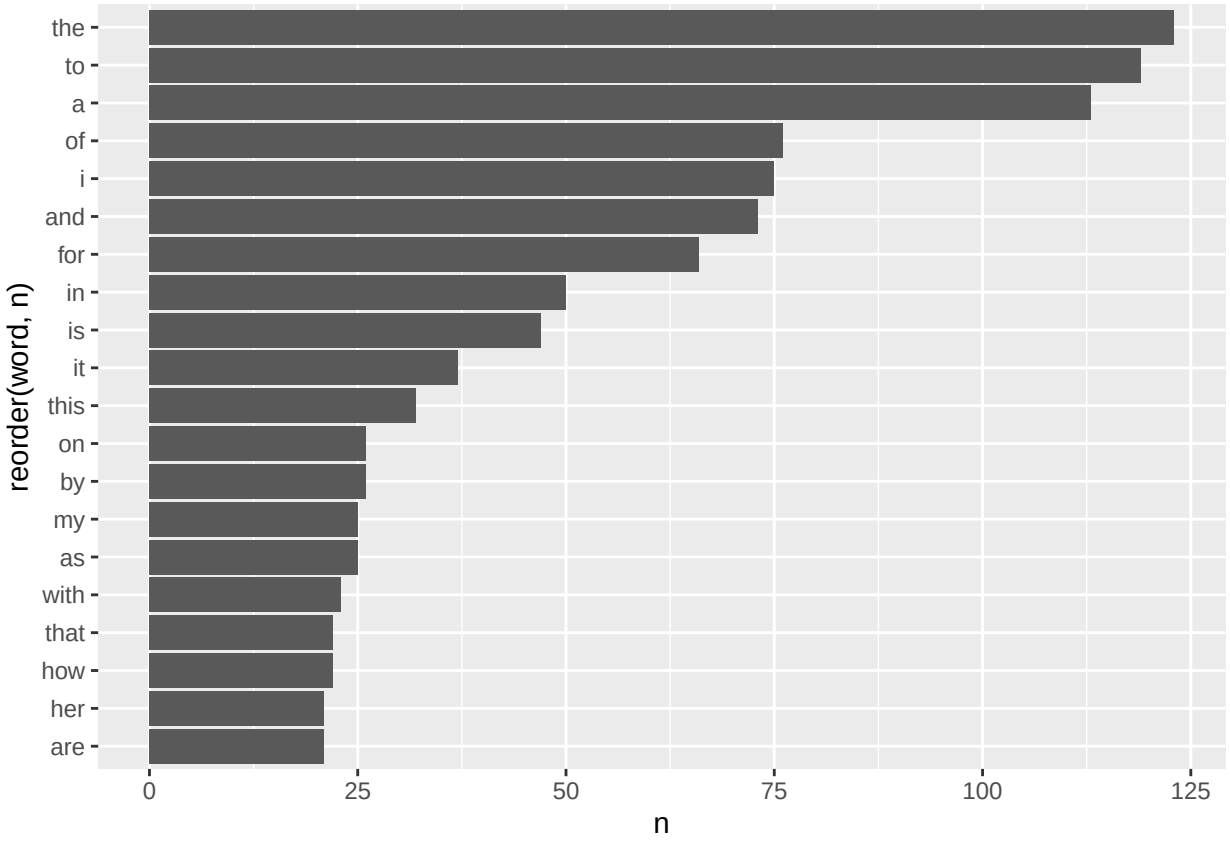
-**Tokenize:** Split up the abstracts into individual words

-**Stop Words:** Remove words that are too frequent and uninformative, like "a", "an", and "the".

-**Bag of Words:** We want columns representing all words in the entire corpus, and the rows representing abstracts, with each cell indicating the counts of words in that abstract.

