INFO370 Problem Set 7: Bag-of-words, and k-NN

March 6, 2020

Introduction

This problem set is about classification, prediction, and training-testing split (again!), but also about Bag-of-words (BOW), TF-IDF, and k-NN. It has the following aims:

- get some hands-on experience with text processing and BOW
- play with k-nearest neighbors
- give you some (more) experience with hyperparameter tuning
- make you to think about vectorizing your code (loops are slow!) and efficient data types.

The tough ones will also implement the TF-IDF transformation.

The task is in many ways very similar to the PS6: split data into training/validation chunks, fit the model, and compute accuracy on the validation part. However, instead of numeric data and binary outcome you are dealing with 29 different categories and text data.

Please submit a) your code (notebooks, rmd, whatever) *and* b) the results in a final output form (html or pdf).

This is groupwork.

Text data

The text dataset contains 12,924 "pages" from 29 "books". It is a csv file containing the following fields:

name name of the book, in a compact form. You can probably easily deduce the actual name.

size total book size in bytes

lines total book size in lines

pagenr current page number. These are not really pages, every text is just chopped into 25-line blocks or so. If you want to re-assemble a text, you use these numbers.

text text of the page

You only need *name* and *text* in this assignment.

The texts include books pulled from Project Gutenberg, and texts from Canterbury corpus. The former are books, the latter are various texts, including CS conference papers. The former are in pure ASCII (or rather UTF-8), some of the latter in a different format (this are the `.sh`, `.pp` and such things).

Warning This data and the task are large. When reasonably optimized, it takes 20min and at max 8GB ram on my computer. Start slow and analyze a random sample of, say, only 100 texts first. When your code works reasonably well, increase the sample as far as your computer can stand. In particular this applies to creating BOW and converting to TF-IDF (memory-hungry) and k-NN predictions (slow with certain metrics).

1 Explore the data

As the first step, explore the data.

1. Load the data. You may drop *size*, *lines*, and *pagenr*.
2. Ensure that you don't have any missing *name*, and empty *text* in your data.
3. Create a summary table where you show how many chunks of each book you have in data. Order this by size.
4. Explore the data: check out a few pages from various titles, as a minimum take a look how do a few books and a few CS papers look like.

2 First Task: Tokenize

Your first task is too create your own custom tokenizer and tokenize the texts. The tokenizer should achieve the following things:

1. convert all texts to lower case
2. remove punctuation and other weird characters. I recommend to replace these with space. This is to be done in order to make the code to recognize strings like "end" and "end." as the same word.

   Note: you are not really able to remove all the punctuation characters. But you should at least remove the most common ones. I recommend to print out a sample of your tokens, and when you see some "improper" characters, add code to remove those.
3. tokenize texts to words. If you replaced punctuation with spaces, you can just use pandas' `str.split` method.

   As a result your data should still contain the same texts, but instead of a long text string, the individual texts should be lists of tokens.
4. remove stopwords. It is up to you to decide which stopwords to remove, I recommend to include at least *the* and *a*.

Hint: use pandas' `.str` methods as much as you can. Try to avoid looping and base-python string methods.

When you are done with tokenize, it's time to create vocabulary. Vocabulary is a list of all tokens you encountered in the text. Let's agree that we order it alphabetically.

5. Create such vocabulary and order it alphabetically.

   Hint: I recommend to create an empty set and add tokens to it for each case. Check out how to add a list of elements to a set.

3 Second and the Largest Task: Implement BOW

Next, we have to convert the texts to BOW-s. This is the most complex part of the problem set, essentially you are going to write CountVectorizer from scratch. But it is not terribly complex either, you can do a well-structured BOW code in four lines.

You can proceed in many ways, I recommend the following:

1. create a data frame of zeros with column names equal to the vocabulary and as many rows as you have texts.

Note that the full data matrix (or data frame) will be 13k × 64k ≈ 800M numbers. Using default 64-bit integers (or floats), the result will be ~7GB. I strongly recommend to use *float32* datatype instead. You can also experiment with other data types (warning: they are slow).

I recommend to use data frame here instead of numpy matrix as data frame has index. Index is great if you want to access columns not by their number but by their name. So if you set columns equal to vocabulary tokens, you can refer to the columns by just using the tokens. Otherwise you have to figure what are the order number of the columns. In this way you can avoid some loops or other inefficient constructs.