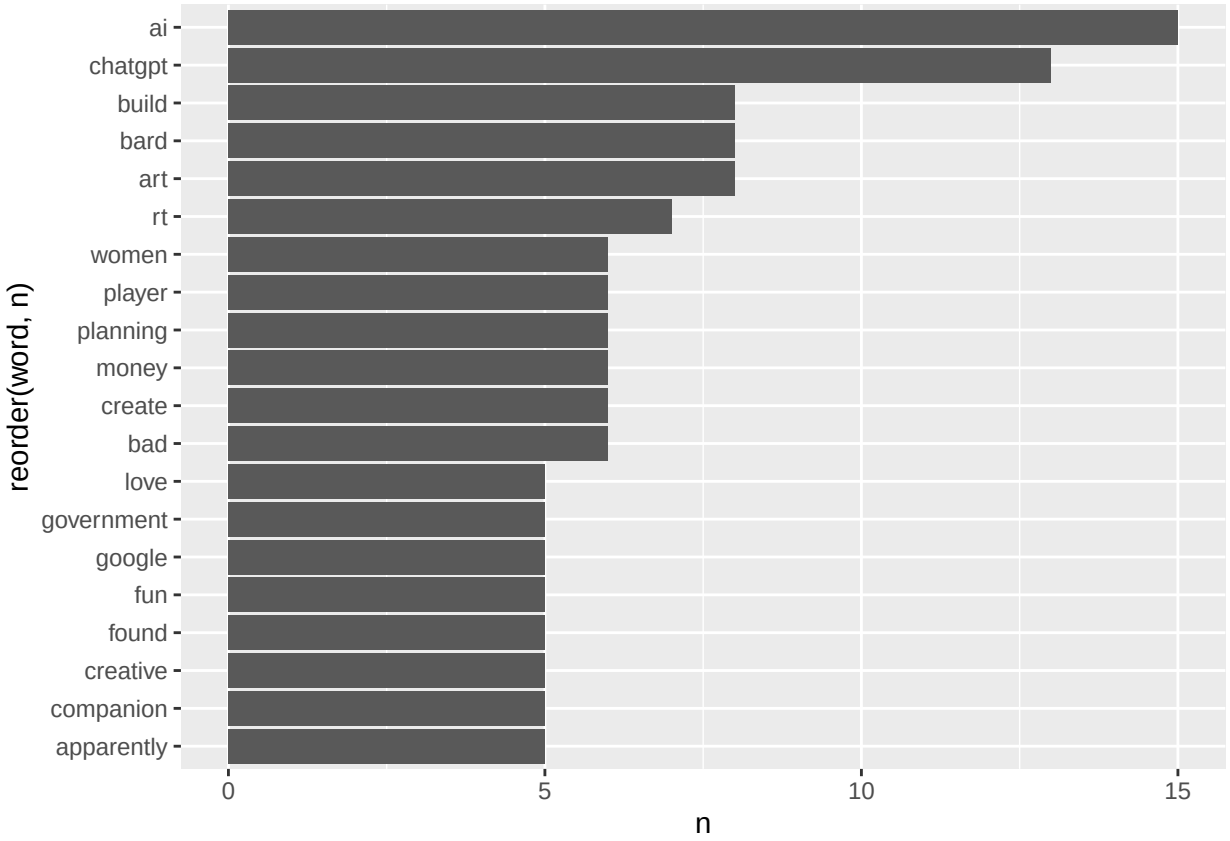
We'll look at the distribution of words as we go to inform how we're doing. Let's first tokenize and then look at what words in our corpus. First, let's tokenize and count the number of instances of each word. The key function we're using here is `unnest_tokens`, which we're using the break up the big abstract string for each patent into individual strings with just one word each.

```
umd_abstracts %>%
  unnest_tokens(word, 'patent_abstract') %>%  # tokenize
  count(word, sort = TRUE) %>% # count by word
  arrange(desc(n)) %>% # Everything from this point on is just to graph
  head(20) %>%
  ggplot(aes(x = reorder(word, n), y = n)) +
  geom_bar(stat= 'identity') +
  coord_flip()
```



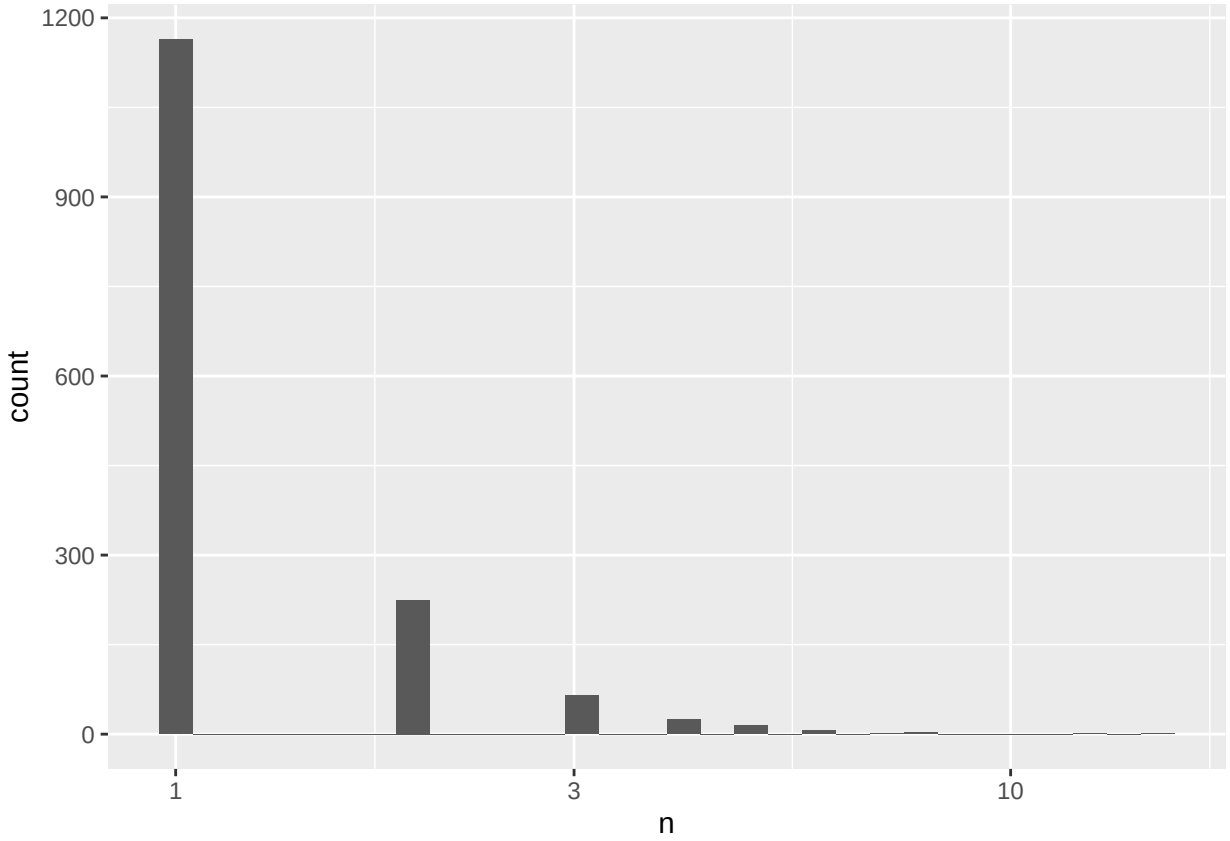
Looks like there are a lot of stop words being caught! Let's take those out. To do this, we use the `stop_words` from the `tidytext` package and use an `anti_join` to remove all instances of that word.

```
umd_abstracts %>%
  unnest_tokens(word, 'patent_abstract') %>%  # tokenize
  anti_join(stop_words)%>% # Remove stop words
  count(word, sort = TRUE) %>% # count by word
  arrange(desc(n)) %>% # Everything from this point on is just to graph
  head(20) %>%
  ggplot(aes(x = reorder(word, n), y = n)) +
  geom_bar(stat= 'identity') +
  coord_flip()
```



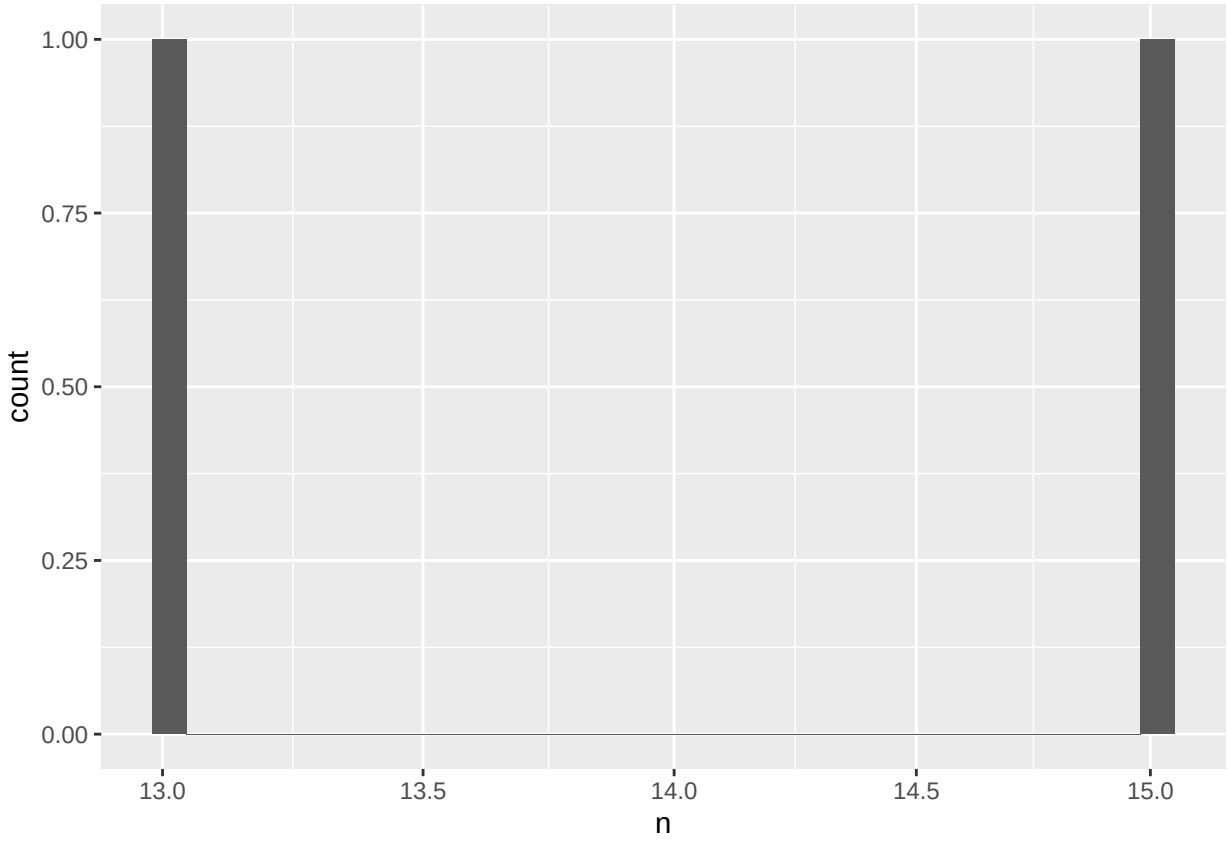
That looks better! Let's also take a look at the distribution of word counts by using a histogram on a log scale.

```
umd_abstracts %>%
  unnest_tokens(word, 'patent_abstract') %>%  # tokenize
  anti_join(stop_words)%>% # Remove stop words
  count(word, sort = TRUE) %>%
  ggplot(aes(n)) +
  geom_histogram() +
  scale_x_log10()
```