2. loop over all your texts. For each text

   (a) compute the frequency of tokens in this text.
   Hint: convert the tokens list to a series, and use `.value_counts` method.

   (b) add the frequency counts to the corresponding row and columns in the data frame.
   Hint: you can select multiple columns of a data frame by specifying a list of column names. Even better, instead of list you can give a series of column names, and `.value_counts` you computed above is a series. So you can do something like this:

   ```
   df[counts.index] += counts
   ```

   This adds to each column, named in the series *counts*, a number in *counts*.

   This is pretty much it.

I recommend to start slow and instead create BOW of the tiny example *texts-test.csv* (also on canvas). It's bow should look like

```
created bow of shape (3, 13)
     i  index   is  knowing    l  loop  mastering  others    r  running  strength  tried  using  yourself
0  0.0    0.0  2.0      2.0  0.0   0.0        0.0     1.0  0.0      0.0       0.0    0.0    0.0       1.0
1  0.0    0.0  2.0      0.0  0.0   1.0        2.0     1.0  0.0      0.0       1.0    0.0    0.0       1.0
2  1.0    1.0  0.0      0.0  1.0   0.0        1.0     0.0  0.0      0.0       0.0    1.0    1.0       0.0
3  1.0    1.0  0.0      0.0  2.0   1.0        0.0     0.0  1.0      1.0       0.0    1.0    1.0       0.0
```

Congrats ☺! You are over the hard part, the rest is much more smooth sailing.

4 Model

Now the fun part. Your task is to categorize the texts using k-NN. Split the data into training-validation parts, use training to fit your k-NN, and validation to compute accuracy. Experiment with different k and different distance metrics.

1. split your data into training/testing chunks. Remember: BOW is your data matrix (design matrix) and *name* is your target variable.

2. pick a k and use cosine similarity.

   Hint: check out the arguments for `KNeighborsClassifier`. The documentation does not mention cosine similarity but *metric='cosine'* works just beautifully.

3. fit your model on training data...

4. ...and predict on validation data. Again, this may be slow, so please start small and only increase your sample size if the speed is sufficient.

5. compute accuracy on validation data. As we have 29 categories now you probably don't want to print a confusion matrix. Instead, just count the correct vs incorrect predictions.

   Hint: don't count them, just take mean of the logical variable: is the prediction correct.

6. try different k-s, and try different metrics.

   Here is a suggestion: if you want to store your accuracy values in an array together with the corresponding k-s, you can use *Series* as follows:

```
klist = [1,2,3,5,7,11]  # all the k-s you want to check here
accuracy = pd.Series(0, index=klist)  # create an empty series indexed by k-s
for k in klist:
    a = computeAccuracy(X, y, k)  # do your computations here
    accuracy[k] = a  # you just use 'k' as index here as you defined this as index!
## now you can use your series in various ways, e.g. for plotting:
plt.plot(klist, a)
```

   You can use a similar approach for metrics too, the index does not have to be numeric.

Finally, it is time to take a closer look at a few correct/incorrect predictions.

7. Print out a few correct and incorrect predictions. Can you understand what is going on?

   Note: if you do TF-IDF then answer this question after you are done with that.

5 Extra Credit: Implement TF-IDF transformation (10 EC points)

Consult lecture notes, section 6.1.4 TF-IDF. These notes follow Murphy (2012) approach. If you are using another approach (there are many ways to do TF-IDF), please cite the source, and provide the respective formulas.

1. Implement TF-IDF transformation. Try to use as much vectorized operations as you can (i.e. no loops). It only takes a few lines.

For the reference, here is TF-IDF of the tiny test dataset (the order of tokens is the same as for the BOW example):

```
0.    0.    0.316 0.761 0.    0.    0.    0.199 0.    0.    0.    0.    0.    0.199
0.    0.    0.316 0.    0.    0.199 0.316 0.199 0.    0.    0.480 0.    0.    0.199
0.199 0.199 0.    0.    0.199 0.    0.199 0.    0.    0.    0.    0.199 0.199 0.
0.199 0.199 0.    0.    0.316 0.199 0.    0.    0.480 0.480 0.    0.199 0.199 0.
```

2. repeat what you did in question 4 above above with tf-idf version of the data.

3. compare BOW and TF-IDF results. Which ones are better?

4. new it is time to do the last question of the previous section.

References

Murphy, K. P., 2012. Machine Learning: A Probabilistic Perspective. MIT Press, Cambridge, MA.