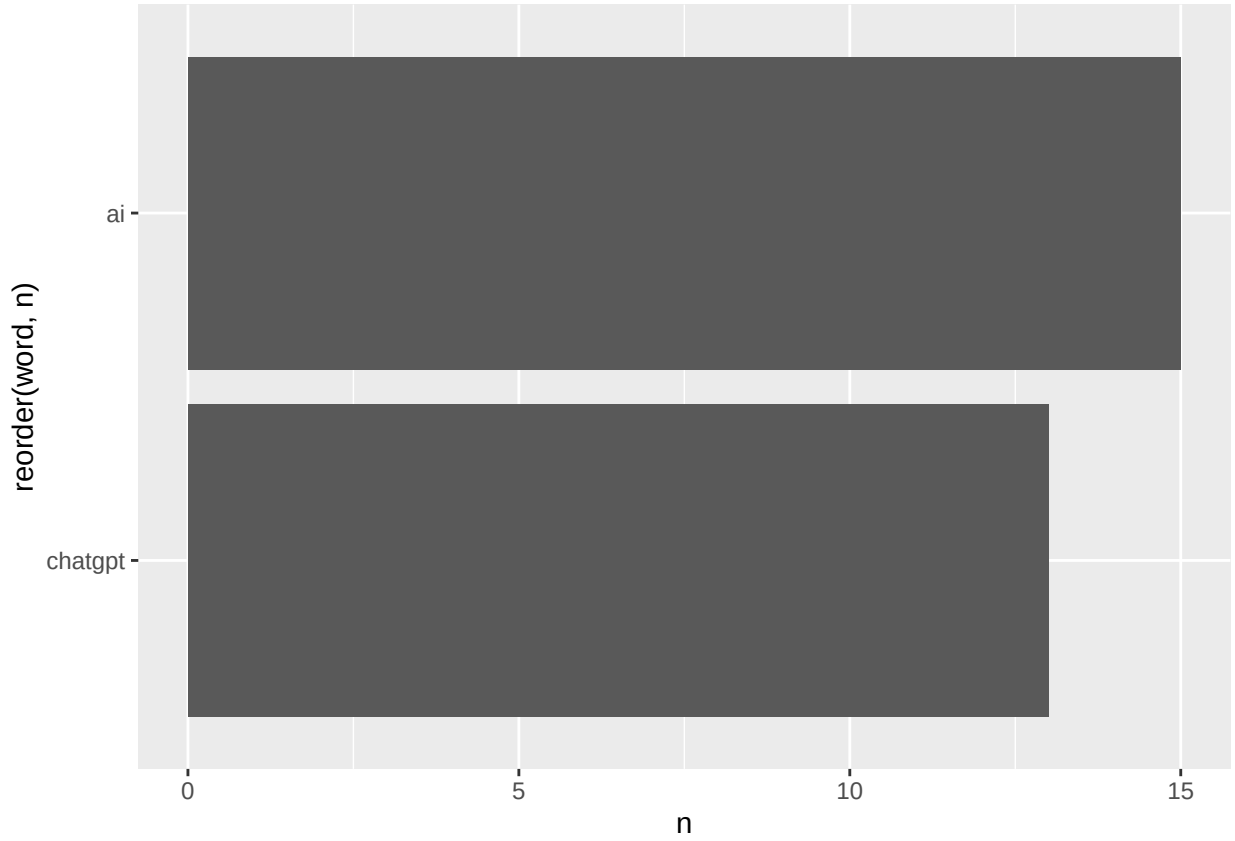
It looks like we have a lot of words that only happen once, or otherwise very infrequently. We'll also remove some of the most infrequent words, as they are likely typos, or are so rare that they are not useful.

```
umd_abstracts %>%
  unnest_tokens(word, 'patent_abstract') %>%  # tokenize
  anti_join(stop_words)%>% # Remove stop words
  count(word, sort = TRUE) %>% # count by word
  filter(n >= 10) %>%  # Remove words that occur less than 10 times
  ggplot(aes(n)) +
  geom_histogram() +
  scale_x_log10()
```



Now that we've explored the data a little bit and know what steps we should take to clean it up, we can create our features. From this exercise, we know we need to tokenize, remove stop words, and remove infrequent words. We can also take additional steps at this stage to clean up the data a bit more. For example, we might consider removing all numbers or digits. We can think about stemming in order to group similar words together under a single root (e.g., invent, invention, inventor). If we were to stem (this step is optional, and it's actually possible your models run better without stemming), it might look something like this:

```
# NOTE: This uses the corpus package, which we did not bring in at the beginning
umd_abstracts %>%
  unnest_tokens(word, 'patent_abstract') %>%  # tokenize
  anti_join(stop_words)%>% # Remove stop words
  mutate(word = corpus::text_tokens(word, stemmer = "en") %>% unlist()) %>% # add stemming process
  count(word, sort = TRUE) %>% # count by word
  filter(n >= 10) %>%  # Remove words that occur less than 10 times
  arrange(desc(n)) %>% # Everything from this point on is just to graph
  head(20) %>%
  ggplot(aes(x = reorder(word, n), y = n)) +
  geom_bar(stat= 'identity') +
  coord_flip()
```



Creating Features

To create the features for our machine learning model, we will take all of the words in our corpus and count the number of times that they appear in the abstract. We will end up with a sparse data frame, with the columns representing each word, and rows representing the patent.

We'll start by extracting out all of the words that we want to include. We'll exclude all words that occur less than ten times, because we want to avoid using extremely rare words since they are likely not informative and could contain lots of typos. We'll also remove stop words.

```
word_list <- umd_abstracts %>%
  unnest_tokens(word, 'patent_abstract') %>%
  anti_join(stop_words)%>%
  count(word, sort = TRUE) %>%
  filter(n >= 5)%>%
  pull(word)
```

Next, we'll generate counts of words that each patent abstract contains. To do this, we'll take the bag of words for each abstract, then find which words occur how many times, and include that in our final dataset.

There's a lot of steps here, so feel free to explore the code below by looking each step one by one, split up conveniently for you line by line.

```
patent_features <- umd_abstracts %>%
  unnest_tokens(word, 'patent_abstract') %>%
  anti_join(stop_words)%>%            # remove stop words
  filter(word %in% word_list) %>%     # filter for only words in the wordlist
  count(ID, word) %>%                 # count word useage by abstract
```

```
  spread(word, n) %>%                    # convert to wide format
  map_df(replace_na, 0)                  # replace NA with 0 to create dataset
```

Now, we need to join this back with our original dataset, using ID, to get the labels matched up with our features.

```
full_data <- umd_abstracts %>%
  right_join(patent_features, by = 'ID') %>%
  select(-X,-timestamp, -date_utc,-patent_abstract) # Remove extra variables
```

The full data contains the ID variable (which we will use to make our train/test split in the next section), as well as our features (each word) and the label (True/False for whether the topic was on cell biology or not)

Here, you might consider doing a bit more feature engineering and data manipulation. For example, you might consider scaling the variables, in order to avoid the influence of more frequent words. You can try cleaning the text data a bit more, to remove certain words that might be stop words in this specific context. You can also consider adding additional variables, such as length of abstract in number of words.

Train and Test Split

For simplicity, we'll consider a simple holdout sample. At the end, we show how to do cross validation using the `caret` package in R. The cross validation code will be very similar to this, except repeated for multiple combinations of training and testing data.

```
# 30% holdout sample
test <- full_data %>% sample_frac(.3)

# Rest in the training set
train <- full_data %>% anti_join(test, by = 'ID') %>% select(-ID)

# Remove ID after using to create train/test
# We don't want to use ID to run models!
test <- test %>% select(-ID)
```

Fitting Models

Now, we can fit some machine learning models. We'll do some simple ones here: K-Nearest Neighbors and Decision Trees. You can also use Logistic Regression, or Naive Bayes, or Support Vector Machines, or any number of other, more complicated models, though we won't cover them here. If you are familiar with ensemble models, such as Random Forests, I'd suggest trying those out as well.

First attempt at a model

Let's start with a K-Nearest Neighbors model. This simply checks the class of closest k neighbors, and takes a vote of them to predict what the class of the data point will be. We can fit this model using the `class` package.

```
# Create separate training and testing features and labels objects
train_features <- train %>% select(-label)
test_features <- test %>% select(-label)

train_label <- train$label
test_label <- test$label

# Predicted values from K-NN, with K = 11
knnpred <- knn(train_features,test_features,train_label, k = 7)
```

The `knnpred` object has the predicted values for each of the `test_features` that we gave it. Let's take a look at what the predicted values are. We'll put the predicted values in a data frame with the actual values.

```
pred_actual <- data.frame(predicted = knnpred, actual = test_label)
pred_actual %>% head()
```

```
##   predicted actual
## 1      True   True
## 2      True   True
## 3      True   True
## 4      True   True
## 5      True  False
## 6      True  False
```

Now that we have the predicted and actual values in one data frame, we can create a confusion matrix and evaluate how well our model is performing.

```
pred_actual %>% table()
```

```
##          actual
## predicted False True
##     False     1    0
##     True      9   17
```

```
confusionMatrix(pred_actual %>% table(), positive = 'True')
```

```
## Confusion Matrix and Statistics
##
##          actual
## predicted False True
##     False     1    0
##     True      9   17
##
##                Accuracy : 0.6667
##                  95% CI : (0.4604, 0.8348)
##     No Information Rate : 0.6296
##     P-Value [Acc > NIR] : 0.427897
##
##                   Kappa : 0.1227
##
##  Mcnemar's Test P-Value : 0.007661
##
##             Sensitivity : 1.0000
##             Specificity : 0.1000
##          Pos Pred Value : 0.6538
##          Neg Pred Value : 1.0000
##              Prevalence : 0.6296
##          Detection Rate : 0.6296
##    Detection Prevalence : 0.9630
##       Balanced Accuracy : 0.5500
##
##        'Positive' Class : True
##
```

Note that we don't actually see the words "precision" or "recall" here – instead, we can find them by their alternate names: sensitivity (for recall) and positive predictive value (for precision). We can also use the

`precision` and `recall` functions (also in the `caret` package). Note that we use `relevant` to specify which outcome we're trying to predict (similar to the `positive` argument above).

Running a Decision Tree model

With the training and testing datasets that we've created, running the actual tree model is actually quite simple. If you have used `R` for running linear models before, the format is very similar.

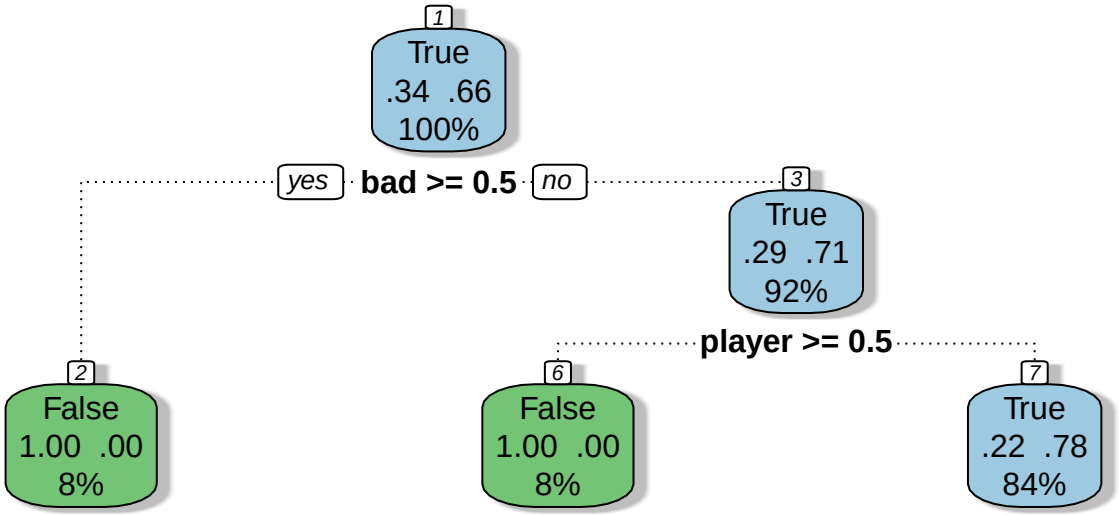
```
treemod <- rpart(label ~ .,
                 data = train,
                 method = 'class',
                 control = rpart.control(minsplit = 15))
```

Let's break down each of the arguments in this function. First, we specify the model, putting the label that we want to predict on the left side of the "~" and all the features we want to include on the right. We include arguments for the dataframe from which we're taking the data, and the tree method we want to use (in this case, since we are doing a classification tree, we use 'class'). Then, we can use the `control` argument to set decision tree parameters. In this case, we are setting the minimum number of observations needed in a node to add a split.

We have stored the model in the `treemod` object. Let's look at what the model gave us. We can use summary to look at the summary of the model, but it might be easier to look a visualization instead.

```
# You can try running the summary, but it will give a LOT of output
# summary(treemod)

# The fancy tree visualization
fancyRpartPlot(treemod, sub = "")
```



Evaluating the Model

Now that we have a model, we need to test it. We can get predictions using the `predict` function.

```
pred <- predict(treemod, test)
head(pred)
```

```
##       False      True
## 1 0.2222222 0.7777778
## 2 0.2222222 0.7777778
## 3 0.2222222 0.7777778
## 4 0.2222222 0.7777778
## 5 0.2222222 0.7777778
## 6 0.2222222 0.7777778
```

Note that this gives us the prediction scores for both cell biology-related patents and not cel biology-related patents. They are basically the same thing, since you can get one from taking one minus the other. We'll focus on the "True" case, since we want to identify the patents most likely to be related to cell biology. Let's create a dataframe with the scores and actual values.

```
test_pred <- data.frame(score = pred[,2], actual = test$label)
head(test_pred)
```

```
##       score actual
## 1 0.7777778   True
## 2 0.7777778   True
## 3 0.7777778   True
## 4 0.7777778   True
## 5 0.7777778  False
## 6 0.7777778  False
```

We need a way to convert from the scores to an actual prediction. Typically, we use some sort of arbitrary value as a cutoff, or take a certain percentage of the most likely observations (for example, the top 10% most likely are predicted to be related to cell biology).

We'll go with the latter for this example. Let's go with a 20% threshold.

```
test_pred <- test_pred %>% arrange(desc(score))
test_pred$pred <- 'False'

top_scores <- floor(nrow(test_pred)*0.2)
test_pred$pred[1:top_scores] <- 'True'
```

We can get the values of precision and recall using `confusionMatrix` function from the `caret` package. First, we create a table with the confusion matrix, then run the function with the table as the argument. Note that we specify what the `positive` value is – since we are trying to predict what patents are about cell biology, we have 'True' as our positive value. In addition, make sure your predicted values are first in the table, or else you'll get the opposite results as you want! We fix this in our table by using `select`.

```
pred_table <- test_pred %>% select(pred, actual) %>% table()
confusionMatrix(pred_table, positive = 'True')
```

```
## Confusion Matrix and Statistics
##
##        actual
## pred    False True
##   False     9   13
##   True      1    4
##
```

```
##                Accuracy : 0.4815
##                  95% CI : (0.2867, 0.6805)
##     No Information Rate : 0.6296
##     P-Value [Acc > NIR] : 0.961543
##
##                   Kappa : 0.1085
##
##  Mcnemar's Test P-Value : 0.003283
##
##             Sensitivity : 0.2353
##             Specificity : 0.9000
##          Pos Pred Value : 0.8000
##          Neg Pred Value : 0.4091
##              Prevalence : 0.6296
##          Detection Rate : 0.1481
##    Detection Prevalence : 0.1852
##       Balanced Accuracy : 0.5676
##
##        'Positive' Class : True
##
```

```
precision(pred_table, relevant = 'True')
```

```
## [1] 0.8
```

```
recall(pred_table, relevant = 'True')
```

```
## [1] 0.2352941
```

Looping through models

We've shown an example of two different types of models: K-Nearest Neighbors and Decision Trees. In the real world, we would want to try many different models with many different values for the tuning parameters (e.g., K for K-NN, max depth for trees). Using a loop, we can automatically run through many different models very quickly. Below is an example of trying many different Decision Tree models.

```
# We will look at minsplit values of 5, 10, 15, 20
splits <- c(5,10,15,20)

# We'll look at maxdepths of 2, 3, 4, 5
depths <- c(2,3,4,5)

# We'll consider predicting the top 5%, 10%, and 20% as cell biology
percent <- c(.05, .1, .2)

# How many different models are we running?
nmods <- length(splits)*length(depths)*length(percent)

# We will store results in this data frame
results <- data.frame(splits = rep(NA,nmods),
                      depths = rep(NA, nmods),
                      percent = rep(NA,nmods),
                      precision = rep(NA,nmods),
                      recall = rep(NA,nmods))

# The model number that we will iterate on (aka models run so far)
```

```r
mod_num <- 1

# The loop
for(i in 1:length(splits)){
  for(j in 1:length(depths)){
    s <- splits[i]
    d <- depths[j]
    # Running the model
    treemod <- rpart(label ~ .,
                     data = train,
                     method = 'class',
                     control = rpart.control(minsplit = s, maxdepth = d))

    # Find the predictions
    pred <- predict(treemod, test)

    # Attach scores to the test set
    # Then sort by descending order
    test_pred <- data.frame(score = pred[,2], actual = test$label)
    test_pred <- test_pred %>% arrange(desc(score))

    # Make predictions based on scores
    # We loop through each threshold value here.
    for(k in 1:length(percent)){
      p <- percent[k]

      # Predict the top % as True
      test_pred$pred <- 'False'
      top_scores <- floor(nrow(test_pred)*p)
      test_pred$pred[1:top_scores] <- 'True'

      # Confusion Matrix
      pred_tab <- test_pred %>% select(pred, actual) %>% table()

      # Store results
      results[mod_num,] <- c(s,
                             d,
                             p,
                             precision(pred_tab, relevant = 'True'),
                             recall(pred_tab, relevant = 'True'))
      # Increment the model number
      mod_num <- mod_num + 1
    }
  }
}

# All results are stored in the "results" dataframe
head(results)
```

```
##   splits depths percent precision     recall
## 1      5      2    0.05       1.0 0.05882353
## 2      5      2    0.10       1.0 0.11764706
## 3      5      2    0.20       0.8 0.23529412
## 4      5      3    0.05       1.0 0.05882353
```

```
## 5      5      3    0.10       1.0 0.11764706
## 6      5      3    0.20       0.8 0.23529412
```

```
# Best recall? Top 5 in descending order
results %>% arrange(desc(recall)) %>% head()
```

```
##   splits depths percent precision    recall
## 1      5      2     0.2       0.8 0.2352941
## 2      5      3     0.2       0.8 0.2352941
## 3      5      4     0.2       0.8 0.2352941
## 4      5      5     0.2       0.8 0.2352941
## 5     10      2     0.2       0.8 0.2352941
## 6     10      3     0.2       0.8 0.2352941
```

```
# Best precision? Top 5 in descending order
results %>% arrange(desc(precision)) %>% head()
```

```
##   splits depths percent precision     recall
## 1      5      2    0.05         1 0.05882353
## 2      5      2    0.10         1 0.11764706
## 3      5      3    0.05         1 0.05882353
## 4      5      3    0.10         1 0.11764706
## 5      5      4    0.05         1 0.05882353
## 6      5      4    0.10         1 0.11764706
```

You can also loop through many different values of K for K-NN, and also different tuning parameters for other types of models. For example, for Logistic Regression, you might consider an L1 or L2 penalty (Lasso or Ridge Regression), with different values for the penalty parameter.

Simplifying the Process with Caret

We've already used the `caret` package for tools like the confusion matrix and precision/recall. However, we can also use it to do other validation methods more easily, such as k-fold cross validation. Here, we'll look at a quick example of using `caret` to find the training and test sets need to do 10-fold cross validation.

```
# Create a list of 10 folds (each element has indices of the fold)
flds <- createFolds(full_data$ID, k = 10, list = TRUE, returnTrain = FALSE)
str(flds)
```

```
## List of 10
##  $ Fold01: int [1:4] 23 38 69 72
##  $ Fold02: int [1:12] 3 4 14 15 40 44 48 54 56 61 ...
##  $ Fold03: int [1:7] 6 17 19 45 52 74 75
##  $ Fold04: int [1:10] 9 11 28 32 35 49 50 57 58 59
##  $ Fold05: int [1:10] 1 2 5 51 60 63 65 77 80 87
##  $ Fold06: int [1:13] 13 22 25 30 39 41 46 55 64 70 ...
##  $ Fold07: int [1:10] 7 12 21 24 29 33 34 43 47 76
##  $ Fold08: int [1:12] 8 16 27 42 53 62 67 73 79 84 ...
##  $ Fold09: int [1:7] 18 20 36 71 82 90 91
##  $ Fold10: int [1:6] 10 26 31 37 85 89
```

As you can see, we created a list of 10 vectors, each containing a fold. So, to do our 10-fold cross validation, we can take the first fold, and create our train and test sets from that, take the second fold and create test and train from that, and so on. For example, to create test and train using the first fold, we can use the following code:

```
# Create train and test using fold 1 as test
patent_test01 <- full_data[flds$Fold1,]
```

```
patent_train01 <- full_data[-flds$Fold1,]
```

You can then use these train and test sets as we have above. You should put this into a loop, and store the value of metrics such as accuracy, precision, and recall, similar to what we've done in the loop of trying out different parameters.